Package 'dbframe'

February 1, 2016

Type Package

Title An R to SQL interface

Date 2010-10-02

Version 0.4.0

Author Gray Calhoun

Depends methods, utils, DBI, xtable

Suggests testthat, RSQLite, RSQLite.extfuns

Maintainer Gray Calhoun <gray@clhn.org>

Description Some code to conveniently work with databases.

License MIT + file LICENSE

LazyLoad yes

LazyData yes

R topics documented:

`dbframe-package` *An overview of the dbframe package*

Description

This package and documentation are very much under development. Right now, it only provides an interface with SQLite databases and has untested code that will try to guess the right interface for other databases. The function `dbframe` creates a new dbframe object associated with one of the tables in an SQLite database. Queries on the database can then be executed through the `dbframe` object's methods. This package also provides some convenient functions for printing and manipulating data frames, but its main role is to make it easier to write queries.

Details

This package is a collection of functions that I'm developing to work more easily with databases. I find it annoying to have to type SQL statements into functions like `dbSendQuery`, etc. to explicitly pull data from a database into an R dataframe, especially since I often forget the details of SQL. When the database is updated regularly, this is especially annoying since the R dataframe gets out of date and the SQL statement needs to be executed again. This package is my attempt automate this sort of routine use of SQL. I've tried to make the syntax look more "R-like" than, eg, DBI. Please let me know if you have any suggestions for functionality, appearance, or syntax.

Queries: So far, I've just implemented `insert` and `select` queries. `insert` allows you to insert new records into a database from an R data frame. `select` allows you to retrieve those records from the data base. `select` also allows some basic data analysis, more or less anything that is available as an aggregate function in the SQL database.

Notes on implementation: I'm attempting to make this package a literate program (Knuth, 1984). That means that the source code is embedded in the documentation. The "implementation" section of each help page presents and discusses the source code for its functions. Right now, the formatting is pretty rudimentary, but I hope to improve the appearance at some point.

This package is currently in alpha. I use it in my research and will add functionality as I need it, but I don't have the time to extend it in some obviously important directions that are unrelated to my workflow. I plan to do so in the future, though, and even now would like to implement specific requests if people are interested.

Author(s)

Gray Calhoun <`gcalhoun@iastate.edu`>

References

Knuth, Donald E., Literate Programming. *The Computer Journal*, **27**(2):97–111, 1984.

Examples

```
filename <- tempfile(fileext = ".db")
example.dbframe <- dbframe("package1", dbname = filename, clear = TRUE)
data(chickwts)
insert(example.dbframe) <- chickwts
select(example.dbframe, "avg(weight)", group.by = "feed")
unlink(filename)
```

`booktabs` *Construct a nice table from a data frame*

Description

Constructs an attractive LaTeX table (using booktabs) from an arbitrary data frame.

Usage

```
booktabs(dframe, align = "l", digits = 1, numberformat = FALSE,
         purgeduplicates = TRUE, tabular.environment = "tabularx",
         scientific = FALSE, include.rownames = FALSE,
         sanitize.text.function = function(x) x, drop = NULL,...)
```

Arguments

`dframe`	A data frame or an object that can be coerced to a data.frame.
`align`	A character vector specifying each column's alignment. Each element should be "l", "c", or "r".
`digits`	A vector of integers specifying the number of digits to display for each column.
`numberformat`	logical vector indicating which columns should be formatted as numbers. This adds padding to the left side of the numbers so that the column aligns on the decimal point, switches to math mode to turn off old-style numbering (if needed), and converts hyphens into minus signs.
`purgeduplicates`	A logical vector that labels which columns should have duplicate entries removed.
`tabular.environment`	Defaults to using tabularx in LaTeX.
`scientific`	FALSE unless you want to use scientific notation in the tables.
`include.rownames`	Logical indicating whether should the data frame's rownames should be included. Probably not.
`sanitize.text.function`	Disables escaping out backslashes, etc. This is the same as in '`xtable`'.
`drop`	A character vector indicating some columns to omit. This can be useful if those columns are used for sorting or to create derived columns.
`...`	Additional arguments to pass to `xtable`.

Details

This function uses `xtable` to generate LaTeX code, then modifies the table. The containing Latex document should use the "booktabs" package.

Value

A character object containing LaTeX code for a table.

Implementation

The basic implementation is straightforward.

```
<<*>>=
    booktabs <- function(dframe, align = "l", digits = 1,
                         numberformat = FALSE, purgeduplicates = TRUE,
                         tabular.environment = "tabularx",
                         include.rownames = FALSE,
                         sanitize.text.function = function(x) x,
                         drop = NULL,...) {
      <<Define platform independent null file>>
      <<Format arguments>>
      return(<<Assemble Latex code for table>>)
    }
```

To assemple the Latex code, we just call '`xtable`' on the data frame, then substitute out the first line to make the table span the entire page. Note that '`devnull`' is defined elsewhere in the package.

```
<<Assemble Latex code for table>>=
    gsub(sprintf("\\\\begin\\{%s\\}", tabular.environment),
         sprintf("\\\\begin\\{%s\\}\\{\\\\textwidth\\}", tabular.environment),
         print(xtable(dframe, align = align, digits = digits,...),
               file = devnull, floating = FALSE,
               add.to.row = list(pos=list(-1, 0, nrow(dframe)),
                 command = c("\\toprule ", "\\midrule ", "\\bottomrule ")),
               tabular.environment = tabular.environment,
               sanitize.text.function = sanitize.text.function,
               include.rownames = include.rownames, hline.after = NULL))
```

The variable '`devnull`' is defined to be a platform independent '`/dev/null`':

```
<<Define platform independent null file>>=
    devnull <- switch(Sys.info()["sysname"],
      Windows = "NUL",
      Linux   = "/dev/null",
      Darwin  = "/dev/null",
     {warning("Your OS is not explicitly supported; we'll assume /dev/null exists.")
       "/dev/null"})
```

