Data Management in S

John M. Chambers
AT&T Bell Laboratories, Murray Hill, New Jersey

December 23, 1991

Contents

Abstract

The details of data management in S are discussed, as implemented in the 1991 release of S. A number of extensions have been introduced since the publication of the 1988 version of S, as described in the book *The New S Language*, by R. A. Becker, J. M. Chambers and A. R. Wilk. These include:

- the ability to attach S objects as databases;
- a facility to compile S objects as static C databases;
- user-defined classes of database objects;
- explicit control over frames during evaluation;
- cleaner and more efficient techniques for data search;
- a consistent and thorough approach to backout on error.

This discussion defines the semantics of database management and evaluation frames. It outlines how to use the new features, likely to be relevant for users with substantial applications of S (particularly applications involving large amounts of data).

1 Introduction

This report outlines the semantics of data management in the 1991 version of S. A number of new concepts and extended facilities have been added since the 1988 version of S described in [1]. Specifically, there is a new model for database management that treats databases as S objects; in addition to the file-system directories traditionally used, list-like objects can be attached, databases can be compiled into C code, and users can define *classes* of database objects (e.g., to interface to database management software).

The majority of the discussion below concerns permanent data objects, organized in S into *databases*. Objects are also created and used in temporary *frames* as well. Section 2 gives the general background for both. Section 3 discusses frames as they are treated by the S evaluator. Section 4 shows how frames can be created and manipulated explicitly by user functions. The remaining sections discuss the treatment of permanent databases—the search list concept, the semantics of database management, and the various kinds of database objects. The discussion is meant to be self-contained, not relying on reading [1], except where we specifically refer to it. Material that is new since [1] will be noted in footnotes. If you don't care which features are new, you can ignore the footnotes.

2 Names and Objects

The ability to associate names with data is fundamental to any programming language. We describe here the mechanisms for this association in S. For casual use of the language, an understanding of this mechanism is usually not necessary; just assume that when you want an object, it will be found and when you assign an object, it will be put somewhere (permanently or temporarily) in an appropriate way.

We begin with some basic notions of the role of names. In the expression

```
mean(population)
```

the character string `"population"` is a name used to identify some object. The S evaluator will search for an object associated with that name. If it finds one, the name evaluates to (a copy of) that object. The evaluator also looks for a function object named `"mean"`, and in the expression

```
population - mean(population)
```

the evaluator needs a function object of name `"-"` and proceeds to look for one.

In all these examples, the *syntactic* use of the character strings identified them as names for objects. In particular, the S parser recognizes a name as a string formed from letters, numbers and `"."`, and not starting with a number. Other strings are syntactically recognized as infix or unary operators, and therefore also as names. The syntactic sense is not how we want to talk about names here; rather, we will discuss the database mechanism in S by which essentially *any* character string can be associated with an object as its name.[1]

The specific topic of this report is the association of objects with names, individually and within a collection of objects that will be called a *frame* or *database* depending on the context. S allows users to access and modify such collections in the process of computation. The facilities for doing so are automatic and casual users of the language rarely if ever need to think about what happens. However, programming applications can sometimes benefit from a more explicit knowledge of evaluation frames or databases.

[1] In [1], this generality is not true of permanent names, as we will discuss below.

3 Evaluation and Frames

During the evaluation of an expression, the S evaluator creates, uses, and releases *frames* in which names and objects are associated temporarily while evaluating function calls and other expressions. A frame is an internal construct of the evaluator, which can be be modeled as an S object, specifically a list. The named components of the list correspond to objects of the same name in the frame.

The evaluator initially creates a top-level frame corresponding to the user expression. When it proceeds to evaluate a function call, it begins by finding the definition of the function; that is, the function object corresponding to the name in the call. For example, in the expression

```
mean(population)
```

the function object `mean` might have the definition

```
function(x, trim)
{
    if(missing(trim))
        sum(as.double(x))/length(x)
    else .S(mean(x, trim), "mean")
}
```

To evaluate the function call, the evaluator will create a new frame, initially containing the (unevaluated) argument expressions, in this case `x` and `trim`. Ordinary assignments in the body of the function take place in this frame, and, during evaluation of the body, the search for a named object begins in this frame. More details are given in [1], page 343.

All the frames in existence at any point during evaluation together form another internal structure, the list of frames. The top-level frame is the first element of that list. New frames created by the evaluator are added at the end of the list. Frames can be referred to by number, interpreted as their position as elements in the frames list. In particular, this allows expressions to step outside the ordinary evaluation mechanism—-for example, by getting or assigning data in a particular frame via `get()` or `assign()`, and by evaluating an expression there via `eval()`.

The frame in which a function call occurs is the *parent* of the frame in which that function call is evaluated. Every frame thus has a unique parent (including the top-level frame which for completeness is considered its own parent). In particular, knowing the parent of a frame is essential to the S philosophy of *lazy evaluation* of arguments. Arguments to a function call are not evaluated until the actual object is needed (typically when some low-level computations have to be done). At that point the expression for the actual argument, if there was one, is evaluated in the *parent* frame of the current frame, and the resulting value is associated with the name of the argument in the current frame. The lazy evaluation mechanism is the reason that the argument `trim` could be omitted in our example, even though no default value was provided. An evaluation frame remains in existence until the corresponding function call (if any) is complete, until an explicit `return()` expression is evaluated, or until the parent of the current frame ceases to exist. It is then released by the evaluator.

