Loading data on demand

Thomas Lumley

Dept of Biostatistics,
University of Washington

R Core Development Team

useR — Rennes — 2009–7–9
Integrated database/statistics packages

Charlton Heston brings SAS down from Mt Sinai
Relational databases and R

Relational databases are the natural habitat of large data sets.

R has good interfaces: RJDBC, RODBC, R-DBI

No need to have database storage built in to R.
Use case: moderately large data

Data sets from national surveys or large cohort studies have $10^3$–$10^5$ observations and $10^2$–$10^4$ variables.

Loading the entire data set into R is inconvenient, especially on 32-bit systems.

A single computation will typically use only $10^0$–$10^1$ variables, fits easily in memory.

Data entry and data management is easier in a relational database.

Example: Behavioral Risk Factor Surveillance System, a telephone survey of 450,000 individuals in US.
User interface

Goal: allow an object that wraps a database to be used in place of a data frame

Can’t just write methods: `model.frame` doesn’t dispatch on the `data=` argument, relies on internal structure of data frames.

Need to load data for relevant variables and then call methods based on data frames.
Simple case

doSomething <- function(formula, database) {
  varlist <- paste( all.vars(formula), collapse ="", ")
  query <- paste("SELECT", varlist, "FROM", database$tablename)
  dataframe <- dbGetQuery(database$connection, query)
  ## now actually do Something
  fitModel(formula, dataframe)
}

First construct a query to load all the variables you need, then submit it to the database to get the data frame, then proceed as usual.

Refinements: some variables may be in memory, not in the database, we may need to define new variables, we may want to wrap an existing set of code.
Wrapping existing code

Define a generic function to dispatch on the second (data) argument

```r
doSomething <- function(formula, data, ...){
    UseMethod("doSomething", data)
}
```

and set the existing function as the default method

```r
doSOMething.database <- (formula, database, ...){
    varlist <- paste( all.vars(formula), collapse="", ")
    query <- paste("SELECT", varlist, "FROM", database$tablename)
    dataframe <- dbGetQuery(database$connection, query)
    ## now actually do Something
    doSomething(formula, dataframe, ...)
}
```
Allowing variables in memory

To allow the function to pick up variables from memory, just restrict the database query to variables that are in the database

\[
dbvars <- \text{names(dbGetQuery(conn, "select * from table limit 1"))}
formulavars <- \text{all.vars(formula)}
varlist <- paste( intersect(formulavars, dbvars), collapse="", ")
\]

[In practice we would find the list of names in the database first and cache it in an R object]

Now \text{model.frame()} will automatically pick up variables in memory, unless they are masked by variables in the database table — the same situation as for data frames.
Allowing updates

Three approaches:

- Write new variables into the database with SQL code: needs permission, reference semantics, restricted to SQL syntax

- Create new variables in memory and save to the database: needs permission, reference semantics, high network traffic

- Store the expressions and create new variables on data load: wasted effort

Since data transfer will be the bottleneck, the third strategy is not really a waste of effort.
Design

A database object stores the connection, table name, new variable information

New variables are created with the `update` method

```r
mydata <- update(mydata, avgchol = (chol1 + chol2)/2, 
                 hibp = (systolic>140) | (diastolic>90) )
```

- An expression can use variables in the database or previously defined ones, but not simultaneously defined ones.
- Multiple `update()`s give a stack of lists of expressions
- Use `all.vars` going down the stack to find which variables to query from the database
- Return up the stack, evaluating the expressions with `eval()` and adding variables to the data frame

Implemented in `survey`, `mitools` packages, using R-DBI and RODBC interfaces.
Wrapping existing code

Survey design objects contain metadata, and data frame in `$variables` slot

Database-backed design objects have no `$variables` slot, contain database connection information, inherit from survey design objects.

Each method loads data into the `$variables` slot, calls `NextMethod` to dispatch.
Wrapping existing code

> svymean
function (x, design, na.rm = FALSE, ...)
{
  .svycheck(design)
  UseMethod("svymean", design)
}

> survey:::svymean.DBIsvydesign
function (x, design, ...)
{
  design$variables <- getvars(x, design$db$connection,
    design$db$ tablename,
    updates = design$updates)
  NextMethod("svymean", design)
}
Subsets

Could add a subset argument, translated into a SQL `WHERE` clause.

Need minor changes to convert R to SQL syntax: inorder traversal of parsed syntax tree for R expression, emit SQL code.

```r
> Rexpr<-quote( sex == "MALE" & state %in% c("MI","MO","ME","MA"))
> sqlexpr(Rexpr)
"( (sex=="MALE") AND (state IN ("MI","MO","ME","MA"))) "
```

Not in packages at the moment.
Example: BRFSS

On 1Gb laptop, complete BRFSS data cannot be loaded (450,000 records)

On any 32-bit system, BRFSS dataset is too big for convenient use (≈ 1.5Gb in memory).

A few variables can easily be loaded.

With database-backed design using SQLite, overhead of data loading is about 2 minutes and computer remains responsive.