

Package ‘sqldf’

February 14, 2010

Version 0.3-4

Date 2010-02-13

Title Perform SQL Selects on R Data Frames

Author G. Grothendieck <ggrothendieck@gmail.com>

Maintainer G. Grothendieck <ggrothendieck@gmail.com>

Description Manipulate R data frames using SQL.

Depends R (>= 2.10.0), DBI (>= 0.2-5), RSQLite (>= 0.8-0), gsubfn, proto, chron

Suggests RH2

License GPL-2

URL <http://sqldf.googlecode.com>

Repository CRAN

Date/Publication 2010-02-14 14:44:31

R topics documented:

sqldf-package	2
read.csv.sql	2
sqldf	3

Index	12
--------------	-----------

sqldf-package *sqldf package overview*

Description

Provides an easy way to perform SQL selects on R data frames.

Details

The package contains a single function `sqldf` whose help file contains more information and examples.

References

The `sqldf` help page contains the primary documentation. The `sqldf` home page <http://sqldf.googlecode.com> contains links to SQLite pages that may be helpful in formulating queries.

read.csv.sql *Read File Filtered by SQL*

Description

Read a file into R filtering it with an sql statement. Only the filtered portion is processed by R so that files larger than R can otherwise handle can be accommodated.

Usage

```
read.csv.sql(file, sql = "select * from file", header = TRUE, sep = ",", row.names,
read.csv2.sql(file, sql = "select * from file", header = TRUE, sep = ";", row.names
```

Arguments

<code>file</code>	As in <code>read.csv</code> .
<code>sql</code>	character string holding an SQL statement. The table representing the file should be referred to as <code>file</code> .
<code>header</code>	As in <code>read.csv</code> .
<code>sep</code>	As in <code>read.csv</code> .
<code>row.names</code>	As in <code>read.csv</code> .
<code>eol</code>	Character which ends line.
<code>skip</code>	Skip indicated number of lines in input file.
<code>filter</code>	If specified, this should be a shell/batch command that the input file is piped through. For <code>read.csv2.sql</code> it is by default the following on non-Windows systems: <code>tr , ..</code> . This translates all commas in the file to dots. On Windows similar functionality is provided but to do that using a <code>vbscript</code> file that is included with <code>sqldf</code> to emulate the <code>tr</code> command.

dbname	As in <code>sqldf</code> except that the default is <code>tempfile()</code> .
drv	This argument is ignored. Currently the only database SQLite supported by <code>read.csv.sql</code> and <code>read.csv2.sql</code> is SQLite. Note that the H2 database has a builtin SQL function, <code>CSVREAD</code> , which can be used in place of <code>read.csv.sql</code> .
...	Passed to <code>sqldf</code> .

Details

Reads the indicated file into an sql database creating the database if it does not already exist. Then it applies the sql statement returning the result as a data frame. If the database did not exist prior to this statement it is removed.

Note that it uses facilities of SQLite to read the file which are intended for speed and therefore not as flexible as in R. For example, it does not recognize quoted fields as special but will regard the quotes as part of the field. See the `sqldf` help for more information.

`read.csv2.sql` is like `read.csv.sql` except the default `sep` is ";" and the default `filter` translates all commas in the file to decimal points (i.e. to dots).

On Windows, if the `filter` argument is used and if Rtools is detected in the registry then the Rtools bin directory is added to the search path facilitating use of those tools without explicitly setting any the path.

Value

If the sql statement is a select statement then a data frame is returned.

Examples

```
## Not run:
write.table(iris, "iris.csv", sep = ",", quote = FALSE, row.names = FALSE)
iris2 <- read.csv.sql("iris.csv",
sql = "select * from file where Sepal.Length > 5")

## End(Not run)
```

sqldf

SQL select on data frames

Description

SQL select on data frames

Usage

```
sqldf(x, stringsAsFactors = TRUE, col.classes = NULL,
      row.names = FALSE, envir = parent.frame(), method = c("auto", "raw"),
      file.format = list(), dbname, drv = getOption("sqldf.driver"),
      user, password = "", host = "localhost",
      dll = getOption("sqldf.dll"), connection = getOption("sqldf.connection"))
```