The frames created during evaluation are accessible to user expressions through a variety of functions.[2] The most relevant for the present discussion are:

```
sys.frame(n)    # the n-th frame, as a list object
sys.parent(i)   # the parent frame i generations back
sys.frames()    # a list containing all the frames
sys.parents()   # all the parent frames.
```

Parent frames are identified by numeric position in the frames list. Notice that `sys.frames()` and `sys.parents()` are both needed to give the complete information about the frames list.

The function `eval()` operates like the S evaluator itself, but under explicit control of the user. Its first argument is an S object that is an expression and its second argument, if included, defines the frame in which the evaluation of the expression should take place. For example,

```
eval(expression(mean(population)), state.data)
```

says to take the object `state.data` to define a new frame in which the expression `mean(population)` is to be evaluated. For this to make sense, `state.data` should be a list-like object whose named components will define objects in the new frame. The second argument to `eval()` can also be a number, in which case it means to use the corresponding element in the frames list as the frame for evaluation. In particular, a frame can be created explicitly (section 4) for this purpose.

Evaluation frames can have attributes. If the function object being called has attributes, the S evaluator sets the attribute list of the frame to contain

[2] A few new functions have been added since [1], and some calling sequences extended. See the on-line documentation for more details.

those attributes. Similarly, attributes of an object used to create a frame, via `eval()` or the function `new.frame()` in section 4 are transferred to the evaluation frame created. Attributes of evaluation frames can be accessed or set by the function `frame.attr()`:

```
frame.attr("tol") # get the attribute "tol" of the current frame
frame.attr("tol") <- .001 # set it
```

The function `frame.attributes()` returns the list of all the attributes of the current frame.

4 Explicitly Creating Frames

Frames are created, used, and released as a result of function calls; these steps require no special user action. However, some applications want more control over the process[3]. In typical examples, various expressions are to be evaluated in terms of a current frame that is explicitly constructed. This can be done by repeated use of `eval()` with the frame (as a list) given as the second argument, as illustrated on page 5. However, with multiple expressions this mechanism is inefficient and clumsy. Instead, we can create a new frame explicitly from an object. This frame then stays around and can be supplied as a numeric argument to `eval()`. The entire frame can be passed back to a calling function by number, providing a very efficient mechanism for communicating intermediate results.

The two functions that accomplish this are:

```
new.frame(object)
move.frame(frame, to)
```

The first of these takes any (recursive) object and creates a new frame in the frames list. The objects in the frame will be the components of `object`. The value of `new.frame()` is the numeric index of the new frame in the frames list. The new frame will be a child of the frame from which `new.frame()` was called. Under normal circumstances, it will disappear when its parent frame does. To transmit the contents of the frame back to a previous function we could use the `sys.frame()` function, but this would involve explicitly copying the frame. If the intention is to use the frame for further computations, a simpler and more efficient way is to *transfer parentage*. The call

[3]All this material is new.

to `move.frame()` causes the frame in the first argument, identified by number, to become a child of the frame specified, also by number, in the second argument. By default, `to` is taken to be the parent of the frame calling `move.frame()`. Notice that frames disappear automatically only when they explicitly return at the end of a function computation, or when their parent frame disappears. Thus, transferring ownership to a parent guarantees that `frame` will survive, even when the current function evaluation is complete. This is precisely the purpose of the transfer.

Transmitting large amounts of data by re-assigning parentage of frames is much more efficient than returning the data from a function, since no actual transfer of data occurs, only some changes in the internal bookkeeping of the evaluator. For this reason, it is recommended in applications where repeated evaluations take place under the control of some function. The mechanism is not intended to replace the usual way in which S functions return their values; that is the natural and clean way most computations should be handled, and the inefficiencies are irrelevant in simple applications. Use of explicit frames inevitably violates the "functional language" aspects of S and will tend to obscure what is going on. Also, of course, something eventually has to be physically returned, since all the frames will disappear at the completion of the top level expression.

As a simple use of the new mechanism, here is a function, `xframe()`, which is given some names, and which constructs a frame containing all the objects corresponding to those names. It makes the caller of `xframe()` the parent of this new frame and returns the frame number as its value.

```
xframe <- function(names) {
    assignment <- substitute(x <- x)
    n <- new.frame()
    for(var in names) {
        assignment[[1]] <- assignment[[2]] <- as.name(var)
        eval(assignment, n)
    }
    move.frame(n)
    n
}
```

So, for example,

```
n <- xframe(c("Fuel", "Weight", "Displacement"))
```

would create a frame containing the three objects named, and would set `n` to the numerical index of this frame. Then the function calling `xframe()` could evaluate expressions in frame `n`. The first line of the `for` loop in `xframe()` constructs an S assignment expression that copies each variable into the local frame; the first time through the loop the evaluated expression would be

```
Fuel <- Fuel
```

and so on.

