Package: SQRL (via r-universe)

October 29, 2024

Type Package

Title Enhances Interaction with 'ODBC' Databases

Version 1.0.2

Date 2023-12-03

Author Mike Lee

Maintainer Mike Lee <random.deviate@gmail.com>

Description Provides simple and powerful interfaces that facilitate interaction with 'ODBC' data sources. Each data source gets its own unique and dedicated interface, wrapped around 'RODBC'. Communication settings are remembered between queries, and are managed silently in the background. The interfaces support multi-statement 'SQL' scripts, which can be parameterised via metaprogramming structures and embedded 'R' expressions.

License GPL-3

Depends R (>= 3.3.0)

Imports RODBC

Suggests tools, utils

NeedsCompilation no

Repository CRAN

Date/Publication 2023-12-03 05:00:02 UTC

Contents

SQRL-package *Simple and Powerful ODBC Interfaces*

Description

Streamlines interactive exploratory work, and short-order ad hoc jobs, on ODBC data sources.

Details

Automatically generates a dedicated and like-named interface function to each ODBC DSN (Open DataBase Connectivity Data Source Name). These functions manage communications behind the scenes, whilst supporting multi-statement SQL scripts. Hybrid (SQRL) scripting syntax allows SQL with embedded R, thereby enabling parameterisation of queries, feedback of intermediate results, the addition of flow-control structures within and around SQL, and the use of libraries of stored scripts. Additional sources and interfaces can be defined at any time. The package is a wrapper about **RODBC**.

Author(s)

Mike Lee

See Also

sqrlSources, sqrlUsage

Examples

```
require(SQRL)

# Show (automatic) data sources.
sqrlSources()

## Not run:
# If 'ratatoskr' were one of those sources (i.e.,
# if a DSN of that name was found), then a multi-
# statement query could be submitted like so:
ratatoskr("use yggdrasil; select messages from ",
          "vedfolnir where addressee = 'nidhogg' ",
          "limit ", 5)

# Submit a parameterised query from file.
ratatoskr("messages.sqrl", year = 950)

# Obtain help on usage.
ratatoskr("help")

## End(Not run)
```

```
# Define a new data source (interface).
sqrlSource("mysource",
           driver = "MYSQL ODBC 5.3 ANSI Driver",
           server = "localhost",
           user = "<uid>",
           password = "<pwd>")

## Not run:
# Submit a query to the new source.
mysource("select * from database.table")

## End(Not run)
```

`sqrlAll` *Broadcast a Command to All Data Sources*

Description

Passes a single command to every **SQRL** data source in turn.

Usage

```
sqrlAll(...)
```

Arguments

`...`	The command to broadcast (as per `sqrlUsage`).

Value

Returns a named list, containing the result of the command for each data source. The list is invisible, except when retrieving (getting) a named parameter value.

Note

The command is passed to all **SQRL** data sources, whether or not they have interface functions.

The command can be a SQL query.

See Also

`sqrlOff`, `sqrlUsage`

Examples

```
# Show all interfaces (visible return).
sqrlAll("interface")

# Enable all connection indicators.
sqrlAll(visible = TRUE)

# Close all open channels.
sqrlAll("close")

# Remove all defined sources.
sqrlAll("remove")
```

---

sqrlConfig *Configuration Files*

Description

This material does not describe a function, but (rather) the file format used to configure **SQRL** interfaces and **RODBC** communications.

Configuration files can be used to define new data sources, set blanket parameter values for existing sources, or set individually named parameter values.

Example Configuration File

```
# Parameters for RODBC::odbcConnect/RODBC::odbcDriverConnect.
dsn                 = NULL
uid                 = "Blake"
pwd                 = 'C:/some/other/file.txt'
connection          = "driver=<driver>;dbalias=alpha;uid=<uid>;pwd=<pwd>;"
case                = "nochange"
believeNRows        = TRUE
colQuote            = c("`", "'")
tabQuote            = '"'
interpretDot        = TRUE
DBMSencoding        = ""
rows_at_time        = 100
readOnlyOptimize    = FALSE

# Additional parameters for RODBC::sqlQuery.
errors              = TRUE
as.is               = TRUE
max                 = 0
buffsize            = 1000
nullstring          = NA_character_
na.strings          = c("NA", "-", "")
dec                 = "."
```

```
stringsAsFactors    = FALSE

# Parameters for SQRL.
aCollapse           = ','
autoclose           = TRUE
driver              = "{IBM DB2 ODBC DRIVER}"
interface           = "Z"
lCollapse           = "\n"
library             = "my/library/file.sqrl"
ping                = "select 1 from dual"
verbose             = FALSE
visible             = TRUE
prompt              = "Z"
retry               = TRUE
scdo                = TRUE
wintitle            = "(Zen)"
```

Commentary on Example File

This is a sample configuration file, exhibiting almost all parameters. In general, a file need not include all of these (default values are in place).

