Interacting with Data using the **filehash** Package for R

Roger D. Peng <rpeng@jhsph.edu>
Department of Biostatistics
Johns Hopkins Bloomberg School of Public Health

Abstract

The **filehash** package for R implements a simple key-value style database where character string keys are associated with data values that are stored on the disk. A simple interface is provided for inserting, retrieving, and deleting data from the database. Utilities are provided that allow **filehash** databases to be treated much like environments and lists are already used in R. These utilities are provided to encourage interactive and exploratory analysis on large datasets. Three different file formats for representing the database are currently available and new formats can easily be incorporated by third parties for use in the **filehash** framework.

1 Overview and Motivation

Working with large datasets in R can be cumbersome because of the need to keep objects in physical memory. While many might generally see that as a feature of the system, the need to keep whole objects in memory creates challenges to those who might want to work interactively with large datasets. Here we take a simple definition of "large dataset" to be any dataset that cannot be loaded into R as a single R object because of memory limitations. For example, a very large data frame might be too large for all of the columns and rows to be loaded at once. In such a situation, one might load only a subset of the rows or columns, if that is possible.

In a key-value database, an arbitrary data object (a "value") has a "key" associated with it, usually a character string. When one requests the value associated with a particular key, it is the database's job to match up the key with the correct value and return the value to the requester.

The most straightforward example of a key-value database in R is the global environment. Every object in R has a name and a value associated with it. When you execute at the R prompt

```
> x <- 1
> print(x)
```

the first line assigns the value 1 to the name/key "x". The second line requests the value of "x" and prints out 1 to the console. R handles the task of finding the appropriate value for "x" by searching through a series of environments, including the namespaces of the packages on the search list.

In most cases, R stores the values associated with keys in memory, so that the value of `x` in the example above was stored in and retrieved from physical memory. However, the idea of a key-value database can be generalized beyond this particular configuration. For example, as of R 2.0.0, much of the R code for R packages is stored in a lazy-loaded database, where the values are initially stored on disk and loaded into memory on first access (Ripley, 2004). Hence, when R starts up, it uses relatively little memory, while the memory usage increases as more objects are requested. Data could also be stored on other computers (e.g. websites) and retrieved over the network.

The general S language concept of a database is described in Chapter 5 of the Green Book (Chambers, 1998) and earlier in Chambers (1991). Although the S and R languages have different semantics with respect to how variable names are looked up and bound to values, the general concept of using a key-value database applies to both languages. Duncan Temple Lang has implemented this general database framework for R in the **RObjectTables** package of Omegahat (Temple Lang, 2002). The **RObjectTables** package provides an interface for connecting R with arbitrary backend systems, allowing data values to be stored in potentially

any format or location. While the package itself does not include a specific implementation, some examples are provided on the package's website.

The **filehash** package provides a full read-write implementation of a key-value database for R. The package does not depend on any external packages (beyond those provided in a standard R installation) or software systems and is written entirely in R, making it readily usable on most platforms. The **filehash** package can be thought of as a specific implementation of the database concept described in Chambers (1991), taking a slightly different approach to the problem. Both Temple Lang (2002) and Chambers (1991) focus on generalizing the notion of "attach()-ing" a database in an R/S session so that variable names can be looked up automatically via the search list. The **filehash** package represents a database as an instance of an S4 class and operates directly on the S4 object via various methods.

Key-value databases are sometimes called hash tables and indeed, the name of the package comes from the idea of having a "file-based hash table". With **filehash** the values are stored in a file on the disk rather than in memory. When a user requests the values associated with a key, **filehash** finds the object on the disk, loads the value into R and returns it to the user. The package offers two formats for storing data on the disk: The values can be stored (1) concatenated together in a single file or (2) separately as a directory of files.

2 Related R packages

There are other packages on CRAN designed specifically to help users work with large datasets. Two packages that come immediately to mind are the **g.data** package by David Brahm (Brahm, 2002) and the **biglm** package by Thomas Lumley. The **g.data** package takes advantage of the lazy evaluation mechanism in R via the `delayedAssign` function. Briefly, objects are loaded into R as promises to load the actual data associated with an object name. The first time an object is requested, the promise is evaluated and the data are loaded. From then on, the data reside in memory. The mechanism used in **g.data** is similar to the one used by the lazy-loaded databases described in Ripley (2004). The **biglm** package allows users to fit linear models on datasets that are too large to fit in memory. However, the **biglm** package does not provide methods for dealing with large datasets in general. The **filehash** package also draws inspiration from Luke Tierney's experimental **gdbm** package which implements a key-value database via the GNU dbm (GDBM) library. The use of GDBM creates an external dependence since the GDBM C library has to be compiled on each system. In addition, I encountered a problem where databases created on 32-bit machines could not be transferred to and read on 64-bit machines (and vice versa). However, with the increasing use of 64-bit machines in the future, it seems this problem will eventually go away.