5 Databases; the Search List

In the model of evaluation for S, a database is a collection of named objects, internal to the evaluator. More precisely, for each currently active database and any character string, five primitive operations are defined, corresponding to the generic functions:

```
exists(name, ...)
get(name, ...)
assign(name, value, ...)
remove(names, ...)
objects(...)
```

These functions can refer to either frames or databases. In the latter case, their semantics are:

`exists()`: returns `TRUE` or `FALSE` according to whether an object of this name exists in the database;

`get()`: returns (a copy of) the object, if it exists;

`assign()`: associates (a copy of) a specified object in the database with the given name;

`remove()`: deletes the objects, if any, of the given names in the database;

`objects()`: returns the character vector of the names of all the objects in the database (this is not strictly required—see section 10—but will be provided for all standard types of databases).

The general scope of databases is the topic of the rest of this report; essentially, anything can be a database that associates some character strings with S objects in such a way as to define the primitives.

This scope is in fact *very* general, and constitutes what we will call the database mechanism in S[4]. Specifically, a database element in the search list may be:

- the name of a file-system directory;
- a list-like S object;
- one or more compiled C-language images of S databases;
- an object inheriting from some class of user-defined databases.

Beginning in section 7 we will discuss each of these kinds of database.

Databases are attached by a call of the form

```
attach(object, pos)
```

The optional argument `pos` says where in the search list the new database should appear, with the default being position 2. A third argument, `name`, can be used to specify the name corresponding to the new database, in the character vector returned by the function `search()` (see the example on page 21). The `object` argument to `attach()` is used to determine the kind of database by applying, in order, the following rules:

1. If the object inherits from a class for which there is a method for the function `dbobjects()`, then this inheritance determines the type of database. One special case should be noted: there is a method for class `"compiled"` for this function, whose existence just serves as a way to force interpretation of an object as a compiled database. In this case the object is interpreted as a character vector naming one or more collections of objects, compiled from C code as described in section 9. In all other cases, the object will be treated as a user-defined database class.

2. If the object is recursive (e.g., a list) it is interpreted as an object-type database, and its named components are copied to form the dictionary. All the components of the object must be named.

3. If the value of the object is a character string corresponding to a file-system directory, this directory is attached.

[4]In [1], only file system directories could be used as databases, and that only in a more restricted way than described in section 7 below. The search list is much more general, and the explicit `.Search.list` character vector no longer exists.

4. If the object is a character vector whose elements specify one or more compiled databases, these are concatenated and attached.

An error results if none of these interpretations succeeds.

All the currently attached databases make up the *search list*. References to databases by number correspond to positions in the search list, in the same way that references to evaluation frames by number correspond to positions in the frames list. Thus,

```
get("x", where=2)
```

will look for an object named `"x"` in the second database on the current search list. The search list is part of the S evaluator, and not an ordinary object itself. Databases are removed from the search list by the function `detach()`. The functions `attach()` and `detach()` together define the search list, and should be thought of a primitives interacting with the S evaluator.

Based on the way S has been installed for a particular machine, the search list at the beginning of an S session will contain various databases for S functions and datasets, probably including some libraries. The first database on the search list is always the *working* database, in which ordinary assignments in the top-level expression take place. Initially, this will usually be a subdirectory named `".Data"` of the current directory or of the user's login directory. Subsequently, use of `attach()` or `detach()` may modify the search list so that any database becomes the working database (but a non-writable database can't function as the working data).

The initial contents of the search list can be controlled by defining one or both of the shell variables `S_WORK` or `S_PATH`. Both of these are interpreted as path variables in the usual sense of the shell; that is, their value as a character string is interpreted as names, separated by the character `":"`. If `S_WORK` is defined, the names will be tried in order as candidates for the working data. If `S_PATH` is defined, the names it contains will be attached (if they exist) as databases following the working data in the search list. The flexibility supplied by this mechanism can be useful in setting up nonstandard use of S, but the mechanism is somewhat limited since databases are identified by character strings, rather than general S expressions. As experience with the database mechanism grows, the approach to initializing the search list may also evolve. On the whole, if applications can work with the default initial search list, it's a good idea to do so.

In addition to the databases in the search list, there is one more database, called the *session* database[5]. This database is initially empty; objects can be assigned to it by the `assign()` function and in all respects it can be used like any other database, except that it has no external counterpart and disappears at the conclusion of the S session. The session database is searched before any of the external databases and for this reason it is convenient sometimes to think of it as "database 0". This is how it appears as an argument to functions like `assign()` or `get()`:

```
assign("Check",TRUE, where=0)
```

creates an object named `Check` in the session database. The session database has the database semantics (section 6), even though it does not start off from attaching an external object.

In the process of evaluating users' expressions, the S evaluator will attempt to match names with corresponding objects. We describe now the rules by which this matching takes place. At any time in the evaluation of an S expression, there is a current frame, typically associated with the function call currently being evaluated. On encountering a name, the evaluator tries in turn to match the name first in the current frame, in the top-level frame (frame 1), in the session database, and from the search list.