Parameters may be defined as the path to some other file. The *driver* and *dsn* parameters will take that path as their value. For all other parameters, a value will be read from within the file. Such files may contain only a single line with nothing but the value on it, or they may adhere to the full (multiple line) 'parameter = value' format (as above).

Configuration scripts are parsed and evaluated as R, but any expressions in which the final assignment is made via = (as opposed to <-) are interpreted as requests to set **SQRL/RODBC** parameter values, rather than R environment variables.

See Also

sqrlParams, sqrlSource

Examples

```
# Define a new source (not from file).
sqrlSource("Orac", "UID=Avon;PWD=<pwd>",
           "Driver=Oracle 19 ODBC driver",
           "Server=db.starone.mil:1521/dwprd")

# Review its configuration (parameter values).
Orac("config")

# Create a file, containing only 'TRUE'.
file1 <- normalizePath(tempfile(), "/", FALSE)
writeLines("TRUE", file1)

# Create a file, containing named parameter values.
file2 <- normalizePath(tempfile(), "/", FALSE)
writeLines(c("dsn = 'Aristo'",
```

```
              "uid = 'Ensor'",
              "autoclose = TRUE",
              "as.is = FALSE"),
            file2)

# Create a configuration file, referencing the two above.
# Observe the use of temporary (non-parameter) variable 'x'.
file3 <- tempfile()
writeLines(c("aCollapse = ', '",
             paste0("verbose = \"", file1, "\""),
             "x <- 4",
             "max = sqrt(100 * x)",
             "as.is = TRUE",
             paste0("autoclose = '", file2, "'")),
           file3)
readLines(file3)

# Configure from the main (third) file.
Orac(config = file3)

# Alternative file-import forms.
Orac("config", file3)
Orac(paste("config", file3))

# Confirm imported values.
# Only 'autoclose' has been read from file2.
Orac("config")

# Import one parameter value from a file
# containing only a single unnamed value.
Orac(readOnlyOptimize = file1)
Orac("readOnlyOptimize")

# Import only a single specific parameter value
# from a file containing several named values.
Orac(uid = file2)
Orac("uid")

# Define and configure a new source from file.
sqrlSource("Caro", file2)
Caro("config")

# Configuration can also be performed from a named list,
# which could be generated by any R function or script.
Caro(config = list(autoclose = FALSE, max = 100))
Caro("config")[c("autoclose", "max")]

# Delete files.
unlink(c(file1, file2, file3))
```

`sqrlInterface` *Creates Data Source Interfaces*

Description

Creates, renames, and removes data-source interface functions.

Communications with data sources (including SQL queries) are conducted through these interfaces.

Usage

```
sqrlInterface(...)
```

Arguments

...	The name of a registered data source, and the name to use for its interface.

Details

The source and interface names may be supplied as two character strings, `("source", "interface")`, or as a `(source = "interface")`, or `("source" = "interface")`, parameter-value pair.

The setting of an interface whose name would clash with that of any other function already on the R search path is prevented. An error will be thrown if a potential conflict is detected. Conversely, a successful call of this function guarantees both the existence of the new interface, and the uniqueness of its name (amongst functions).

If the interface name is specified as either `NULL` or `"remove"`, then any existing interface is deleted (and no new interface is created).

If only a single string is supplied, the name of that source's interface function is returned.

Use of this functions is not ordinarily required, except when a registered data source does not already have an interface function.

Value

Returns the name of the source's interface function (visibly on get, invisibly on set).

Note

Interfaces are stored in a publicly accessible environment, `SQRL:Face`. This is attached to the R search path when the package is loaded.

See Also

`sqrlSource` `sqrlSources`

Examples

```
# Define a new data source, named 'entropy'.
sqrlSource("entropy", uid = "ludwig",
           driver = "{SQL Server Native Client 11.0}",
           server = "Clausius", database = "Gibbs")

# The source comes with an interface of the same name.
sqrlInterface("entropy")
entropy("sources")

# Change the name of the interface function.
sqrlInterface(entropy = "S")

# An equivalent alternative form.
sqrlInterface("entropy", "S")

# The name of the source remains unchanged.
sqrlInterface("entropy")
S("sources")

# Submit a dummy 'query', via the interface.
# Normally, you'd put some SQL in here, but
# the source would have to exist for that.
S("<R> 'hello, world'")

# Remove the source's interface function.
sqrlInterface(entropy = NULL)

# An equivalent alternative form.
sqrlInterface("entropy", "remove")

# The source remains, but has no interface.
sqrlInterface("entropy")
sqrlSources()
```

---

sqrlOff *Close Connections and Deactivate the Package*

Description

Closes all connections, detaches the interface environment (*SQRL:Face*) from the search path, and unloads the **SQRL** namespace. No further communication with any data source will be possible through **SQRL** (until it is reloaded).