Arguments

<code>x</code>	Character string representing an SQL select statement or character vector whose components each represent a successive SQL statement to be executed. The select statement syntax must conform to the particular database being used. If <code>x</code> is missing then it establishes a connection which subsequent sqldf statements access. In that case the database is not destroyed until the next sqldf statement with no <code>x</code> .
<code>stringsAsFactors</code>	If TRUE then output "character" columns are converted to "factor" if the heuristic is unable to determine the class. If <code>method="raw"</code> then <code>stringsAsFactors</code> is ignored.
<code>col.classes</code>	Not currently used.
<code>row.names</code>	For TRUE the tables in the data base are given a <code>row_names</code> column filled with the row names of the corresponding data frames. Note that in SQLite a special <code>rowid</code> (or equivalently <code>oid</code> or <code>_rowid_</code>) is available in any case.
<code>envir</code>	The environment where the data frames representing the tables are to be found.
<code>method</code>	"auto" means automatically assign the class of each column using the heuristic described later. "raw" means use whatever classes are returned by the database with no automatic processing.
<code>file.format</code>	A list whose components are passed to <code>sqliteImportFile</code> . Components may include <code>sep</code> , <code>header</code> , <code>row.names</code> , <code>skip</code> , <code>eol</code> and <code>filter</code> . Except for <code>filter</code> they are passed to <code>sqliteImportFile</code> and have the same default values as in <code>sqliteImportFile</code> (except for <code>eol</code> which defaults to the end of line character(s) for the operating system in use). <code>filter</code> may optionally contain a batch/shell command through which the input file is piped prior to reading it in. <code>file.format</code> may be set to NULL in order not to search for input file objects at all. The <code>file.format</code> can also be specified as an attribute in each file object itself in which case such specification overrides any given through the argument list. There is further discussion of <code>file.format</code> below.
<code>dbname</code>	Name of the database. For SQLite data bases this defaults to <code>":memory:"</code> which results in an embedded database.
<code>drv</code>	"SQLite" or "MySQL". If not specified then the "dbDriver" option is checked and if that is not set then "SQLite" is used unless the RMySQL package is loaded.
<code>user</code>	user name. Not needed for embedded databases.
<code>password</code>	password. Not needed for embedded databases.
<code>host</code>	host. Default of "localhost" is normally sufficient.
<code>dll</code>	Name of an SQLite loadable extension to automatically load. Default is <code>"libspatial-1.dll"</code> . If found on PATH then it is automatically loaded and the SQLite functions it in will be accessible.
<code>connection</code>	If this is NULL then a connection is created; otherwise the indicated connection is used. The default is the value of the option <code>sqldf.connection</code> . If neither <code>connection</code> nor <code>sqldf.connection</code> are specified a connection is automatically generated on-the-fly and closed on exit of the call to <code>sqldf</code> . If this

argument is not `NULL` then the specified connection is left open on termination of the `sqldf` call. Usually this argument is left unspecified. It can be used to make repeated calls to a database without reloading it.

Details

The typical action of `sqldf` is to

create a database in memory

read in the data frames and files used in the select statement. This is done by scanning the select statement to see which words in the select statement are of class "data.frame" or "file" in the parent frame, or the specified environment if `envir` is used, and for each object found by reading it into the database if it is a data frame. Note that this heuristic usually reads in the wanted data frames and files but on occasion may harmlessly reads in extra ones too.

run the select statement getting the result as a data frame

assign the classes of the returned data frame's columns if `method = "auto"`. This is done by checking all the column names in the read-in data frames and if any are the same as in the output data frame their class (and their factor levels if factor) is used. If they are not matched then they are returned as is except that if `stringsAsFactors = TRUE` then any character strings are converted to factors. If `method = "raw"` then the classes are returned as is from the database and `stringsAsFactors` is ignored.

cleanup If the database was created by `sqldf` then it is deleted; otherwise, all tables that were created are dropped in order to leave the database in the same state that it was before. The database connection is terminated.

Warning. Although `sqldf` is usually used with on-the-fly databases which it automatically sets up and destroys if you wish to use it with existing databases be sure to back up your database prior to using it since incorrect operation could destroy the entire database.

Value

The result of the specified select statement is output as a data frame. If a vector of sql statements is given as `x` then the result of the last one is returned. If the `x` and `connection` arguments are missing then it returns a new connection and also places this connection in the option `sqldf.connection`.

Note

If `row.names = TRUE` is used then any `NATURAL JOIN` will make use of it which may not be what was intended.