A little bit of routine reformatting needs to happen before calling '`xtable`'.

```
<<Format arguments>>=
    dframe <- as.data.frame(dframe)
    <<Drop user-specified columns>>
    <<Correct dimensions of arguments>>
    <<Pad formatting columns to accomodate xtables handling of rownames>>
    <<Reformat numeric columns>>
    <<Remove duplicates from specified columns>>
```

The user can choose to leave out some of the columns. This can be useful if there are columns that are important for sorting the data frame, but are not of interest on their own.

```
<<Drop user-specified columns>>=
    if (!is.null(drop)) {
      columnnames <- names(dframe)
      if (!all(drop %in% columnnames)) {
        warning("'drop' contains some columns not in 'dframe'")
      }
      dframe <- dframe[, setdiff(names(dframe), drop), drop = FALSE]
    }
```

For conveneince, we let arguments that affect column-by-column formatting be written as a single value if the same value applies to each column. In that case, we repeat the value the correct number of times.

```
<<Correct dimensions of arguments>>=
    ncol <- ncol(dframe) + include.rownames
    if (length(align) == 1) align <- rep(align, ncol)
    if (length(digits) == 1) digits <- rep(digits, ncol)
    if (length(numberformat) == 1) numberformat <- rep(numberformat, ncol)
    if (length(purgeduplicates) == 1)
      purgeduplicates <- rep(purgeduplicates, ncol)
```

The way ‘xtable’ handles the row names is kind of annoying: alignment and digits need to specified for it, even if the row names will not be shown. To avoid doing that, we pad the necessary arguments if the row names aren’t going to be shown.

```
<<Pad formatting columns to accomodate xtables handling of rownames>>=
    if (!include.rownames) {
      align <- c("l", align)
      digits <- c(0, digits)
    }
```

Columns that contain numbers are reformatted to align at the decimal point and use the correct minus sign.

```
<<Reformat numeric columns>>=
    dframe[,numberformat] <- lapply(which(numberformat), function (i) {
      emptyRows <- is.na(dframe[,i])
      rowTex <- rep("", length(emptyRows))
      rowTex[!emptyRows] <-
        gsub("-", "\\\\!\\\\!-", sprintf("$%s$", gsub(" ", "\\\\enskip",
           format(round(as.numeric(dframe[!emptyRows,i]),
                        digits[i + !include.rownames])))))
      rowTex
    })
```

Finally, we remove duplicates in a slightly clever way

```
<<Remove duplicates from specified columns>>=
    repeats <- function(x) c(FALSE, x[-1] == x[seq_len(length(x) - 1)])
    purgeindex <- which(purgeduplicates)
    for (i in rev(seq_along(purgeindex))) {
      dframe[repeats(dframe[[i]]) &
             duplicated(dframe[, purgeindex[seq_len(i)], drop = FALSE]),
             purgeindex[i]] <- NA
    }
```

Unit tests

```
<<test-booktabs.R>>=
    library(testthat)
    library(xtable)
    filename <- tempfile(fileext = ".db")

    data(longley)
    test_that("booktabs executes at all", {
      expect_that(booktabs(longley), is_a("character"))
    })

    test_that("Columns that are labeled 'numberformat' are formatted", {
      d <- data.frame(x = c(-1.324, 0.93), y = c(10.443, 1.235))
      expect_that(booktabs(d, numberformat = TRUE,
                           purgeduplicates = FALSE, digits = 2, align = "c"),
        prints_text("\$\\\\\\\\\\!\\\\\\\\\\!-1.32\$ & \$10.44\$"))
      expect_that(booktabs(d, numberformat = TRUE,
                           purgeduplicates = FALSE, digits = 2, align = "c"),
        prints_text("\$\\\\\\\\enskip0.93\$ & \$\\\\\\\\enskip1.24\$"))
    })

    test_that("Argument checking works as expected", {
      expect_that(booktabs(longley, drop = "WXYZ"),
        gives_warning("'drop' contains some columns not in 'dframe'"))
    })
```

Author(s)

Gray Calhoun <gcalhoun@iastate.edu>

See Also

`xtable`, `print.xtable`

Examples

```
data(chickwts)
cat(booktabs(head(chickwts)))
```

---

clear   *Remove a table from the database*

Description

The function ‘clear’ removes the tables associated with a collection of ‘dbframe’ objects from their data bases.

Usage

```
clear(...)
```

Arguments

`...`	Each argument listed should be a dbframe object.

Details

The table is removed and not just emptied, so the column names and types are discarded as well.

Implementation

The first step is to define a generic '`clear`' function. The actual function is a method for the '`dbframe`' class. Methods for other classes could be defined as well.

```
<<*>>=
    setGeneric("clear", function(...)
               standardGeneric("clear"), signature = "...")

    setMethod("clear", signature = "dbframe", function(...) {
      x <- list(...)
      sapply(x, function(y) {
        stopifnot(!readonly(y))
        dbc <- dbConnect(y)
        results <- <<Remove table referenced by "y" from database>>
        dbDisconnect(dbc)
        results
      })})
```

Removing the table is pretty easy since we can just use methods defined by DBI.

```
<<Remove table referenced by "y" from database>>=
res <-
  if (is.linked(y)) {
    dbRemoveTable(dbc, tablename(y),...)
  } else {
    FALSE
  }
```

Author(s)

Gray Calhoun <`gcalhoun@iastate.edu`>

See Also

dbRemoveTable

Examples

```
filename <- tempfile(fileext = ".db")
example.dbframe <- dbframe("clear1", dbname = filename)
example2.dbframe <- dbframe("clear2", dbname = filename)
clear(example.dbframe, example2.dbframe)

data(chickwts)
insert(example.dbframe) <- chickwts
head(example.dbframe)
```

```
clear(example.dbframe)
head(example.dbframe)
unlink(filename)
```

---

dbframe — *Create a* `dbframe` *object*

Description

This is a constructor for the dbframe class. I've only made a constructor for SQLite databases, but will (probably) add others in the future.