At this point, it is useful to represent the search process diagrammatically[6]. The figure suggests the search procedure involved in evaluating the expression `sqrt(lottery.payoff)`, typed by the user. The user's expression is evaluated initially in Frame 1. As soon as a function call, to `sqrt()` in this case, is to be evaluated, the name `"sqrt"` is searched for in Frame 1 and the various databases. Once found, a new frame (Frame 2) is opened, the arguments to `sqrt()` are matched against the definition and the body of the function is evaluated in Frame 2. That body itself contains a call to another function, the operator `"^"`. Another search finds this object, and a new frame is created (Frame 3).

Since this new function contains an interface to the internal C code for operators, we are now at the point where the arguments to `"^"`, and thus to `sqrt()`, have to be evaluated. Eventually this requires finding the object

[5] In [1], this role was partly played by *frame* 0. The more consistent and much more general approach to databases now allows a cleaner handling of the frame 0 idea. For compatibility, you can refer to the session database as either frame 0 or database 0.

[6] Compare this with the picture in [1], page 118. The search semantics are essentially identical; what can go into the boxes and the treatment of databases are new.

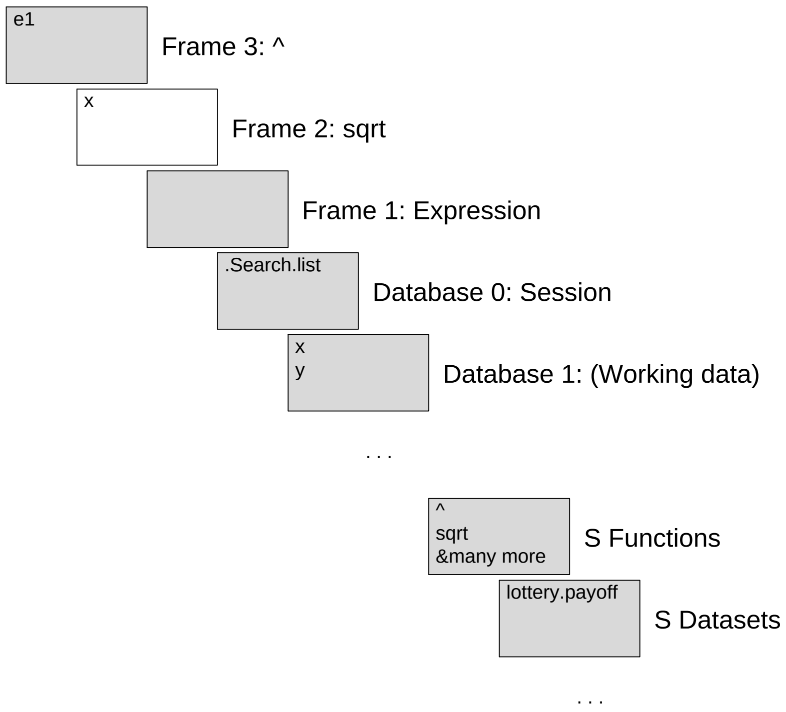

Figure 1: *The search for a named object proceeds through the shaded boxes top to bottom: the local frame, frame 1, the session database and then each database on the search list. The expression* `sqrt(lottery.payoff)` *is being evaluated.*

`lottery.payoff`, in this case from the S datasets, assuming no object of this name appeared in earlier databases.

At this stage, a few general comments are needed:

- Each of the databases has an associated in-memory dictionary; searching in a database means a hash-table lookup in the dictionary. The time taken to look up a name remains small over a wide range of the number of databases in the search list or the size of the individual dictionaries.
- The database in the search list is part of the internals of S. The database and its dictionary are kept consistent with changes initiated by the S process. Changes made by *other* processes to the external database

that was originally attached may not be reflected. See the discussion on page 14 below.

- Depending on the kind of database, the dictionary may contain the names of the objects, the entire objects, or a mixture of names and objects.

- When only a name is found, the database mechanism proceeds to "read" the object. It may be directed in this case, by the S option `"keep"` to retain the object in the dictionary either for the rest of the session or just for the current top-level expression (see the detailed documentation for `options()`).

6 Database Semantics

This section describes how the database mechanism organizes and controls the access and assignment of data[7]. The information in this section gives details that are useful in understanding the relative efficiency of alternative organizations of data and also defines more precisely the behavior of the five database primitives. Understanding this material is important for the design of new database classes (section 10).

Of the five primitives for database management on page 8, `assign()` and `remove()` alter the contents of the database. This includes altering an object by a replacement (e.g., of a subset or attribute). Any such changes in any of the databases (including the session database) are kept pending until successful completion of the top-level expression containing them. This means that, while the changes take place immediately as viewed by the computations in the expression, the changes are not permanently *committed* until successful completion of the expression. Any error or interrupt causes the databases to be restored to their state at the beginning of the expression.

The dictionaries are tables kept internally. Conceptually, when an object, say `database`, is first attached to the search list, the dictionary is initialized to the value of the function

```
dbobjects(database)
```

[7]Explicit database semantics, based on the five primitives, is new. So is the use of dictionaries for all databases, which substantially improves the efficiency of attaching large databases. The commitment semantics have been extended to all databases and to include removal as well as assignment.