Usage

```
sqrlOff()
```

Value

Returns invisible NULL.

Note

Calls to RODBC::odbcCloseAll will close any connection channels open in **SQRL**.

See Also

SQRL

Examples

```
## Not run:
# Calling sqrlOff() will deactivate and unload SQRL.
sqrlOff()

## End(Not run)
```

---

sqrlParams *Control and Communication Parameters*

Description

This material does not describe a function, but (rather) the various parameters governing ODBC communications and package behaviour. The majority of these are passed through to **RODBC**.

SQRL adopts a set-and-forget approach, wherein changes to the values of these parameters are persistent, and all subsequent communications make use of those values. Each registered (**SQRL**) data source has its own independent set of values.

Parameters

aCollapse: A character string (typically a single character). When an atomic object (typically a vector of character or integer type) is pasted into SQL, that object is first collapsed to a single string, with *aCollapse* separating each sequential pair of the object's elements. The default value is comma (",").

as.is: A logical vector, or a numeric vector (of column indices), or a character vector (of column names). Argument to RODBC::sqlQuery (see also utils::read.table). Tells **RODBC** which character columns of a table, as returned by a query to the ODBC connection, *not* to convert to some other data type (i.e., which character columns to leave as is). Due to **SQRL**'s set-and-forget approach to parameters, it is inconvenient to change *as.is* on a query-by-query basis. That being the case, it is often best defined as a logical singleton (either TRUE or FALSE). Alternatively, temporary query-specific values can be set within SQRL scripts (see sqrlSource). The default value is FALSE (convert all character columns).

autoclose: A logical singleton. Tells **SQRL** whether or not to automatically close the data source connection after each query (in general, a sequence of multiple statements). The default value is FALSE, which leaves the connection open. When set to TRUE, connections will open only for the duration of each query. When user input is required for authentication each time a new connection is opened, the default setting will be more convenient.

believeNRows: A logical singleton. Argument to RODBC::odbcDriverConnect. Tells **RODBC** whether or not to trust the nominal number of rows returned by the ODBC connection. Locked while the connection is open. The default value is TRUE, except for SQLite (FALSE).

buffsize: A positive integer. Argument to RODBC::sqlQuery. Specifies the number of rows (of a query result) to fetch at a time. The default value is 1000.

case: A character string, specifically one of "nochange", "toupper", "tolower", "mysql", "postgresql", or "msaccess". Argument to RODBC::odbcDriverConnect. Specifies case-changing behaviour for table and column names. Locked while the connection is open. The default value is "nochange".

channel: An **RODBC** connection handle. Returned by RODBC::odbcDriverConnect. Argument to RODBC::sqlQuery. This parameter is read-only.

colQuote: A character vector of length 0, 1, or 2, or NULL. Argument to RODBC::odbcDriverConnect. Specifies the quote character(s) for column names. A vector of length zero means no quotes, of length one means apply the specified quote character at both ends of a name, and of length two means apply the first character to the start of the name and the second character to the end of the name. Locked while the connection is open. The default value is a backtick for MySQL, and a double-quote for everything else.

connection: A character string. Argument to RODBC::odbcDriverConnect. Specifies an ODBC connection string. The content of this string will be database-management system (DBMS) dependent. Overrides *dsn*, should both be defined. Locked while the connection is open. Defaults to the empty string (connect via DSN instead). Will accept NULL as an alias for the empty string. Can be specified as a character vector of named (and/or unnamed) components, from which a single string will be constructed (see the examples in sqrlSource). Setting *connection* resets *dsn*, unless *connection* contains the "<dsn>" placeholder (see sqrlSource).