Usage

```
dbframe(table, dbname = NULL, dbdriver = "SQLite", data = NULL,
        readonly = FALSE, clear = FALSE,...)
dbframe_sqlite(table, dbname, readonly = FALSE,...)
dbframe_sqlite_temporary(table, dbname = ":memory:", readonly = FALSE,...)
dbframe_unknown(table, readonly = FALSE,...)
```

Arguments

`table`	A character string. This will typically be the name of the table in the SQLite database, but it can also be the name of a view, or an SQL select statement that defines a query.
`dbdriver`	The name of the database driver to use. Right now, only "SQLite" is supported.
`dbname`	The file name of the SQLite database.
`data`	An optional data frame to insert into the table.
`readonly`	Logical: prevent the user from inserting to or clearing the linked table.
`clear`	Logical: if the table already exists, should it be removed before making the link?
`...`	Optional arguments that will be stored (if necessary) and added to "dbConnect"

Details

This function constructs a new '`dbframe`' objet referencing the table "table" in the database file "dbname". If the argument '`data`' is not null, its values are inserted into the table. If '`clear`' is '`TRUE`', any existing version of the table is removed first.

'`dbframe{\char95}unknown`' is a blind guess at a general way to implement a generic constructor. It may or may not work and I haven't tested it yet. The optional arguments are stored (as a list) as a slot of the '`dbframe`' object and are passed to '`dbConnect`'. If you use it, please let me know and suggest changes/improvements.

Value

Returns a `dbframe` object.

Implementation

These functions are pretty simple; ‘dbframe’ creates a new object corresponding to “dbdriver” and then removes or inserts data to the table according to the arguments. If “dbdriver” hasn’t been implemented yet, we initialize a “dbframe_unknown” object that will try to guess how to open a connection to the database (I have no idea if this will work well or not; if you use it, please be careful and give me feedback on how well it goes).

```
<<*>>=
    dbframe <- function(table, dbname = NULL, dbdriver = "SQLite",
                        data = NULL, readonly = FALSE, clear = FALSE,...) {
      x <- switch(dbdriver,
                  "SQLite" = {
                    if (is.null(dbname)) {
                      warning("'dbname' is null; setting to ':memory:'")
                      dbname <- ":memory:"
                    }
                    if (dbname %in% c(":memory:", "")) {
                      dbframe_sqlite_temporary(table, dbname, readonly,...)
                    } else {
                      dbframe_sqlite(table, dbname, readonly,...)
                    }
                  },
                  dbframe_unknown(table, readonly,...))
      <<Clear old table and insert new data>>
      return(x)
    }

    <<Define "dbframe_sqlite">>
    <<Define "dbframe_sqlite_temporary">>
    <<Define "dbframe_unknown">>
```

Manipulating the ‘dbframe’ object after creating it is easy and we just use the existing ‘dbframe’ methods.

```
<<Clear old table and insert new data>>=
    if (clear) clear.result <- clear(x)
    if (!is.null(data)) insert(x) <- data
```

‘dbframe{\char95}sqlite’ makes sure that the right ‘RSQLite’ libraries are loaded and creates a new ‘dbframe{\char95}sqlite’ object.

```
<<Define "dbframe_sqlite">>=
    dbframe_sqlite <- function(table, dbname, readonly = FALSE,...) {
      <<Load SQLite libraries>>
      return(new("dbframe_sqlite", table = unname(table), rowid = integer(),
                 dbname = unname(dbname), readonly = unname(readonly),
                 dbConnect.arguments = list(...)))
    }

<<Define "dbframe_sqlite_temporary">>=
    dbframe_sqlite_temporary <-
      function(table, dbname = ":memory:", readonly = FALSE,...)
      stop("Temporary SQLite databases aren't implemented.")
```

The ‘dbframe’ package only suggests the ‘RSQLite’ and ‘RSQLite.extfuns’ packages; it doesn’t depend formally on them. Consequently those packages aren’t loaded when ‘dbframe’ is first load; I put off loading those packages until it is clear that the user wants to interface with an SQLite database, which happens when a ‘dbframe{\char95}sqlite’ object is created. As things stand now, this delay doesn’t do very much. When I add support for other SQL databases, though, I plan to use the same sort of approach and delay loading the necessary libraries until the appropriate constructor is called. This saves us from loading several database libraries when we’re only going to use one of them, or from requiring the user to load the libraries explicity.

```
<<Load SQLite libraries>>=
    require(RSQLite)
    require(RSQLite.extfuns)
```

‘dbframe{\char95}unknown’ is kind of a crapshoot. It stores the optional arguments in a list so that it can pass them to ‘dbConnect’ as needed.

```
<<Define "dbframe_unknown">>=
    dbframe_unknown <- function(table, readonly = FALSE,...) {
      return(new("dbframe", table = unname(table),
                 readonly = unname(readonly),
                 dbConnect.arguments = list(...)))
    }
```

Unit Tests

```
<<test-dbframe.R>>=
    library(testthat)

    data(morley)
    filename <- tempfile(fileext = ".db")

    test_that("Basic constructor works", {
      d1 <- dbframe("test1", dbname = filename, data = morley)
      expect_that(d1, is_a("dbframe_sqlite"))
      expect_that(morley, is_equivalent_to(select(d1)))
    })

    test_that("Simple methods work", {
      d1 <- dbframe("test2", dbname = filename, data = morley)
      expect_that(nrow(morley), equals(nrow(d1)))
    })
```

Author(s)

Gray Calhoun <gcalhoun@iastate.edu>

See Also

dbframe-class, clear, insert<-

Examples

```
data(chickwts)
filename <- tempfile(fileext = ".db")
```

```
example <- dbframe("dbframe1", dbname = "filename", dbdriver = "SQLite",
                   data = chickwts)
tail(example)
## an example where "table" is a select statement on its own





## clean up
unlink(filename)
```

`dbframe-class` *Class* `"dbframe"`

Description

Each "dbframe" object references a particular table inside an SQL database. The class and its methods are designed to query and manipulate the table easily inside R.