In practice, the initialization of the table is built-in for the standard kinds of databases (directories, objects, and compiled), but the definition of the `dbobjects()` method is important for the user-defined database classes of section 10. The value of this function can be a character vector, a list, or `NULL`. In the first case, only the names of the objects are stored in the table. When an object from the database is to be accessed, it will still need to be read. If the value of `dbobjects()` is a list, both the names and the objects will be stored in the table and no further reading will be needed. If the value is `NULL`, the database is to be used without a dictionary. This means that each query must be answered by using the `exists()` primitive, with obviously a serious performance penalty. Databases without dictionaries are provided for possible use with user-defined database classes, but all of the built-in forms of S database provide dictionaries.

The dictionary for a database and any objects stored there persist until that particular database is detached, or until the end of the session with S. This is important for two reasons. First, since all databases will normally have dictionaries, the computations required to find a particular object do not appreciably slow down when a number of possibly large databases are on the search list. Second, attaching and detaching databases only requires substantial computations related to the newly attached database, and not recomputation of tables for the other databases in the search list.

There are a few exceptions to the general pattern of commitment as described so far. Two special situations need to be considered:

- when the function `synchronize()` is called;
- when the database is detached from the search list.

The function `synchronize()` is an explicit request by the user to make the database and its dictionary consistent. If `synchronize()` is called without arguments, it will commit all changes to all databases. If `synchronize()` is called with a vector of one or more numbers as an argument,

```
synchronize(i)
```

it will also re-attach the databases in position(s) `i` of the search list. This will update the corresponding dictionaries with respect to changes made by other processes, or changes to the underlying object or other external entity. Clearly, reattaching involves considerably more computation than just committing. It should (and must) be used when the contents of the dictionary may be out of date and the S process needs the current version of the external database.

Detaching a database may require action when changes are held only in the internal dictionary. When the database is detached, the internal information disappears, and in certain cases S will take some action to save that information. What happens depends on the kind of database. For directory databases, no action is taken. Compiled databases are regarded as read-only, so again no action is taken. For object databases, a new object may be assigned in the working data, depending on the argument `save` to the function `detach()` (see the online documentation). For user-defined database classes, a method can be defined for the function `dbdetach()` to carry out any desired action.

One detail should be noted. In the current implementation, the detach action for object databases is a standard assignment to the working data, which will itself not be committed until the expression completes. Data can be lost if the expression dies after detaching a modified object database. To avoid this, use `synchronize()`:

```
save.detach <- function(n) {
    detach(n)
    synchronize()
}
```

The function `save.detach()` guarantees to commit the saved version of some detached object.

When an error or interrupt occurs in an S expression, all the changes to databases initiated in that expression are backed out. Commitment and back-out are managed by keeping an internal back-out structure for each database that, essentially, records the pending actions for that database and the previous contents of the dictionary for the same objects. On error, the dictionary is restored to its contents before the expression began. The same mechanism is used to restore dictionary contents for objects that are read during an expression but are not to be kept permanently hashed in the dictionary.

As mentioned, the search list and the databases in it are part of the internal S evaluator. As with evaluation frames, some S functions are provided to inspect the databases:

```
database.object(n)
database.type(n)
database.status(n)
```

Each of these takes as argument an index into the current search list. The first returns the current object corresponding to the given database. The second returns the type (`"directory"`, `"object"`, `"compiled"` or `"user"`). The third returns the status, indicating whether the database has been modified since being attached. See the online documentation for more details on any of these functions.

Database objects can have attributes, copied in from the attached objects. To access an attribute named `"abc"` of the database in position `n`

```
database.attr("abc",n)
```

If the position is not specified, the first database in the list having the attribute will be used. Database attributes can also be set by using the same function on the left of an assignment.

In any of the functions on page 8, the database may be specified numerically, as a position in the search list. It may also be given explicitly–for example, as a character vector. The interpretation in this case is that the corresponding database will be attached temporarily, if it is not already on the search list. In the current implementation, a limited (but fairly large) number of such databases will be attached and kept around until the end of the current top-level expression. The intent is to make repeated reference to an explicitly named database reasonably efficient. There are some limitations, however, and the mechanism is ambiguous in the case of objects (what if the object changes between references to it?). A better strategy for repeated access is to attach the object and then arrange to detach it on exit from the appropriate function.

7 Directory Databases

If the rules on page 9 identify the database as a directory in the file system, the files in that directory are interpreted as S objects. Internal code in the S executive reads and writes S objects as files. This is as described in [1], section 5.4, except for the mechanism needed to overcome limitations in file-system naming abilities. As noted, it is a semantic requirement that *any* character string can be unambiguously associated with an S object in a frame or database. File systems prohibit the use of some character strings as file names and may in addition set an upper limit on the length of file names. To overcome these limitations, the S executive maintains an aliasing mechanism to rename objects if necessary and to keep a table of the names[8].

From the user's viewpoint, this detail is only relevant in that one should *not* manipulate the files in the directory casually outside of S unless absolutely conversant with the way the database mechanism works. S databases should be examined, dumped, or restored using the utilities in S:

```
names <- objects(where)  # the names of all the objects
data.dump(names, file)   # dump specified objects
data.restore(file)       # restore objects from a dump
```

In particular, using the `ls` command in the operating system will not give a valid list of the objects in the database, and removing or moving files at the operating system level is dangerous.