The R Special Interest Group on Databases has developed a number of packages that provide an R interface to commonly used relational database management systems (RDBMS) such as MySQL (**RMySQL**), PostgreSQL (**RPgSQL**), and Oracle (**ROracle**). These packages use the S4 classes and generics defined in the **DBI** package and have the advantage that they offer much better database functionality, inherited via the use of a true database management system. However, this benefit comes with the cost of having to install and use third-party software. While installing an RDBMS may not be an issue—many systems have them pre-installed and the **RSQLite** package comes bundled with the source for the RDBMS—the need for the RDBMS and knowledge of structured query language (SQL) nevertheless adds some overhead. This overhead may serve as an impediment for users in need of a database for simpler applications.

3 Creating a filehash database

Databases can be created with **filehash** using the `dbCreate` function. The one required argument is the name of the database, which we call here "mydb".

```
> library(filehash)
> dbCreate("mydb")

[1] TRUE

> db <- dbInit("mydb")
```

You can also specify the `type` argument which controls how the database is represented on the backend. We will discuss the different backends in further detail later. For now, we use the default backend which is called "DB1".

Once the database is created, it must be initialized in order to be accessed. The `dbInit` function returns an S4 object inheriting from class "filehash". Since this is a newly created database, there are no objects in it.

4 Accessing a filehash database

The primary interface to filehash databases consists of the functions `dbFetch`, `dbInsert`, `dbExists`, `dbList`, and `dbDelete`. These functions are all generic—specific methods exists for each type of database backend. They all take as their first argument an object of class "filehash". To insert some data into the database we can simply call `dbInsert`

```
> dbInsert(db, "a", rnorm(100))
```

Here we have associated with the key "a" 100 standard normal random variates. We can retrieve those values with `dbFetch`.

```
> value <- dbFetch(db, "a")
> mean(value)

[1] 0.002912563
```

The function `dbList` lists all of the keys that are available in the database, `dbExists` tests to see if a given key is in the database, and `dbDelete` deletes a key-value pair from the database

```
> dbInsert(db, "b", 123)
> dbDelete(db, "a")
> dbList(db)

[1] "b"

> dbExists(db, "a")

[1] FALSE
```

While using functions like `dbInsert` and `dbFetch` is straightforward it can often be easier on the fingers to use standard R subset and accessor functions like `$`, `[[`, and `[`. Filehash databases have methods for these functions so that objects can be accessed in a more compact manner. Similarly, replacement methods for these functions are also available. The `[` function can be used to access multiple objects from the database, in which case a list is returned.

```
> db$a <- rnorm(100, 1)
> mean(db$a)

[1] 1.011141

> mean(db[["a"]])

[1] 1.011141

> db$b <- rnorm(100, 2)
> dbList(db)

[1] "a" "b"
```

For all of the accessor functions, only character indices are allowed. Numeric indices are caught and an error is given.

```
> e <- local({
+     err <- function(e) e
+     tryCatch(db[[1]], error = err)
+ })
> conditionMessage(e)

[1] "numeric indices not allowed"
```

Finally, there is method for the `with` generic function which operates much like using `with` on lists or environments.

The following three statements all return the same value.

```
> with(db, c(a = mean(a), b = mean(b)))

       a        b
1.011141 2.012793
```

When using `with`, the values of "a" and "b" are looked up in the database.

```
> sapply(db[c("a", "b")], mean)

       a        b
1.011141 2.012793
```

Here, using `[` on `db` returns a list with the values associated with "a" and "b". Then `sapply` is applied in the usual way on the returned list.

```
> unlist(lapply(db, mean))

       a        b
1.011141 2.012793
```

In the last statement we call `lapply` directly on the "filehash" object. The **filehash** package defines a method for `lapply` that allows the user to apply a function on all the elements of a database directly. The method essentially loops through all the keys in the database, loads each object separately and applies the supplied function to each object. `lapply` returns a named list with each element being the result of applying the supplied function to an object in the database. There is an argument `keep.names` to the `lapply` method which, if set to `FALSE`, will drop all the names from the list.

5 Loading filehash databases

An alternative way of working with a filehash database is to load it into an environment and access the element names directly, without having to use any of the accessor functions. The **filehash** function `dbLoad` works much like the standard R `load` function except that `dbLoad` loads active bindings into a given environment rather than the actual data. The active bindings are created via the `makeActiveBinding` function in the **base** package. `dbLoad` takes a filehash database and creates symbols in an environment corresponding to the keys in the database. It then calls `makeActiveBinding` to associate with each key a function which loads the data associated with a given key. Conceptually, active bindings are like pointers to the database. After calling `dbLoad`, anytime an object with an active binding is accessed the associated function (installed by `makeActiveBinding`) loads the data from the database.

We can create a simple database to demonstrate the active binding mechanism.

```
> dbCreate("testDB")
```

```
[1] TRUE
```

```
> db <- dbInit("testDB")
> db$x <- rnorm(100)
> db$y <- runif(100)
> db$a <- letters
> dbLoad(db)
> ls()
```

```
[1] "a"  "db" "x"  "y"
```

Notice that we appear to have some additional objects in our workspace. However, the values of these objects are not stored in memory—they are stored in the database. When one of the objects is accessed, the value is automatically loaded from the database.

```
> mean(y)
```

```
[1] 0.5118129
```

```
> sort(a)
```

```
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
[15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

If I assign a different value to one of these objects, its associated value is updated in the database via the active binding mechanism.