Objects from the Class

Objects can be created by calls of the form `new("dbframe", ...)`, but I recommend that you use the constructor '`dbframe`'.

The '`dbframe{\char95}sqlite`' class creates a link to a table in an SQLite database, and '`dbframe`' acts as a superclass. '`dbframe{\char95}sqlite{\char95}temporary`' implements temporary SQLite databases. I'd like to add other classes that store links to other databases (i.e. MySQL and PostgreSQL) as well. Until then, '`dbframe`' also works as an experimental class that tries to guess how to open a connection to the right database—please let me know how well it works, or if it works at all.

Slots

`dbframe`**:**

`dbConnect.arguments`**:** Object of class `"list"`, a list of arguments to pass to '`dbConnect`' when it's time to connect to the database.

`readonly`**:** Object of class `"logical"` that indicates whether the user is allowed to write to the database. Be careful! This is implemented only as a safeguard against carelessness; it is still very easy to write to the table.

`table`**:** Object of class `"character"`, the name of the table associated with this '`dbframe`' object.

Additional slots for `dbframe_sqlite`**:**

`dbname`**:** Object of class `"character"`, the filename of the associated SQLite database.

`rowid`**:** Object of class `"integer"`, the rowid of the last value inserted into the table.

Methods

`dbframe`**:**

clear `signature(... = "dbframe")`: Remove the table corresponding to the '`dbframe`' object from its database. Please see the '`clear`' documentation for details.

dbname `signature(x = "dbframe_sqlite")`: Returns the filename of the SQLite database associated with the object.

```
<<Define "dbname" method>>=
    setMethod("dbname", signature = c("dbframe"), function(x) x@dbname)
```

dbConnect `signature(drv = "dbframe")`: Establishes a connection with the database associated with the object.

```
<<Define "dbConnect" method for "dbframe">>=
    setMethod("dbConnect", signature = "dbframe",
              definition = function(drv,...)
              do.call("dbConnect", drv@dbConnect.arguments))
```

insert<- `signature(x = "dbframe")`: Insert records into the ‘`dbframe`’ table. Please see individual documentation for details.

readonly `signature(x = "dbframe")`: Determine whether the dbframe object is read-only.

```
<<Define "readonly" method>>=
   setMethod("readonly", signature = c("dbframe"), function(x) x@readonly)
```

select Query the ‘`dbframe`’ table. Please see individual documentation for details.

tablename `signature(x = "dbframe")`: Returns the table name associated with the object.

```
<<Define "tablename" method>>=
    setMethod("tablename", signature = c("dbframe"), function(x) x@table)
```

is.linked `signature(x = "dbframe")`: Check whether the table associated with the ‘`dbframe`’ exists.

```
<<Define "is.linked" method>>=
    setMethod("is.linked", signature = c("dbframe"), function(x,...) {
      dbc <- dbConnect(x,...)
      answer <- tablename(x) %in% dbListTables(dbc)
      dbDisconnect(dbc)
      return(answer)
    })
```

For `dbframe_sqlite`:

dbConnect `signature(drv = "dbframe_sqlite")`: Establishes a connection with the database associated with the object.

```
<<Define "dbConnect" method for "dbframe_sqlite">>=
    setMethod("dbConnect", signature = "dbframe_sqlite",
      definition = function(drv,...) return(do.call("dbConnect",
        c(drv = "SQLite", dbname = dbname(drv), list(...),
          dbConnect.arguments = drv@dbConnect.arguments))))
```

rowid `signature(x = "dbframe_sqlite")`: Returns the rowid associated with the last insert.

```
<<Define "rowid" method>>=
    setMethod("rowid", signature = c("dbframe_sqlite"),
              function(x,...) x@rowid)
```

rowid<- `signature(x = "dbframe_sqlite")`: Assigns the rowid associated with the last insert.

```
<<Define "rowid<-" method>>=
   setMethod("rowid<-", signature = c("dbframe_sqlite"), function(x,...,value) {
      x@rowid <- as.integer(value)
      return(x)})
```

S3 Methods

as.data.frame `as.data.frame.dbframe`: Coerce a ‘`dbframe`’ object to a data frame.

```
<<Define "as.data.frame" method>>=
    as.data.frame.dbframe <- function(x,...) select(x,...)
```

dim `dim.dbframe`: Determine the number of rows and columns in a dbframe.

```
<<Define "dim" method>>=
    dim.dbframe <- function(x) {
      nrows <- select(x, "count(*)")[[1]]
      ncols <- length(select(x, limit = 0))
      c(nrows, ncols)
    }
```

Implementation

The basic definition of these classes is straightforward. Individual methods are defined above; I'll define the generic functions here as necessary; note that the generic function for '`dbConnect`' is defined in the DBI package.

```
<<*>>=
    setClass("dbframe", representation(table = "character",
                                       readonly = "logical",
                                       dbConnect.arguments = "list"))

    setClass("dbframe_sqlite", contains = "dbframe",
             representation(rowid = "integer", dbname = "character"))

    <<Define "dbConnect" method for "dbframe">>
    <<Define "dbConnect" method for "dbframe_sqlite">>

    <<Define "as.data.frame" method>>
    <<Define "dim" method>>

    setGeneric("tablename", function(x) standardGeneric("tablename"))
    <<Define "tablename" method>>
    setGeneric("dbname", function(x) standardGeneric("dbname"))
    <<Define "dbname" method>>
    setGeneric("readonly", function(x) standardGeneric("readonly"))
    <<Define "readonly" method>>
    setGeneric("is.linked", function(x,...) standardGeneric("is.linked"))
    <<Define "is.linked" method>>
    setGeneric("rowid", function(x,...) standardGeneric("rowid"))
    <<Define "rowid" method>>
    setGeneric("rowid<-", function(x,...,value) standardGeneric("rowid<-"))
    <<Define "rowid<-" method>>
```

Author(s)

Gray Calhoun <`gcalhoun@iastate.edu`>

See Also

`dbframe`, `clear`, `insert<-`, `select`

head	*Retrieve head or tail of a table*

Description

Retrieves the first or last few records from a ‘dbframe’. This function mimics the corresponding data frame methods.