DBMSencoding: A character string. Argument to RODBC::odbcDriverConnect. Names the encoding returned by the DBMS. Locked while the connection is open. Defaults to the empty string (use encoding of the R locale). Will accept NULL as an alias for the empty string.

dec: A character string (typically a single character). Argument to RODBC::sqlQuery. Defines the decimal-place marker to be used when converting data from text to numeric format. The default value is options("dec"), as set by **RODBC**.

driver: A character string. The name or file path of the ODBC driver for the source (either currently in use, or to be used when a channel is opened). This determines the requisite dialect of SQL. Locked while the connection channel is open. Defaults to the empty string. Will accept NULL as an alias for the empty string.

dsn: A character string. Argument to RODBC::odbcConnect. Specifies the data source name (DSN) to connect to. Can be a file path. Overridden by *connection*, when that parameter is defined. Setting *dsn* resets *connection*, unless *connection* contains the "<dsn>" placeholder (see sqrlSource). Setting *dsn* also sets *driver*, if the DSN exists and the associated driver can be identified. Locked while the connection is open. Defaults to the empty string. Will accept NULL as an alias for the empty string.

errors: A logical singleton. Argument to RODBC::sqlQuery. Controls whether or not to throw R errors in response to DBMS/ODBC exceptions. The default value is TRUE (this differs from the **RODBC** default).

interface: A character string, or NULL. The name of the **SQRL** interface function for this data source (see sqrlInterface). Setting NULL or "remove" removes the interface. The default value is NULL (undefined).

interpretDot: A logical singleton. Argument to RODBC::odbcDriverConnect. Locked while the connection is open. Controls whether or not to interpret table names of the form "aaa.bbb" as table "bbb" in schema/database "aaa". The default value is TRUE.

lCollapse: A character string (typically a single character). When a list-like object (typically an actual list) is pasted into SQL, that object is first collapsed to a single string, with *lCollapse* separating each sequential pair of the object's elements (to each of which, *aCollapse* will have first been applied). The default value is the empty string ("").

library: A character vector. Empty by default. Holds named procedures, as defined by the user (see sqrlScript and sqrlUsage). Setting to NULL empties the library.

max: An integer. Argument to RODBC::sqlQuery. Caps the number of rows fetched back to R. The default value is 0 (meaning unlimited; retrieve all rows).

na.strings: A character vector. Argument to RODBC::sqlQuery. Specifies strings to be mapped to NA within character data. The default value is "NA".

name: A character string. The name of this **SQRL** data source. While often identical to the names of both the underlying ODBC data source and the **SQRL** interface function, it need match neither in general. Multiple **SQRL** sources may interface with the same ODBC source. This parameter is write once, and cannot be changed after creation of the **SQRL** source. There is no default value.

nullstring: A character string. Argument to RODBC::sqlQuery. The string with which to replace SQL_NULL_DATA items within character columns. The default value is NA_character_.

ping: A character string. Defines a reliable, trusted, SQL statement, used by **SQRL** to verify source connections. The initial value is NULL, which causes a simple, DBMS-dependent, statement to be determined the first time a connection is opened. Manual definition may be necessary in the event that **SQRL** fails to identify an appropriate statement for the particular DBMS of the source. An invalid ping statement may lead to incorrect assessments of whether or not the connection is open. Manually setting NULL causes the statement to be redetermined the next time a connection is opened. Pings are submitted verbatim, without passing through **SQRL**'s parser.

prompt: A character string (typically a single character). Defines an indicator to be applied to the R command prompt when the connection is open and *visible* is TRUE. Defaults to the first character of *name*. Single-letter indicators are recommended since, if two sources are open and one indicator is a substring of the other, then **SQRL** may fail to correctly update the prompt when one source is closed. Can be set to an empty string, in which case nothing is applied to the prompt. Will accept NULL as an alias for the empty string.

pwd: A character string. Argument to RODBC::odbcConnect. Specifies a password to use at the next authentication request. This need not match the password that was used to open the current channel. Defaults to the empty string (interpreted as do not supply a password to the ODBC driver). Will accept NULL as an alias for the empty string. Write-only.

readOnlyOptimize: A logical singleton. Argument to RODBC::odbcDriverConnect. Specifies whether or not to optimise the ODBC connection for read-only access. Locked while the connection is open. The default value is FALSE.

result: An arbitrary object, being the final outcome of the last successful query or procedure. Read-mostly. Can be set to NULL (its default value), to free memory.

retry: A logical singleton, TRUE by default. Should a query fail due to an apparent network outage or other unexpected loss of connection, a *ping* (above) is made to verify that occurrence before reconnecting and resubmitting the failed query. This process is automatic and silent, unless manual input is required for authentication. However, as any temporary tables will not have survived the initial connection loss, a non-existence error may eventually occur. Failure of the second attempt is always fatal (no third attempt will be made). Changing *retry* to FALSE disables this action.

