



JOHNS HOPKINS  
BLOOMBERG  
SCHOOL of PUBLIC HEALTH

---

Johns Hopkins University, Dept. of Biostatistics Working Papers

---

6-1-2006

# INTERACTING WITH DATA USING THE FILEHASH PACKAGE FOR R

Roger Peng

*Department of Biostatistics, Johns Hopkins Bloomberg School of Public Health, [rpeng@jhsph.edu](mailto:rpeng@jhsph.edu)*

---

## Suggested Citation

Peng, Roger, "INTERACTING WITH DATA USING THE FILEHASH PACKAGE FOR R" (June 2006). *Johns Hopkins University, Dept. of Biostatistics Working Papers*. Working Paper 108.  
<http://biostats.bepress.com/jhubiostat/paper108>

This working paper is hosted by The Berkeley Electronic Press (bepress) and may not be commercially reproduced without the permission of the copyright holder.

Copyright © 2011 by the authors

# 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 sometimes 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.

The **filehash** package implements a simple key-value database in R. 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 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. R handles the job of finding the appropriate value for “x” by searching through a series of environments, including the namespaces of the packages on the search list.

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.

There are other packages on CRAN designed to help users work with large datasets. The 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 **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 deal with large datasets in general.

## 2 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”. 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”.

```
> library(filehash)

Simple key-value database (version 0.6-2)

> dbCreate("mydb")

[1] TRUE

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

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.

## 3 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 exist 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))

[1] TRUE
```

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)

[1] TRUE
```

```
> dbDelete(db, "a")

[1] TRUE
```

```
> 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] "use of 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.

## 4 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 actual data objects. 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 symbol/key a function which loads the data associated with a given symbol/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 everytime. This behavior is distinctly different from the the delayed assignment approach taken in **g.data** where the data must 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.

## 5 A larger example

As an example of using `filehash` for interacting with a much larger database, we will use some air pollution and weather data that were originally collected as part of the National Morbidity, Mortality, and Air Pollution Study (Samet et al., 2000a,b). The full database, which includes daily mortality data is available from the `NMMAPSdata` package (Peng and Welty, 2004). For the purposes of this example, we have extracted just the air pollution and weather data.

Briefly, the database consists of data from 108 cities in the United States. For each city, there is a data frame of 5114 rows and 274 columns. The database contains daily measurements for the U.S. Environmental Protection Agency's "criteria" pollutants—particulate matter, nitrogen dioxide, sulphur dioxide, ozone, carbon monoxide—as well as temperature, dew point temperature, and relative humidity. For the gaseous pollutants, hourly values are available. For particulate matter (both  $PM_{10}$  and  $PM_{2.5}$ ), 24 hour averages are available. The filehash database can be downloaded from <http://www.biostat.jhsph.edu/~rpeng/RR/>.

Here, we load the database and list the first 10 cities.

```
> db <- dbInit("metpoll")
> dbLoad(db)
> ls()[1:10]

[1] "akr"  "albu" "anch" "arlv" "atla" "aust" "bake" "balt"
[9] "batr" "bidd"
```

The keys for the database are abbreviated versions of the city names. The full city names can be found in the "citynames.csv" data frame available from the author's website.

Normally, it would have been impossible to load all of the data from all of the cities at once, but `dbLoad` merely creates active bindings for the data. As our session progresses, data for each city will be dynamically loaded as needed.

Now we can simply access each city's data frame by its abbreviated name, as if every data frame had already been loaded into the workspace. For example we can regress daily  $PM_{10}$  on temperature in Chicago, Illinois,

```
> lm(pm10tmean ~ tmpd, data = chic)

Call:
lm(formula = pm10tmean ~ tmpd, data = chic)

Coefficients:
(Intercept)          tmpd
   -18.0002         0.3565
```

or in New York City, New York.

```
> lm(pm10tmean ~ tmpd, data = ny)

Call:
lm(formula = pm10tmean ~ tmpd, data = ny)

Coefficients:
(Intercept)          tmpd
   -17.3641         0.3121
```

Meanwhile, our overall memory usage has only increased slightly since initially starting up R.

```
> gc()[, 1:2]
      used (Mb)
Ncells 279990 15.0
Vcells 144546  1.2
```

The **filehash** package is most useful in situations where one does not require simultaneous access to all the data at once. In that situation, one would have to have all the data loaded into memory in order to operate on it (e.g. fit models, compute statistics) and **filehash** would not be able to help. However, in situations where one would like to have ready access to the data and can operate on single elements or perhaps small groups of elements at a time, the **filehash** package may be of use.

## 6 Filehash database backends

Currently, the **filehash** package can represent databases in three 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 database 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 work around 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.

The last format is 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.

## 7 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.

## 8 Discussion

The **filehash** package has been designed to 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 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.

Peng, R. D. and Welty, L. J. (2004), “The NMMAPSdata Package,” *R News*, 4, 10–14.

Samet, J. M., Dominici, F., Zeger, S. L., Schwartz, J., and Dockery, D. W. (2000a), *The National Morbidity, Mortality, and Air Pollution Study, Part I: Methods and Methodological Issues*, Health Effects Institute, Cambridge MA.

Samet, J. M., Zeger, S. L., Dominici, F., Curriero, F., Coursac, I., Dockery, D. W., Schwartz, J., and Zanobetti, A. (2000b), *The National Morbidity, Mortality, and Air Pollution Study, Part II: Morbidity and Mortality from Air Pollution in the United States*, Health Effects Institute, Cambridge, MA.