Usage

```
## S3 method for class 'dbframe'
head(x, n = 6L, ...)

## S3 method for class 'dbframe'
tail(x, n = 6L, ...)
```

Arguments

x	A dbframe object
n	An integer. If positive, the number of records to retrieve. If negative, these functions will retrieve all but ‘n’ records.
...	Other arguments to pass to ‘select’

Value

Returns a data frame with the records.

Implementation

‘head’ and ‘tail’ are really basic functions; the implementation is pretty straightforward. The only complication is that the “...” arguments can be passed multiple times, so I pull them out and store them in a list.

```
<<*>>=
    head.dbframe <- function(x, n = 6L,...) {
      if (n >= 0) {
        <<Return the first |n| records>>
      } else {
        <<Return all but the last |n| records>>
      }
    }

    tail.dbframe <- function(x, n = 6L,...) {
      if (n >= 0) {
        <<Return the last |n| records>>
      } else {
        <<Return all but the first |n| records>>
      }
    }
```

We use select statements with “limit” and “offset” to get the records.

```
<<Return the first |n| records>>=
    return(select(x, limit = n, as.data.frame = TRUE,...))

<<Return all but the last |n| records>>=
    return(select(x, limit = n + nrow(x), as.data.frame = TRUE,...))

<<Return the last |n| records>>=
    return(select(x, limit = sprintf("%d,%d", nrow(x) - n, n),
                  as.data.frame = TRUE,...))

<<Return all but the first |n| records>>=
    return(select(x, limit = sprintf("%d,%d", -n, nrow(x) + n),
                  as.data.frame = TRUE,...))
```

Unit tests

```
<<test-head.R>>=
    library(testthat)
    filename <- tempfile(fileext = ".db")

    data(morley)
    test_that("head and tail return the right number of records", {
      dbf <- dbframe("tab1", dbname = filename, data = morley)
      expect_that(nrow(head(dbf)), equals(6))
      expect_that(nrow(tail(dbf)), equals(6))

      nrec <- sample(1:nrow(morley), 1)
      expect_that(nrow(head(dbf, nrec)), equals(nrec))
      expect_that(nrow(tail(dbf, nrec)), equals(nrec))

      expect_that(nrow(head(dbf, -nrec)), equals(nrow(morley) - nrec))
      expect_that(nrow(tail(dbf, -nrec)), equals(nrow(morley) - nrec))
    })
```

Author(s)

Gray Calhoun <gcalhoun@iastate.edu>

See Also

head, tail, select

Examples

```
data(chickwts)
filename <- tempfile(fileext = ".db")
chicksdb <- dbframe("head1", dbdriver = "SQLite", dbname = filename,
                    data = chickwts)
head(chickwts)
tail(chickwts, -60)
tail(chickwts)
unlink(filename)
```

`insert<-` *Insert a data frame into the* SQL *database*

Description

This function inserts data from a data frame into a table referenced by a dbframe object. As the example makes clear (I hope) this function is dead easy to use; the columns of '`value`' can be in any order, and, if the table referenced by '`x`' exits, only the columns that already defined in the table are inserted.

Usage

```
insert(x,...) <- value
```

Arguments

`x`	A dbframe object that links to a table in an SQL database.
`value`	A data frame containing the data to insert into the database.
`...`	Additional arguments to pass to '`dbWriteTable`'.

Value

Since this is an assignment function, it returns the dbframe '`x`'. The function is used for its side effect, which is to insert the contents of '`value`' into the table reference by '`x`'.

Implementation

I've implemented this function as a method of the dbframe class; presumably it could be extended to other classes. So, the main steps in the defining the function are to define a generic and then set the specific method for the dbframe class. The difference between the main method and the method for '`dbframe{\char95}sqlite{\char95}temporary`' objects is that the second method shouldn't open and close the database connection.

```
<<*>>=
    setGeneric("insert<-", function(x,..., value) standardGeneric("insert<-"))

    setMethod("insert<-", signature = "dbframe", function(x,...,value) {
      stopifnot(!readonly(x))
      dbc <- dbConnect(x)
      <<Determine whether the table exists>>
      <<Write to database and store rowid>>
      dbDisconnect(dbc)
      return(x)
    })
```

We can use the '`dbExistsTable`' method (from DBI) to determine whether the table already exists in the database. If it does, we find out which columns in '`value`' already exists, because we will only insert those columns into the database. If the table does not exist, we insert all of the columns of '`value`'.

```
<<Determine whether the table exists>>=
    cols <-
      if (dbExistsTable(dbc, tablename(x))) {
        colnames <- names(select(x, limit = 0))
        colnames[colnames %in% names(value)]
      } else {
        names(value)
      }
```

Writing to the database is straightforward—we just use the ‘`dbWriteTable`’ method (also from DBI). The columns of the data frame ‘`value`’ are rearranged to agree with the table in the database automatically by indexing ‘`value`’ with the vector ‘`cols`’ determined above.

```
<<Write to database and store rowid>>=
    dbWriteTable(dbc, tablename(x), value[, cols, drop=FALSE],
                 row.names = FALSE, overwrite = FALSE, append = TRUE,...)
    rowid(x) <- unname(unlist(dbGetQuery(dbc, "select last_insert_rowid();")))
```

See Also

The DBI package and documentation, `dbWriteTable`, `dbframe-class`

Examples

```
data(chickwts)
filename <- tempfile(fileext = ".db")
chicksdb <- dbframe("insert1", dbdriver = "SQLite",
                    dbname = filename, clear = TRUE)
## Add some records
insert(chicksdb) <- chickwts[1:2,]
select(chicksdb)
## Add some more
insert(chicksdb) <- tail(chickwts)
```

`RepParallel` *Parallel version of* `replicate`

Description

A simple parallel clone of ‘`replicate`’; this function is a wrapper for ‘`mclapply`’ just like ‘`replicate`’ is a wrapper for ‘`sapply`’.