rows_at_time: A positive integer, between 1 and 1024. Argument to RODBC::odbcDriverConnect. Specifies the number of rows to fetch at a time when retrieving query results. Locked while the connection is open. The default value is 100. Manually setting 1 may be necessary with some ODBC drivers.

scdo: A logical singleton. Controls **SQRL** parser behaviour. When TRUE (the default), the parser splits multi-statement scripts on what it considers to be statement-terminating semicolons; submitting the individual statements as each such semicolon is encountered. This may fail in the presence of DBMS-specific procedural-language syntax. When FALSE (the fallback mode), sequential statements are delimited only by SQRL tags, especially the <do> tag (see sqrlScript).

stringsAsFactors: A logical singleton. Argument to RODBC::sqlQuery. Controls the conversion of character columns to factor columns within query results, excluding those columns covered by *as.is*. The default value is FALSE.

tabQuote: A character vector of length 0, 1, or 2, or NULL. Argument to RODBC::odbcDriverConnect. Specifies the quote character(s) for table names. A vector of length zero means no quotes, of length one means apply the specified quote character at both ends of a name, and of length two means apply the first character to the start of the name and the second character to the end of the name. Locked while the connection is open. Defaults to the value of *colQuote*.

uid: A character string. Argument to RODBC::odbcConnect. Specifies the user identity (UID, user name) to use on the data source. Locked while the connection is open. Defaults to the local name of the R user (Sys.info()["user"]). Will accept NULL as an alias for the empty string (which is interpreted as do not pass a UID to the ODBC driver). May be inaccurate when the UID is specified within a DSN.

verbose: A logical singleton. Controls whether or not to display verbose output during query submission. Intended mainly for debugging. The default value is FALSE (verbose output disabled). Verbose output is always disabled within non-interactive sessions.

visible: A logical singleton. Toggles display of the *wintitle* and *prompt* indicators (while an open connection channel exists to the source). The default value is FALSE (do not show indicators). Changing this to TRUE authorises modification of the "prompt" global option (see base::options).

wintitle: A character string, possibly empty. Will accept NULL as an alias for the empty string. Defines an indicator that, unless empty, is displayed on the R window title bar while a connection channel is open to the source, and provided *visible* is TRUE. An asterisk (*) is appended to the indicator while a query is running on the source, and a plus-sign (+) is appended while results

are being retrieved from it. A question mark (?) is appended during connection-testing pings, but these are usually fleeting. If two sources are open and one indicator is a substring of the other, then **SQRL** may fail to correctly update the title when one source is closed. Only works with 'R.exe', 'Rterm.exe' and 'Rgui.exe', and then only while running on a "Windows" operating system. Works with both MDI and SDI modes, but does not work with "RStudio".

Note

Each **SQRL** data source has its own set of the above parameters. Altering the value of a parameter (e.g., *stringsAsFactors*) for one source does not affect the value for any other source. Use sqrlAll to make blanket changes.

See Also

sqrlUsage, **RODBC**

Examples

```
# Define a new source.
sqrlSource("thoth", dbcname = "Karnak",
           driver = "Teradata Database ODBC Driver 16.10")

# Retrieve all parameter values.
thoth("config")

# Retrieve a (fixed) subset of parameter values.
thoth("settings")

# Retrieve a single (named) parameter value.
thoth("as.is")

# Various means of setting a value.
thoth(as.is = TRUE)
thoth("as.is" = TRUE)
thoth("as.is", TRUE)
thoth("as.is", "TRUE")
thoth("as.is TRUE")

# If you wanted the string 'TRUE'.
thoth(as.is = "TRUE")
thoth("as.is 'TRUE'")

# Various means of setting multiple values.
thoth(as.is = TRUE, stringsAsFactors = FALSE)
thoth(list(as.is = TRUE, stringsAsFactors = FALSE))
thoth(config = list(as.is = TRUE, stringsAsFactors = FALSE))
```

sqrlScript *Hybrid Script Syntax*

Description

This material does not describe a function, but (instead) the SQRL scripting syntax for SQL with embedded R.

For instructions on how to submit (run) these scripts from file, refer to sqrlUsage.

Script #01 (Pure SQL)

```
use database;
select * from table;
```

Commentary on Script #01: Multi-statement SQL scripts can be taken directly from 'SQL Developer' (or similar application), and (usually) run without modification. The **SQRL** parser identifies statement-terminating semicolons, and submits each statement in turn, as those semicolons (or the end of the script) are encountered.