`3/2` and `3.0/2` are the same in R but in SQLite the first one causes integer arithmetic to be used whereas the second using floating point. Thus both evaluate to 1.5 in R but they evaluate to 1 and 1.5 respectively in SQLite.

The `dbWriteTable/sqliteImportFile` routines that `sqldf` uses to transfer files to the data base are intended for speed and they are not as flexible as `read.table`. Also they have slightly different defaults. (If more flexible input is needed use the slower `read.table` to read the data into a data frame instead of reading directly from a file.) The default for `sep` is `sep = ", "`. If the first row of the file has one fewer entry than subsequent ones then it is assumed that `header`

`<- row.names <- TRUE` and otherwise that `header <- row.names <- FALSE`. The header can be forced to `header <- TRUE` by specifying `file.format = list(header = TRUE)` as an argument to `sqldf`. `sep` and `row.names` are other `file.format` subarguments. Also, one limitation with `.csv` files is that quotes are not regarded as special within files so a comma within a data field such as "Smith, James" would be regarded as a field delimiter and the quotes would be entered as part of the data which probably is not what is intended.

Typically the SQL result will have the same data as the analogous non-database R code manipulations using data frames but may differ in row names and other attributes. In the examples below we use `identical` in those cases where the two results are the same in all respects or set the row names to `NULL` if they would have otherwise differed only in row names or use `all.equal` if the data portion is the same but attributes aside from row names differ.

The SQLite code has been tested but the MySQL code has only been partly tested.

On MySQL the database must pre-exist. Create a `c:\my.cnf` file on Windows or a `/etc/my.cnf` file on UNIX to contain information about the database. This file may include the username, password, database and port. The password can be omitted if one has not been set and the database can be omitted if its passed as the `dbname` argument to `sqldf`. The `port` argument can usually be omitted as well. See <http://dev.mysql.com/doc/refman/5.0/en/option-files.html>.

In versions of the DBI package prior to DBI 0.2-5, SQL reserved words such as `time` and `date` were automatically translated to `time__1` and `date__1`, etc. to prevent collisions. The new version of DBI used with the current version of `sqldf` automatically quotes those variables instead so that the database will use the column names of `date` and `codetime` instead of `date__1` and `time__1`. The user moving from older versions of `sqldf` to this one should be aware of this change in DBI.

If `getOption("sqldf.dll")`, defaulting to `"libspatial.dll"` is found on the `PATH` then it will be loaded as an SQLite loadable extension. With the default the user will also have access to these additional SQL functions: <http://www.gaia-gis.it/spatialite/spatialite-sql-2.3.1.html> The `sqldf` package does not itself include this or any other `dll` and the user must download and place it on the `PATH` if they wish to use it. The `dll` can be found here: <http://www.gaia-gis.it/spatialite> If `sqldf` does not find the `dll` on the `PATH` then `sqldf` will continue to work but, of course, those functions will not be available. If you add `libspatialite-1.dll` to your `PATH` part way through your session then it will be necessary to set the `sqldf.dll` option to `NULL`: `options(sqldf.dll = NULL)` as well.

References

The `sqldf` home page <http://code.google.com/p/sqldf/> contains more examples as well as links to SQLite pages that may be helpful in formulating queries.