Usage

```
RepParallel(n, expr, simplify = "array",...)
```

Arguments

`n`	An integer giving the number of replications to execute.
`expr`	R code to execute.
`simplify`	logical or character string; should the result be simplified to a vector, matrix or higher dimensional array if possible?
`...`	Optional arguments to ‘`mclapply`’.

Value

Just as in replicate.

Implementation

```
<<*>>=
    RepParallel <- function(n, expr, simplify = "array",...) {
      answer <-
        mclapply(integer(n), eval.parent(substitute(function(...) expr)),...)
      if (!identical(simplify, FALSE) && length(answer))
        return(simplify2array(answer, higher = (simplify == "array")))
      else return(answer)
    }
```

Author(s)

Gray Calhoun <gcalhoun@iastate.edu>

See Also

replicate, mclapply

`rows` *Extract rows from a data frame and present as a list*

Description

This is a convenience function that takes a dataframe “x” and returns a list where each element is a row of x. I use this for loops.

Usage

```
rows(x)
```

Arguments

x	A data frame or an object that can be coerced to a data frame.

Value

A list; each element is a row of x.

Implementation

```
<<*>>=
    rows <- function(x) {
      x <- as.data.frame(x)
      if (nrow(x) > 0) {
        lapply(seq.int(nrow(x)), function(i) x[i,])
      } else {
        list()
      }
    }
```

Author(s)

Gray Calhoun <gcalhoun@iastate.edu>

Examples

```
data(chickwts)
for (r in rows(head(chickwts))) print(r)
```

---

select *Retrieve records from a dbframe*

Description

'select' is a wrapper for the SQL select query and is used to retrieve records from a 'dbframe' object. 'generate.select.sql' assembles a valid SQL select statement from its arguments. The arguments map to clauses in the SQL select statement, so you may need to consult an introduction to SQL to best use these functions.

Usage

```
select(x, cols, as.data.frame = TRUE,...)

generate.select.sql(table, cols = "*", where = NULL, group.by = NULL,
                    having = NULL, order.by = NULL, limit = NULL,...)
```

Arguments

x	A 'dbframe' object that references the table of interest.
table	A character object containing the name of the SQL table to query.
cols	A character vector containing the column names (or functions of the column names) to retrieve from the database.
where	A character object that contains conditions to use to filter the records.
group.by	A character vector that defines groups of records to combine with an aggregate function.
having	A character object that filters the groups defined by "group.by".
order.by	A character vector that lists the columns to be used for sorting the results.
limit	A character vector or number that limits and offsets the SQL query results.
...	Additional arguments to pass to 'dbGetQuery'.
as.data.frame	Logical; if 'TRUE', execute the query and return the results as a data frame. If 'FALSE', return a dbframe that has the SQL statement for the query as its "table"

Source code

The inidividual methods do some minor parsing, but most of the work is done by ‘generate.select.sql’.

```
<<*>>=
    generate.select.sql <- function(table, cols = "*", where = NULL,
      group.by = NULL, having = NULL, order.by = NULL, limit = NULL, ...) {

      <<Format the "select" part of the statement>>
      <<Format the "group by" part of the statement>>
      <<Format the "order by" part of the statement>>
      <<Format the "having" part of the statement>>
      <<Format the "where" part of the statement>>
      <<Format the "limit" part of the statement>>
      return(paste("select", cols, "from", table, where,
                   group.by, having, order.by, limit))
    }

    setGeneric("select", function(x, cols, as.data.frame = TRUE,...)
               standardGeneric("select"))

    setMethod("select", signature = c("ANY", "missing"),
              function(x, cols, as.data.frame = TRUE,...) {
                <<Execute select for c("ANY", "missing")>>})

    setMethod("select", signature = c("dbframe", "character"),
              function(x, cols, as.data.frame = TRUE,...) {
                <<Execute select for c("dbframe", "character")>>})

    setMethod("select", signature = c("data.frame", "character"),
              function(x, cols, as.data.frame = TRUE,...) {
                <<Handle arguments and set up the local environment>>
                <<Export the new data frame to a temporary SQLite database>>
                <<Query the new database and close database connection>>
                return(queryresults)
              })

    setMethod("select", signature = c("list", "character"),
              function(x, cols,...) {
                <<Detect inappropriate uses of the "list" method>>
                <<Define and attach to "main" db>>
                <<Extract the arguments that describe the join>>
                <<Execute the query and return its results>>
              })

    ## setMethod("select", signature = c("dbframe", "list"),
    ##           function(x, cols,...) {
    ##             <Handle lists of a single query element>>
    ##             <Manage arguments for compound queries>>
   ##           <Construct individual SQL select statements for compound queries>>
    ##             <Execute query and return data>>
    ##           })
    <<Define additional useful functions>>
```

Unfortunately, this method doesn't yet allow for joins or compound queries involving data.frames. Maybe a better approach would be to always let the "main" database be a temporary one on disk.