Script #02 (SQL, with R in it)

```
select isotope from periodic_table
where atomic_number = <R> Z </R>
```

Commentary on Script #02: Scripts can be parameterised, via embedded R. In SQRL, XML-style tags mark the beginning and end of an R block. As shown in sqrlUsage, values can either be passed explicitly, or inherited from the calling environment. Embedded R is not restricted to the insertion of values for filtering, and can be used to specify column names, clauses, or entire SQL statements.

Tags are not case sensitive.

Script #03 (Comments, and Arbitrary R)

```
/* SQL-comment within SQL */
-- SQL-comment within SQL
select
  <R>
    /* SQL-comment within R */
    -- SQL-comment within R
    # R comment within R
    letter <- sample(letters, 1)
    paste0("'", letter, "'")
  </R>
```

Commentary on Script #03: The parser supports R comments, and two kinds of SQL comments. For syntax-highlighting purposes, SQL comments may be used within R sections (but R comments

cannot be used within SQL). Since `a -- b` is legitimate R, it would need to be rewritten with a space between the minus signs, or as `a + b`, to avoid being interpreted as a comment.

Otherwise-arbitrary R script is allowed within (tag-delimited) R sections. Evaluation of such R script takes place in a temporary child environment of the calling environment. The final result of that evaluation is then pasted into the surrounding SQL, before its submission. This mechanism cannot be used to insert additional tags into the SQRL script.

Script #04 (R, out of SQL)

```
<R>
  threshold <- 9000
  sq <- function(x) paste0("'", x, "'")
<do> -- This tag terminates the R section.

<R>
  if (!exists("date", inherits = FALSE))
    date <- format(Sys.Date(), "%Y-%m-%d");
; -- This semicolon terminates the R section.

select * from database.table
where event_date = <R> sq(date) </R>
and event_size > <R> threshold </R>
```

Commentary on Script #04: As in scripts #02 and #03, when an R section ends with `</R>`, the result of evaluating that section is pasted into the surrounding SQL (or implied SQL, if those surroundings are blank). However, an R section can also be terminated by a `<do>` tag, or by a non-syntactical semicolon (that being one with nothing but whitespace between it and the previous semicolon, newline, or `<R>` tag). Such sections are evaluated in the same way as those ending in `</R>` tags, but the results are not pasted into SQL. These sections can only appear between, not within, SQL statements.

In this script, the first out-of-SQL R section defines a numerical constant, and also a function for wrapping strings in single quotes. The second out-of-SQL R section supplies a default value for a variable, *date*, if none was explicitly passed in (see sqrlUsage). These two R sections could be combined into one. The function and values from these sections are applied in the subsequent SQL.

Script #05 (Temporary Parameter Settings)

```
<with>
  na.strings = c('NA', "-")   -- comment
  x <- 5; max = x + 1         /* comment */
  verbose = TRUE               # comment
  as.is = "configfile.txt"     # read from file
</with>

-- Run this query with the above settings.
select * from database.table
```

Commentary on Script #05: Tags `<with>` and `</with>` delimit special-purpose blocks of R. They are used to set temporary parameter values, which remain in effect only for the duration

of the script. To be clear, temporary values survive beyond the </with> tag, but not beyond the end of the script. The exception is that temporary values of *ping* are retained if and only if the connection would otherwise be left open without a defined *ping*.

This mechanism only works for a limited number of parameters: *aCollapse*, *as.is*, *buffsize*, *dec*, *errors*, *lCollapse*, *max*, *na.strings*, *nullstring*, *ping*, *pwd*, *retry*, *scdo*, *stringsAsFactors*, and *verbose*.

Script #06 (Procedural Language Blocks)

```
select 1 from dual; -- This semicolon terminates the statement.

declare
  x integer := 0;
begin
  for i in 1..10 loop
    x := x + 1;
    if x > 7 then
      x := 0;
    else
      begin
        null;
      end;
    end if;
  end loop;
end; -- This semicolon terminates the block.

select 2 as N from dual
```

Commentary on Script #06: As with statement-terminating semicolons (refer to script #01), the **SQRL** parser attempts to identify procedural-language block-terminating semicolons, and submits each such block as those semicolons are encountered. The parser also submits (when necessary) upon reaching the end of the script. Although trailing semicolons are not usually mandatory for SQL statements, they usually are mandatory at the end of a procedural block.

If you have a script that came with forward-slashes, /, at the end of procedural blocks, those slashes will normally need to be removed (as, say, 'SQL Developer' would do for you).

When a script contains multiple statements and/or blocks, only the final result is returned. In this case, that would be `data.frame(N = 2L)`.