Examples

```
#
# These examples show how to run a variety of data frame manipulations
# in R without SQL and then again with SQL
#
# head
```

```
a1r <- head(warpbreaks)
a1s <- sqldf("select * from warpbreaks limit 6")
identical(a1r, a1s)

# subset

a2r <- subset(CO2, grepl("^Qn", Plant))
a2s <- sqldf("select * from CO2 where Plant like 'Qn%'")
all.equal(a2r, a2s, check.attributes = FALSE)

data(farms, package = "MASS")
a3r <- subset(farms, Manag %in% c("BF", "HF"))
a3s <- sqldf("select * from farms where Manag in ('BF', 'HF')")
row.names(a3r) <- NULL
identical(a3r, a3s)

a4r <- subset(warpbreaks, breaks >= 20 & breaks <= 30)
a4s <- sqldf("select * from warpbreaks where breaks between 20 and 30",
  row.names = TRUE)
identical(a4r, a4s)

a5r <- subset(farms, Mois == 'M1')
a5s <- sqldf("select * from farms where Mois = 'M1'", row.names = TRUE)
identical(a5r, a5s)

a6r <- subset(farms, Mois == 'M2')
a6s <- sqldf("select * from farms where Mois = 'M2'", row.names = TRUE)
identical(a6r, a6s)

# rbind
a7r <- rbind(a5r, a6r)
a7s <- sqldf("select * from a5s union all select * from a6s", row.names = TRUE)
identical(a7r, a7s)

# aggregate - avg conc and uptake by Plant and Type
a8r <- aggregate(iris[1:2], iris[5], mean)
a8s <- sqldf("select Species, avg(Sepal_Length) `Sepal.Length`,
  avg(Sepal_Width) `Sepal.Width` from iris group by Species")
all.equal(a8r, a8s)

# by - avg conc and total uptake by Plant and Type
a9r <- do.call(rbind, by(iris, iris[5], function(x) with(x,
  data.frame(Species = Species[1],
  mean.Sepal.Length = mean(Sepal.Length),
  mean.Sepal.Width = mean(Sepal.Width),
  mean.Sepal.ratio = mean(Sepal.Length/Sepal.Width))))))
row.names(a9r) <- NULL
a9s <- sqldf("select Species, avg(Sepal_Length) `mean.Sepal.Length`,
  avg(Sepal_Width) `mean.Sepal.Width`,
  avg(Sepal_Length/Sepal_Width) `mean.Sepal.ratio` from iris
  group by Species")
all.equal(a9r, a9s)
```

```

# head - top 3 breaks
a10r <- head(warpbreaks[order(warpbreaks$breaks, decreasing = TRUE), ], 3)
a10s <- sqldf("select * from warpbreaks order by breaks desc limit 3")
row.names(a10r) <- NULL
identical(a10r, a10s)

# head - bottom 3 breaks
a11r <- head(warpbreaks[order(warpbreaks$breaks), ], 3)
a11s <- sqldf("select * from warpbreaks order by breaks limit 3")
# attributes(a11r) <- attributes(a11s) <- NULL
row.names(a11r) <- NULL
identical(a11r, a11s)

# ave - rows for which v exceeds its group average where g is group
DF <- data.frame(g = rep(1:2, each = 5), t = rep(1:5, 2), v = 1:10)
a12r <- subset(DF, v > ave(v, g, FUN = mean))
Gavg <- sqldf("select g, avg(v) as avg_v from DF group by g")
a12s <- sqldf("select DF.g, t, v from DF, Gavg where DF.g = Gavg.g and v > avg_v")
row.names(a12r) <- NULL
identical(a12r, a12s)

# same but reduce the two select statements to one using a subquery
a13s <- sqldf("select g, t, v from DF d1, (select g as g2, avg(v) as avg_v from DF group by
identical(a12r, a13s)

# same but shorten using natural join
a14s <- sqldf("select g, t, v from DF natural join (select g, avg(v) as avg_v from DF group
identical(a12r, a14s)

# table
a15r <- table(warpbreaks$tension, warpbreaks$wool)
a15s <- sqldf("select sum(wool = 'A'), sum(wool = 'B')
  from warpbreaks group by tension")
all.equal(as.data.frame.matrix(a15r), a15s, check.attributes = FALSE)

# reshape
t.names <- paste("t", unique(as.character(DF$t)), sep = "_")
a16r <- reshape(DF, direction = "wide", timevar = "t", idvar = "g", varying = list(t.names))
a16s <- sqldf("select g, sum((t == 1) * v) t_1, sum((t == 2) * v) t_2, sum((t == 3) * v) t_3")
all.equal(a16r, a16s, check.attributes = FALSE)

# order
a17r <- Formaldehyde[order(Formaldehyde$optden, decreasing = TRUE), ]
a17s <- sqldf("select * from Formaldehyde order by optden desc")
row.names(a17r) <- NULL
identical(a17r, a17s)

# centered moving average of length 7
set.seed(1)
DF <- data.frame(x = rnorm(15, 1:15))
s18 <- sqldf("select a.x x, avg(b.x) movavgx from DF a, DF b
  where a.row_names - b.row_names between -3 and 3
  group by a.row_names having count(*) = 7")