Assembling the SQL select statement: To write the "select" part, we add the 'group.by' variables and the 'cols' variables together (and store them in 'cols').

```
<<Format the "select" part of the statement>>=
    <<Add new group.by variables in front of cols variables>>
    labels <- names(cols)
    labels[nchar(labels) > 0] <- paste("AS", labels[nchar(labels) > 0])
    cols <- paste(cols, labels, collapse = ", ")
```

To save typing, I assume that we want to retrieve the grouping variables and so we don't have to specify them explicitly in the 'cols' vector. I think it makes sense to have the grouping variables on the left side of the results set instead of the right side. The next code chunk does both of those.

```
<<Add new group.by variables in front of cols variables>>=
    cols <-
      if (is.null(cols)) {
        group.by
      } else if (is.null(group.by)) {
        cols
      } else {
        if (is.null(names(cols)))
          names(cols) <- rep("", length(cols))
        if (is.null(names(group.by)))
          names(group.by) <- rep("", length(group.by))
        c(group.by[!(names(group.by) %in% names(cols))
                   |  nchar(names(group.by)) == 0], cols)
      }
```

Managing the other arguments is easy. If they're 'NULL' we replace the variable with an empty string; if they're not, we add the appropriate label and replace the variable with a character object that contains a clause for the SQL statement.

```
<<Format the "group by" part of the statement>>=
    group.by <-
      if (is.null(group.by)) {
        ""
      } else {
        paste("group by", paste(group.by, collapse = ", "))
      }

<<Format the "order by" part of the statement>>=
    order.by <-
      if (is.null(order.by)) {
        ""
      } else {
        paste("order by", paste(order.by, collapse = ", "))
      }

<<Format the "having" part of the statement>>=
    having <-
```

```
        if (is.null(having)) {
          ""
        } else {
          paste("having", having)
        }

<<Format the "where" part of the statement>>=
    where <-
      if (is.null(where)) {
        ""
      } else {
        paste("where", where)
      }

<<Format the "limit" part of the statement>>=
    limit <-
      if (is.null(limit)) {
        ""
      } else {
        paste("limit", limit)
      }
```

Details of argument handling for simple queries: The individual methods just call ‘generate.select.sql’ and execute the select statement. If ‘cols’ is "missing" it returns results for ‘cols’ equal to "*" (i.e. all of the columns of the table).

```
<<Execute select for c("ANY", "missing")>>=
    select(x, "*", as.data.frame,...)

<<Execute select for c("dbframe", "character")>>=
    if (!is.linked(x)) {
      warning("Table does not exist in the data base")
      return(list())
    }
    arguments <- list(table = tablename(x), cols = cols,...)
    sql.statement <- do.call("generate.select.sql", arguments)
    if (as.data.frame) {
      dbc <- dbConnect(x)
      d <- do.call("dbGetQuery", c(conn = dbc, statement = sql.statement,
                                   arguments))
      dbDisconnect(dbc)
    } else {
      if (is.null(arguments$readonly)) {
        readonly <- readonly(x)
      } else {
        readonly <- arguments$readonly
        arguments$readonly <- NULL
      }
      d <- do.call("new", c(Class = "dbframe", table = sql.statement,
                   readonly = readonly, dbConnect.arguments = arguments))
    }
    return(d)

<<Handle arguments and set up the local environment>>=
    if (!as.data.frame)
```

```
     warning("'as.data.frame' ignored when selecte is called on a data.frame.")
    tablename <- "dataframe"
    require(RSQLite)
    require(RSQLite.extfuns)
```

```
<<Export the new data frame to a temporary SQLite database>>=
    dbc <- dbConnect("SQLite", dbname = ":memory:")
    dbWriteTable(dbc, tablename, x, row.names = FALSE)
```

```
<<Query the new database and close database connection>>=
    sql.statement <- generate.select.sql(tablename, cols,...)
    queryresults <- dbGetQuery(dbc, sql.statement)
    dbDisconnect(dbc)
```

Details of argument handling for joins: So far, I'm only supporting joins for dbframes that are linked to SQLite data bases and for data.frames.

```
<<Detect inappropriate uses of the "list" method>>=
    if (length(x) == 1) return(select(x[[1]], cols,...))
    if (is.null(names(x))) names(x) <- LETTERS[seq_along(x)]
    tableclasses <- sapply(x, class)
    if (!all(tableclasses %in% c("dbframe_sqlite", "data.frame")))
      stop("Some of your dbframes aren't supported yet")
    if (any(tableclasses == "data.frame")) {
      require(RSQLite)
      require(RSQLite.extfuns)
    }
```

One nice feature of this function is that it handles all of the "attach" commands that are necessary to merge tables that exist in different databases. If all of the dbframes link to the same database, then that one will obviously be the main database; otherwise we connect to a temporary SQLite database and attach everything there.

```
<<Define and attach to "main" db>>=
    dbnames <- tablenames <- rep(NA, length(x))
    for (s in seq_along(x)) {
      if (tableclasses[s] == "dbframe_sqlite") {
        dbnames[s] <- dbname(x[[s]])
        tablenames[s] <- tablename(x[[s]])
      } else {
        dbnames[s] <- "temp"
        tablenames[s] <- names(x)[[s]]
      }
    }
    not.data.frames <- which(tableclasses != "data.frame")
    dbalias <- dbnames
    if (isTRUE(sum(!duplicated(dbnames[not.data.frames])) == 1)) {
      maindbc <- dbConnect(x[[not.data.frames[1]]])
      dbnames[not.data.frames] <- "main"
      dbalias[not.data.frames] <- "main"
    } else {
      maindbc <- dbConnect("SQLite", dbname = ":memory:")
      <<Attach sqlite_dbframes to the main db>>
    }
    <<Write dataframes to the main db>>
```

Any data frames are just going to be written to the temporary database.

```
<<Write dataframes to the main db>>=
    sapply(which(tableclasses == "data.frame"), function(s)
           dbWriteTable(maindbc, paste("temp", tablenames[s], sep = "."),
                                                   x[[s]], row.names = FALSE))
```

Tables that already exist in other databases are attached to the temporary database.

```
<<Attach sqlite_dbframes to the main db>>=
    dbcount <- 0
    unique.databases <- unique(dbnames[!(dbnames %in% c("temp", "main"))])
    for (db in unique.databases) {
      dbcount <- dbcount + 1
      currentalias <- sprintf("ALIAS%d", dbcount)
      dbalias[dbalias == db] <- currentalias
     r <- dbSendQuery(maindbc, sprintf("attach database '%s' as %s", db, currentalias))
      dbClearResult(r)
    }
```