Script #07 (Parser Control)

```
-- Change parser to fallback mode.
<with>
  scdo = FALSE
</with>

-- An outside-of-SQL R section.
<R>
  N <- 2;
<do> -- This tag terminates the section.
```

```
-- The default (scdo = TRUE) SQRL parser would
-- not find the end of this procedural block.
begin
  null;
end /**/ ;
<do> -- This tag terminates the block.

select 1 from dual; -- Semicolon ignored.
<do> -- This tag terminates the statement.

select <R> N </R> from dual
```

Commentary on Script #07: While the parser detects terminal semicolons at the ends of SQL statements and R sections, it remains unsophisticated, and ignorant of DBMS-specific procedural-language syntaxes. That being the case, if you are working with procedural-language extensions to SQL, sooner or later the parser will fail to detect the end of a block. Presently, a simple thing that causes this is the presence of a comment between an end and its semicolon (as appears, above).

Setting the *scdo* parameter to FALSE causes the parser to ignore semicolons and to conclude (out of SQL) R sections, SQL statements, and procedural blocks only upon encountering a <do> tag. This provides a robust multi-statement capability when need be, but will require appropriate modification of any scripts originally developed for some other application.

Note that <do> tags function irrespective of the *scdo* setting. That being the case, the use of these can be preferable to semicolons when a SQRL script is developed from scratch. When *scdo* is TRUE, and a terminating semicolon is followed by a <do> tag, with nothing but whitespace between them, they are treated as a single <do> (only one submission is made). An implied do tag exists at the end of every script.

Script #08 (Manipulation of Results)

```
select
  calDate, Snowfall
from
  Weather.SparseDailySnowfall
where
  calYear = <R> year </R>

-- Submit the above, assign the result to 'a',
-- and immediately begin an R section.
<result -> var>

  names(var)[names(var) == "calDate"] <- "Date"
  var$Snowfall <- as.numeric(var$Snowfall)
  first <- as.Date(paste(year, 1, 1, sep = "-"))
  last <- as.Date(paste(year, 12, 31, sep = "-"))
  alldates <- data.frame(Date = seq(first, last, 1))
  merge(alldates, var, all.x = TRUE)
```

Commentary on Script #08: A <result> tag acts as a combination of the <do> and <R> tags,

wherein the result of the query is assigned to an object within the working (script evaluation) environment. Any syntactically valid R variable name could be used in place of *var*. Whilst the main tag is not case sensitive, the name of the variable is.

An R section begins immediately after the <result> tag. This can be useful when preferred R column names are reserved SQL keywords (such as "date"), when dates come back in unconventional formats (and need conversion), when strict type-conversion control is required (typically in combination with as.is = TRUE), or when data is sparse (zero-valued entries are not stored) and results need to be expanded to explicitly include the implied zero-valued data. The net effect is a single script, combining SQL data extraction with R reformatting.

Using "null" or "NULL" as the variable name stops assignment of the query result (but the R section still begins).

Only the final value of any SQRL script is returned. In this example, that value is the merged data frame.

Note that <result> tags cannot be used to conclude R sections. Also, R sections beginning with <result> tags cannot end in </R>.

When *scdo* is TRUE, and a terminating semicolon is followed by a <result> tag, with nothing but whitespace between them, they are treated as a single <result> tag (only one query submission is made). However, when a <do> tag is followed by a <result> tag, two submissions are made, irrespective of *scdo*. First, the query before the <do> is submitted, and then the (blank) query between the <do> and the <return>. In **SQRL**, blank queries (blank SQL statements) are always allowed, and return no value.

Script #09 (Feedback of Intermediate Results)

```
select distinct customer_number from our.customers
where town in (<R> paste0("'", towns, "'") </R>)

<result -> z>; -- Same thing as <result -> z><do>

select sum(transactions) from online.orders
where customer_number in (<R> z$customer_number </R>)
```

Commentary on Script #09: This script takes one parameter, *towns*, a character vector. By default, atomic vectors are pasted into SQL as comma-separated values. The first query returns a data frame, which we save, as *z*, but have no need to manipulate. Its (integer-valued) *customer_number* column is then applied to the second query, once again as comma-separated values. Only the final result (of the second query) is returned.

Of course, this particular example could instead have been implemented with a join, or temp table.

The pasting behaviour of atomic objects is controlled by the *aCollapse* parameter. Similarly, the pasting behaviour of list-like objects is controlled by the *lCollapse* parameter (which defaults to the empty string). To restore **SQRL**-0.6.3 pasting rules, insert <with> aCollapse = "\n"; lCollapse = ", " </with> at the top of a script. See sqrlUsage and sqrlParams for further detail.