```

```

    order by a.row_names+0",
    row.names = TRUE)
r18 <- data.frame(x = DF[4:12,], movavgx = rowMeans(embed(DF$x, 7)))
row.names(r18) <- NULL
all.equal(r18, s18)

# merge. a19r and a19s are same except row order and row names
A <- data.frame(a1 = c(1, 2, 1), a2 = c(2, 3, 3), a3 = c(3, 1, 2))
B <- data.frame(b1 = 1:2, b2 = 2:1)
a19s <- sqldf("select * from A, B")
a19r <- merge(A, B)
Sort <- function(DF) DF[do.call(order, DF),]
all.equal(Sort(a19s), Sort(a19r), check.attributes = FALSE)

# within Date, of the highest quality records list the one closest
# to noon. Note use of two sql statements in one call to sqldf.

Lines <- "DeployID Date.Time LocationQuality Latitude Longitude
STM05-1 2005/02/28 17:35 Good -35.562 177.158
STM05-1 2005/02/28 19:44 Good -35.487 177.129
STM05-1 2005/02/28 23:01 Unknown -35.399 177.064
STM05-1 2005/03/01 07:28 Unknown -34.978 177.268
STM05-1 2005/03/01 18:06 Poor -34.799 177.027
STM05-1 2005/03/01 18:47 Poor -34.85 177.059
STM05-2 2005/02/28 12:49 Good -35.928 177.328
STM05-2 2005/02/28 21:23 Poor -35.926 177.314
"

DF <- read.table(textConnection(Lines), skip = 1, as.is = TRUE,
  col.names = c("Id", "Date", "Time", "Quality", "Lat", "Long"))

sqldf(c("create temp table DFo as select * from DF order by
  Date DESC, Quality DESC,
  abs(substr(Time, 1, 2) + substr(Time, 4, 2) /60 - 12) DESC",
  "select * from DFo group by Date"))

## Not run:

# test of file connections with sqldf

# create test .csv file of just 3 records
write.table(head(iris, 3), "iris3.dat", sep = ",", quote = FALSE)

# look at contents of iris3.dat
readLines("iris3.dat")

# set up file connection
iris3 <- file("iris3.dat")
sqldf("select * from iris3 where Sepal_Width > 3")

# using a non-default separator
# file.format can be an attribute of file object or an arg passed to sqldf
write.table(head(iris, 3), "iris3.dat", sep = ";", quote = FALSE)

```

```

iris3 <- file("iris3.dat")
sqldf("select * from iris3 where Sepal_Width > 3", file.format = list(sep = ";"))

# same but pass file.format through attribute of file object
attr(iris3, "file.format") <- list(sep = ";")
sqldf("select * from iris3 where Sepal_Width > 3")

# copy file straight to disk without going through R
# and then retrieve portion into R
sqldf("select * from iris3 where Sepal_Width > 3", dbname = tempfile())

### same as previous example except it allows multiple queries against
### the database. We use iris3 from before. This time we use an
### in memory SQLite database.

sqldf() # open a connection
sqldf("select * from iris3 where Sepal_Width > 3")

# At this point we have an iris3 variable in both
# the R workspace and in the SQLite database so we need to
# explicitly let it know we want the version in the database.
# If we were not to do that it would try to use the R version
# by default and fail since sqldf would prevent it from
# overwriting the version already in the database to protect
# the user from inadvertent errors.
sqldf("select * from main.iris3 where Sepal_Width > 4")
sqldf("select * from main.iris3 where Sepal_Width < 4")
sqldf() # close connection

### another way to do this is a mix of sqldf and RSQLite statements
### In that case we need to fetch the connection for use with RSQLite
### and do not have to specifically refer to main since RSQLite can
### only access the database.

con <- sqldf()
# this iris3 refers to the R variable and file
sqldf("select * from iris3 where Sepal_Width > 3")
# these iris3 refer to the database table
dbGetQuery(con, "select from iris3 where Sepal_Width > 4")
dbGetQuery(con, "select from iris3 where Sepal_Width < 4")
sqldf()

### demonstrate use of libspatial.dll loadable extension functions
### In this example stddev_pop and var_pop are libspatial functions.
### This will only work if libspatial.dll is on the PATH.
\dontrun{
Sys.which("libspatialite-1.dll") # shows location of spatialite on PATH
sqldf("select avg(demand) mean, stddev_pop(demand) sd, var_pop(demand) var from BOD")
with(BOD, c(mean = mean(demand), sd = sd(demand), var = var(demand)))
}

```

```
## End(Not run)
```

Index

*Topic **manip**

read.csv.sql, 2

sqldf, 3

*Topic **package**

sqldf-package, 1

read.csv.sql, 2

read.csv2.sql (*read.csv.sql*), 2

read.table, 5

sqldf, 1, 2, 3

sqldf-package, 1