The columns are already specified for the query; the only thing to do is assemble the SQL code for the "table" part. If the join type is not specified, the default is to do an inner join; there is no default for "on" or "using", so one (and only one) of those arguments must be specified.

```
<<Extract the arguments that describe the join>>=
    arguments <- list(...)
    join  <- extract.element("join", "inner", length(x) - 1, arguments)
    on    <- extract.element("on", NA, length(x) - 1, arguments)
    using <- extract.element("using", NA, length(x) - 1, arguments)
    if (any(is.na(on) & is.na(using)))
      stop("'on' and 'using' can't both be specified for the same join.")
    arguments$join  <- NULL
    arguments$on    <- NULL
    arguments$using <- NULL
    arguments$cols  <- cols
    arguments$table <- paste(collapse = " ", c(
      sprintf("%s.%s %s", dbalias[1], tablenames[1], names(x)[1]),
      sprintf("%s join %s.%s %s %s", join, dbalias[-1], tablenames[-1],
                                                          names(x)[-1],
              ifelse(is.na(on), ifelse(is.na(using), "",
                        sprintf("using(%s)", using)), sprintf("on %s", on)))))
```

Actually executing the query is the same as for the other methods. We're not going to worry about detaching the tables explicitly, since we know that they were only attached if we're using a temporary data base.

```
<<Execute the query and return its results>>=
    results <- dbGetQuery(maindbc, do.call(generate.select.sql, arguments))
    dbDisconnect(maindbc)
    return(results)
```

Miscellaneous function:

```
<<Define additional useful functions>>=
   extract.element <- function(name, default, length.required, argument.list) {
     v <- if (name %in% names(argument.list)) argument.list[[name]] else default
      if (is.na(length.required) | length(v) == length.required) return(v)
```

```
          else if (length(v) == 1) return(rep(v, length.required))
          else stop("Incorrect length of argument")
        }
```

Unit Tests

I just have some basic sanity-check type unit tests; i.e. do the functions run at a minimal level.

```
<<test-select.R>>=
    library(testthat)
    data(chickwts)
    chickwts$feed <- as.character(chickwts$feed)
    test_that("insert and select work", {
      <<Individual tests that insert and select work>>})
    test_that("column renaming scheme works", {
      <<Individual tests that column renaming works>>})
    test_that("joins work", {
      <<Individual tests that joins work>>})
```

First we'll check that the methods defined on data frames work, then that they work for dbframes.

```
<<Individual tests that insert and select work>>=
    expect_that(chickwts, is_equivalent_to(select(chickwts)))
    <<Create temporary test database and dbframe>>
    insert(testdbframe) <- chickwts
    expect_that(chickwts, is_equivalent_to(select(testdbframe)))
    <<Remove temporary test database>>

<<Individual tests that column renaming works>>=
    expect_that(
      c("feed", "AverageWeight"),
      is_identical_to(names(select(chickwts,
           c(AverageWeight = "avg(weight)"), group.by = "feed"))))
    <<Create temporary test database and dbframe>>
    insert(testdbframe) <- chickwts
    expect_that(
      c("feed", "AverageWeight"),
      is_identical_to(names(select(testdbframe,
           c(AverageWeight = "avg(weight)"), group.by = "feed"))))
    <<Remove temporary test database>>

<<Individual tests that joins work>>=
    <<Create temporary test database and dbframe>>
    expect_that(select(list(A = chickwts,B =  chickwts),
                    c("feed", weightA = "A.weight", weightB = "B.weight"),
                   using = "feed", order.by = c("feed", "weightA", "weightB")),
                equals({
                  d <- merge(chickwts, chickwts, by = "feed",
                                                   suffixes = c("A", "B"))
                  d$feed <- as.character(d$feed)
                  d[do.call(order, d),]
                }, check.attributes = FALSE))
    avgwts <- dbframe("select2", dbname = testdbfile, clear = TRUE,
                      data = select(chickwts, c(averageweight = "avg(weight)"),
                                                 group.by = c(thefeed = "feed")))
    expect_that(select(list(a = chickwts, b = avgwts),
                       c("feed", "weight", "averageweight"),
                       on = ("feed = thefeed"), order.by = "feed, weight"),
                equals({
                  d <- merge(chickwts, select(avgwts), by.x = "feed",
                                                       by.y = "thefeed")
                  d$feed <- as.character(d$feed)
                  d[do.call(order, d),]
                }, check.attributes = FALSE))
    <<Remove temporary test database>>

<<Create temporary test database and dbframe>>=
    testdbfile <- tempfile(fileext = ".db")
    testdbframe <- dbframe("select1", testdbfile)
    clear(testdbframe)

<<Remove temporary test database>>=
    unlink(testdbfile)
```

References

SQL as understood by SQLite. http://www.sqlite.org/lang_select.html

Examples

```
filename <- tempfile(fileext = ".db")
data(chickwts)
chicksdb <- dbframe("select1", dbname = filename,
                    clear = TRUE, data = chickwts)
select(chicksdb, where = "weight > 200", order.by = "weight")
select(chicksdb, c(averageweight = "avg(weight)"), group.by = "feed")
select(chicksdb, c(averageweight = "avg(weight)"), group.by = "feed",
       having = "averageweight > 250")

## and an example of querying the data frame directly
select(chickwts, c(averageweight = "avg(weight)"),
       group.by = c(thefeed = "feed"))
avgwts <- dbframe("select2", dbname = filename, clear = TRUE,
                  data = select(chickwts, c(averageweight = "avg(weight)"),
                                group.by = c(thefeed = "feed")))
## an example of a join
select(list(a = chicksdb, b = avgwts), c("feed", "weight", "averageweight"),
       on = ("feed = thefeed"), order.by = "feed, weight")
```

Index