```
> y <- rnorm(100, 2)
> mean(y)
```

```
[1] 2.010489
```

If I subsequently remove the database and reload it later, the updated value for "y" persists.

```
> rm(list = ls())
> db <- dbInit("testDB")
> dbLoad(db)
> ls()
```

```
[1] "a"  "db" "x"  "y"
```

```
> mean(y)
```

```
[1] 2.010489
```

Perhaps one disadvantage of the active binding approach taken here is that whenever an object is accessed, the data must be reloaded into R. This behavior is distinctly different from the the delayed assignment approach taken in **g.data** where an object must only be loaded once and then is subsequently in memory. However, when using delayed assignments, if one cycles through all of the objects in the database, one could eventually exhaust the available memory.

6 Cleaning up

When a database is initialized using the default "DB1" format, a file connection is opened for reading and writing to the database file on the disk. This file connection remains open until the database is closed via `dbDisconnect` or the database object in R is removed. Since there is a hard limit on the number of file connections that can be open at once, some protection is need to make sure that file connections are close properly.

Upon initialization, the database stores the file connection number in an environment and registers a finalizer to close the connection. Once a database is closed or removed and garbage collection occurs, the finalizer is called and the file connection closed appropriately.

7 Other filehash utilities

There are a few other utilities included with the **filehash** package. Two of the utilities, `dumpObjects` and `dumpImage`, are analogues of `save` and `save.image`. Rather than save objects to an R workspace, `dumpObjects` saves the given objects to a "filehash" database so that in the future, individual objects can be reloaded if desired. Similarly, `dumpImage` saves the entire workspace to a "filehash" database.

The function `dumpList` takes a list and creates a "filehash" database with values from the list. The list must have a non-empty name for every element in order for `dumpList` to succeed. `dumpDF` creates a "filehash" database from a data frame where each column of the data frame is an element in the database. Essentially, `dumpDF` converts the data frame to a list and calls `dumpList`.

8 Filehash database backends

Currently, the **filehash** package can represent databases in two different formats. The default format is called "DB1" and it stores the keys and values in a single file. From experience, this format works well overall but can be a little slow to initialize when there are many thousands of keys. Briefly, the "filehash" object in R stores a map which associates keys with a byte location in the database file where the corresponding value is stored. Given the byte location, we can `seek` to that location in the file and read the data directly. Before reading in the data, a check is made to make sure that the map is up to date. This format depends critically on having a working `ftell` at the system level and a crude check is made when trying to initialize a database of this format.

The second format is called "RDS" and it stores objects as separate files on the disk in a directory with the same name as the database. This format is the most straightforward and simple of the available formats. When a request is made for a specific key, **filehash** finds the appropriate file in the directory and reads the file into R. The only catch is that on operating systems that use case-insensitive file names, objects whose names differ only in case will collide on the filesystem. To workaround this, object names with capital letters are stored with mangled names on the disk. An advantage of this format is that most of the organizational work is delegated to the filesystem.

There is a third format called "DB" and it is a predecessor of the "DB1" format. This format is like the "DB1" format except the map which associates keys to byte locations is stored in a separate file. Therefore, each database is represented by two separate files—an index file and a data file. This format is retained for back compatibility but users should generally try to use the "DB1" format instead.

9 Extending filehash

The **filehash** package has a mechanism for developing new backend formats, should the need arise. The function `registerFormatDB` can be used to make **filehash** aware of a new database format that may be implemented in a separate R package or a file. `registerFormatDB` takes two arguments: a `name` for the new format (like "DB1" or "RDS") and a list of functions. The list should contain two functions: one function named "create" for creating a database, given the database name, and another function named "initialize" for initializing the database. In addition, one needs to define methods for `dbInsert`, `dbFetch`, etc.

A list of available backend formats can be obtained via the `filehashFormats` function. Upon registering a new backend format, the new format will be listed when `filehashFormats` is called.

The interface for registering new backend formats is still experimental and could change in the future.

10 Discussion

The **filehash** package has been designed be useful in both a programming setting and an interactive setting. Its main purpose is to allow for simpler interaction with large datasets where simultaneous access to the full dataset is not needed. While the package may not be optimal for all settings, one goal was to write a simple package in pure R that users to could install with minimal overhead. In the future I hope to add

functionality for interacting with databases stored on remote computers and perhaps incorporate a "real" database backend. Some work has already begun on developing a backend based on the **RSQLite** package.

References

Brahm, D. E. (2002), "Delayed Data Packages," *R News*, 2, 11–12.

Chambers, J. M. (1991), "Data Management in S," Tech. Rep. 99, AT&T Bell Laboratories Statistics Research, http://stat.bell-labs.com/doc/93.15.ps.

— (1998), *Programming with Data: A Guide to the S Language*, Springer.

Ripley, B. D. (2004), "Lazy Loading and Packages in R 2.0.0," *R News*, 4, 2–4.

Temple Lang, D. (2002), *RObjectTables: User-level attach()'able table support*, r package version 0.3-1.