The aliasing mechanism is important, however, in that it may be needed to implement non-standard database classes that use the file system (section 10 shows an example) and to port S to other operating systems or to non-standard file systems. For those interested, here are some pointers to how the mechanism works. For a given object name and a given directory, S constructs internally a "true file name". A table of the current mapping from object name to true file name (for all names not mapped into themselves) is kept in the directory, and updated automatically.

The mapping can be accessed from S via a routine callable from `.C()`:

```
.C("True_file_name",name, n, TRUE)
```

with `name` the object name, `n` the position of the database, and third argument a flag that says whether the true file name should be constructed and added to the table if it is not there already. If the attached name of the database is known, rather than the position in the search list, `n` can be given as `0`, with `name[2]` set to the name of the database. See the example on page 24.

Internally, two other C subroutines control the definition of file names:

```
char *
true_file_name(name, database, db_name, action)
char *name; long database; char *db_name; int action;
{
    ...
}
```

[8] In [1], only objects with legal file names could be assigned. Names that were too long were truncated, with possible ambiguities introduced.

```
int
bad_name(name)
char *name;
{
    ...
}
```

The second of these returns 0 or 1 according to whether the name is considered illegal as a file name for an S object. The first (which calls the second) returns either its `name` argument or the name of a file to be used instead if `name` is not legal. It may be used for a non-attached directory by supplying the second argument as `-1` and giving the directory name as the third argument. The action argument is an extension of the flag in the third argument of the `.C()`-callable subroutine. As a side effect, calling this routine may update and write out the table of transliterated names for this database. The versions of `true_file_name` and `bad_name` in the standard S source code are designed for Unix file systems, where file names may not contain `"/"`, where there are a few additional proscribed names, and where there may be a system-dependent limit on file name lengths. In addition, it will always be necessary for `bad_name` to prohibit some other names, in order to protect the transliteration mechanism used by `true_file_name`.

This mechanism should allow any character string to be mapped to a file name, so long as the mechanism doesn't run out of transliterated file names or the file system impose some limit on the size of directories. (If the file system allows file names of length M, there are about 10^{M-2} possible transliterations, so the first problem should be academic.)

8 Objects as Databases

Any recursive object in S can be used to define a frame, as we noted in section 4. Equally, any such object can be attached as a database to be persistently available[9]. The semantics of attaching an object are that the new database is initialized with a dictionary containing all the named components of the object. To be precise, the dictionary contains *copies* of the components. In line with S's general dislike of indirect references, changes in the original object and in the objects of the database are independent.

Notice that the dictionary formed from the object contains all the data, not just the names. By comparison to a directory, subsequent access to the

[9] All the facilities in the next three sections are new.

objects will be faster. On the other hand, attaching truly huge objects this way will be fairly expensive in both memory and computing time. If only a small fraction of the data is needed, a directory is likely to be more efficient. If the data only changes rarely, on the other hand, a compiled database as described in the next section will be an attractive possibility.

Another relevant comparison is between an object attached as a database and one attached as a frame, as in section 4. There are again some efficiency issues — the database is only attached once, so access in several expressions should have somewhat lower overhead. More critical is the different position in the search list. Using `new.frame()` produces a frame that will typically be the current frame for calls to `eval()`. Names will be matched to the new frame first. Attaching an object as a database means that the local and top-level frame and the session database, plus any databases earlier in the search list, will override names in the attached object. Which semantics is preferred depends on the application. For the sort of computation described in [3], for example, the usual assumption is that the names in the created frame should take precedence.

9 Compiled Databases

For collections of S objects that are used frequently but that change seldom, and particularly for collections with many objects, there is an advantage in reducing the time and memory required to attach the objects. In particular, it may be desirable to build the objects into a compiled version of S. A mechanism has been designed to accomplish this. From a list of S objects, the function `cdump()` will produce a file of C language source code. This source code (highly unreadable by humans) contains data initializations that construct internal copies of the S objects, plus an entry point that when invoked will enter those definitions into one of the internal dictionaries in the database mechanism.

Let's look at a tiny example to illustrate, although the appeal of compiled dictionaries is strongest for big examples. Nevertheless, suppose we decide to make a compiled dictionary for the arithmetic operators. The first step is to create a C program that contains the S objects. The function `cdump()` does this. It takes two main arguments, a vector of the names of the objects to be dumped and a name to identify the dump, both analogous to the arguments to `dump()` or `data.dump()`. What it does is different. The side-effect of `cdump()` is to write a file of C source code, containing a (large) number of data declarations and a single executable routine. The second argument is required and is the name of the C language entry point that will be used to attach the database. The value of the call to `cdump()` is the name of the file on which the C code has been written. By default, this is the the entry point name followed by `".c"`. In our example, `"arithops"` contains the names of 7 S functions for arithmetic operations. We will create a C dump with the name `"arith_ops"`:

```
> arithops
[1] "%%"  "*"   "+"   "-"   "^"   "/"   "%/%"
> cdump(arithops,"arith_ops")
[1] "arith_ops.c"
```

The C source in the file `arith_ops.c` contains data statements to define objects with the same contents as the 7 functions, a table designed to help hash all these definitions in a dictionary, and a single executable routine named `arith_ops`, which when called will give the information needed by the database mechanism to attach the table.

This C program needs to be compiled and loaded with S. It can be loaded either dynamically or statically; e.g., for static loading:

```
S LOAD arith_ops.c
```

To compile in preparation for dynamic loading, the local directory needs to be made into an S chapter. The shell command

```
S CHAPTER
```

should be run once in the directory. Once the chapter has been set up,

```
make arith_ops.o
```

will create the appropriate object file. Attaching an object that identifies the executable routine then causes the database mechanism to initialize a dictionary to the contents of the compiled database:

```
> dyn.load("arith_ops.o")
> attach("arith_ops")
> objects(2)
[1] "%%"  "%/%" "*"   "+"   "-"   "/"  "^"
> get("*",where=2)
function(e1, e2)
.Internal(e1 * e2, "do_op", T, 4)
```

The attached database can be used like any other, except that assigning and removing in it will have no permanent consequences. Changes will disappear at the end of the session. To revise a compiled database, repeat the `cdump()` computations. (Tinkering with the C code is rarely a good idea. C is an assembly language for this purpose, as any look at the code will quickly establish.)

Compiled databases reduce the overhead of attaching to an essential minimum, just the few instructions needed per name to insert the entry into the internal table. In addition, `cdump()` has an optional argument `optimize`, `TRUE` by default, which triggers some extra work to find common subexpressions in the S objects. Even without optimizing, atomic objects of length one (e.g., character strings or numeric values in expressions) will be detected as common subexpressions. Optimization adds a little to this, usually worthwhile given that the time spent in `cdump()` is unlikely to matter much. Optimization seems to reduce the size of the C code by about 10% in some moderately large examples.

Compiled databases are still somewhat experimental, and may well end up being a standard way to include many of the standard S functions. One limitation and portability issue is that the generated C code for a sizable number of objects can overflow various internal tables in the local C compiler. Partly for this reason there is another mechanism that allows multiple compiled routines (each from a separate `cdump()` call) to be linked together as a single database. A character vector naming all the routines, with class `"compiled"`, will cause all the data to be put in one compiled database when attached. Suppose, for example, `"db_stuff"` was another set of compiled data, compiled on file `db_stuff.o`. The following creates a compiled database object that links the two sets of objects together when attached.

```
> mydb <- c("arith_ops", "dbstuff")
> class(mydb) <- "compiled"
> dyn.load("arith_ops.o")
> dyn.load("dbstuff.o")
> attach(mydb,name="Compiled")
```

The `name=` argument to `attach()` controls the character string associated with an attached database in the character vector returned by the `search()` function. For compiled databases, the default name will be the first element of `mydb`, which we override in the call to `attach()`.

Compiled databases, compared to directories, provide data compression and faster initial access. On the negative side, compiled databases really are built in; the S process pays the full cost of additional memory whether or not the database is attached. On balance, they are very attractive for heavily used collections of objects, particularly S functions. As greater experience is gained, it seems likely that they will be the medium of choice for at least the most frequently used of the S functions. The precise tradeoff remains to be determined and in any case is likely to depend on the particular computer system, since it is in part trading additional memory for less file-system activity.

10 Database Classes

In addition to the built-in forms of database described so far, the database mechanism explicitly anticipates extension to include other kinds of database. One interesting application will be to link S with various database management systems (dbms). In this scenario, objects are accessed by queries into the dbms which return S objects, and S expressions to assign or remove objects are interpreted as requests to update the contents of the dbms.

Such interfaces to dbms will likely be important for a number of applications. First, they allow the data analysis to approach closer to application projects by providing a direct interface to the databases owned by those projects. Second, they provide an open-ended interface to very large databases (e.g., with thousands of objects) without having to translate large or arbitrarily chosen subsets of the database.

To make the interface clean and relatively easy to implement, we use the class/method mechanism [3]. Specifically, a set of generic functions exists corresponding to the database primitives needed to define a new class of database:

```
dbread(database, name)          # read ‘name’ from the database
dbwrite(database, name, object) # write ‘object’ as ‘name’
dbremove(database, names)       # remove the named objects
dbobjects(database)             # the names of all the objects
```

Defining a new class of databases amounts to implementing methods for these four generic functions. The methods must perform the following functions:

`dbread()`: return the S object on `database` corresponding to the character string in `name`;

`dbwrite()`: store a permanent version of `object` in `database` so it can be later retrieved as `name`;

`dbremove()`: remove the object stored as `name` in `database` or warn if the object is not there;

`dbobjects()`: return a character vector giving the names of all the objects currently in `database`, or a list giving both names and current contents, or `NULL`. In either of the first two cases a dictionary will be formed internally by the S evaluator, containing the names and optionally the objects when the database is attached.

For some applications, it is undesirable or even impossible to return a vector of the database objects or their names from `dbobjects()`. For example, the actual database may contain so many objects that it would create an excessively large in-memory dictionary. In this case, `dbobjects()` will return `NULL`. No dictionary will be formed, and every attempt to access an object from the database will be preceded by a call to a fifth generic function, `dbexists()`.

```
dbexists(database, name, mode) # does 'name' exist on the database?
```

A method for this function must exist and must return a logical saying whether the desired object exists. The definition should have the default value `"any"` for `mode`. Since using `dbexists()` significantly complicates and slows down each reference to data from the database, this approach should only be used when keeping a dictionary for the database is clearly a bad idea. Also, such databases should if possible be attached *after* the standard S databases; otherwise, every function reference results in a call to the `dbexists()` method, with a very large overhead.

The object returned by `dbobjects()` can also control the system's strategy in hashing any objects assigned to this database while it is attached. If the object has an attribute `"hash"`, this is treated analogously to the the `"keep"` option, but solely for this database. If the attribute has value `"any"` then all objects are hashed, so that `dbread()` will not be called. If it has value `"function"`, functions will be hashed, other objects will not. By default, no objects are permanently hashed for user-defined database classes.

The details of designing serious database classes are beyond our scope here, but we can sketch a simple example that will cover the essentials. For data arising from dynamic display of networks ([2]), applying file compression techniques to the binary files containing S objects in directory databases was observed to save considerable file space. This suggests keeping the S binary files in compressed form and only expanding them when reading them into S. We can implement databases of such files transparently to the S user by defining a class of objects, of class `"compress"`, say. Methods for objects of this class will read and write compressed files.

First, we need a class of database objects. As usual, a function of the same name as the class will return a suitable object from the class. In this case, the object just needs to contain the character string specifying the directory in which we will keep the objects. It will also have the corresponding class attribute, to drive the choice of methods.

```
compress <- function(directory) {
    unix(paste("if ! test -d", directory,"; mkdir", directory,
       " ",directory,"/__aux ; fi",out = F)
    class(directory) <- "compress"
    directory
}
```

The `unix()` command will create the directory, if it does not already exist, and also a subdirectory, whose purpose will become clear shortly. The object returned by `compress()` is the database object. It will be inserted into the search list; the class attribute will then guide the database mechanism to call the corresponding methods `dbread.compress()`, etc.

Now we can implement the methods. The first point to note is that these are called when the database mechanism is ready to physically read, write, or remove the relevant objects. The methods need not worry about the semantics of commitment discussed in section 6. The methods do, however, have to cope with the semantic requirement that any character string can name an object. Since we're dealing with files here, we can piggyback on the existing mechanism for directory databases, as described in section 7. Let's begin with `dbwrite.compress()`.

```
"dbwrite.compress"<-
function(database, name, object)
{
    temp <- unix("echo x$$")
    dir <- as.character(database)
    tname <- (.C("True_file_name",
            n = as.character(c(name, dir)),
            as.integer(0),
            as.integer(1))$n)[1]
    aux <- paste(dir, "/__aux", sep = "")
```

```
        assign(temp, object, w = aux)
        synchronize()
        unix(paste("cd ", aux, "; compress -f ", temp, "; mv ",
             temp, ".Z ../", tname, sep = ""), output = F)
}
```

This method first assigns the object in the ordinary way, into that subdirectory we created. Calling `synchronize()` forces the physical output of the object. Then the `compress` command in the operating system is used to compress the new file, and the compressed file is moved back to the database directory. To avoid any constraints on file names, the method uses the C language entry point, `True_file_name`, to get back a suitable actual file name, as shown on page 17.

The corresponding method for `dbread()` is analogous:

```
"dbread.compress"<-
function(database, name)
{
        temp <- unix("echo x$$")
        temp.Z <- paste(temp, ".Z", sep = "")
        on.exit(unix(paste("cd", aux, "; rm", temp, temp.Z)))
        dir <- as.character(database)
        tname <- (.C("True_file_name",
                n = as.character(c(name, dir)),
                as.integer(0),
                as.integer(0))$n)[1]
        aux <- paste(dir, "/__aux", sep = "")
        cmd <- paste("cd ", aux, ";ln ../", tname, " ", temp.Z, "; zcat ",
                temp.Z, " >", temp, sep = "")
        unix(cmd, output = F)
        get(temp, w = aux)
}
```

This links the compressed file (so as not to have to copy it), uncompresses it and then gets the uncompressed file (from an ordinary S directory database).

The `dbremove()` and `dbobjects()` methods are even simpler, since for both these purposes the database can just be treated as an ordinary directory database, attached without its class attribute.

```
"dbremove.compress"<-
function(database, name)
      unix(paste("rm ", as.character(database), "/",
          name, sep = ""), output = F)
```

```
"dbobjects.compress"<-
function(database)
    NextMethod("dbobjects")
```

References

[1] R. A. Becker, J. M. Chambers, and A. R. Wilks (1988), *The New S Language*, Wadsworth, Pacific Grove, California.

[2] R. A. Becker, S. G. Eick, and A. R. Wilks (1991), Basics of Network Visualization, *IEEE Computer Graphics and Applications*, **11**, pages 12-14.

[3] J. M. Chambers and T. J. Hastie (eds) (1991), *Statistical Models in S*, Wadsworth, Pacific Grove, California.