Script #10 (Combining Data Sources)

```
<R>
  n <- OtherSource("select distinct customer_number ",
```

```
                    "from our.customers where town in "
                    "(" paste0("'", towns, "'") ")")
<do>

select sum(transactions) from online.orders
where customer_number in (<R> n$customer_number </R>)
```

Commentary on Script #10: This is a repeat of script #09, except now the tables of the first and second queries are hosted on two entirely separate data sources. The function `OtherSource` is the **SQRL** interface to the data source of the first query, while this script should be run from the interface to the final source. Again, there are other ways to do this, but the general idea is for the main R script (performing the modelling and analysis) to make a single call "get me this data", without clutter, or concern for the true horror of where that data comes from.

Script #11 (The sqrl Function)

```
<R>
  -- Check this is the intended source.
  stopifnot(sqrl("name") == "MySource")

  -- Remember initial parameter values.
  initials <- sqrl("settings")

  -- Set some parameter values.
  sqrl(stringsAsFactors = FALSE,
       wintitle = "[MySource]",
       visible = TRUE)
;

-- Submit the query, record the result.
select * from database.table
<result -> out>;

<R>
  -- Restore initial parameter values.
  sqrl(initials)

  -- Return the query result.
  out
```

Commentary on Script #11: The function `sqrl(...)` is a special interface that is automatically defined into the temporary script-evaluation (working) environment. It acts as a cut-down `sqrlAll`, relaying its arguments only to whichever **SQRL** data source is executing the script. It works even when the name of the invoking interface function changes, or when that interface function does not exist. Consequently, `sqrl` should be adopted in preference to hard-coding interface names into scripts.

Here, the `sqrl` function is first used to provide an assert, which, in this case, verifies the script is being run only on the intended data source. A second call then takes a snapshot of the current

RODBC/SQRL parameter settings (the `settings` command is described in `sqrlUsage`). A third call then sets new values for three of those parameters, prior to submission of the SQL query.

Due to **SQRL**'s set-and-forget approach, these values are persistent, and (unless they are changed again) will remain in effect after execution of the script has completed. That being the case, another call of `sqrl` is made after the query, in order to restore the original parameter values.

Note that (temporary) parameter values set via `<with> ... </with>` take precedence over (persistent) values set via `sqrl(...)`, irrespective of the order in which those settings are made within the script. Also, whilst (for example) `sqrl(max = 1)` sets the persistent value for parameter *max*, `sqrl("max")` returns the value currently in effect (which might be a temporary value). Hence, for purposes of setting script-specific parameter values, using `<with>` is, when possible, generally preferable to calling `sqrl`.

As shown in script #17, the `sqrl` function can also be used to make nested queries (its arguments could just as easily be another script).

When a script is being run from some particular interface, explicit calls to that same interface function (as opposed to `sqrl`) are normally blocked. Calls to `sqrlAll`, `sqrlInterface`, `sqrlOff`, `sqrlSource`, and `sqrlSources` are also normally blocked. If you really want a way around these blocks, calls can be made to `SQRL::sqrlAll`, instead of just `sqrlAll`, and so on. Alternatively, `<R> rm("sqrlAll") <do>` (et cetera) removes the block (this also works for unblocking the interface function).

Script #12 (Closing Connections)

```
-- Ensure readOnlyOptimise is TRUE.
<R>
  if (!sqrl("readOnlyOptimize"))
  {
    sqrl("close")
    sqrl(readOnlyOptimize = TRUE)
  }
<do>

-- Pull data (reopening is automatic).
select * from database.table;

-- Close the connection.
<close>
```

Commentary on Script #12: Many communications parameters are "locked while open", and cannot be changed while a connection (channel) exists to the source (see `sqrlParams`). In this example, we want a particular value for one such parameter, namely *readOnlyOptimize*. If its value needs changing, we must first ensure the channel is closed. Within the R section, this is achieved with `sqrl("close")` (see `sqrlUsage`).

When the SQL query is submitted, **SQRL** will automatically open a new connection, if need be.

After the data is pulled, the `<close>` tag closes the channel (because, in this example, we do not want to leave it open). Unlike `<R>sqrl("close")<do>`, `<close>` tags return no value of their own, which means the final value of this SQRL script is still that of the query (as we require).

Be aware that `<close>` does not imply `<do>`, and it is an error to use `<close>` in the presence of partially-formed, or unsubmitted, SQL. Conversely, `<R>sqrl("close")</R>` is allowed within

SQL.

An alternative to putting <close> at the end of a script is to set the *autoclose* parameter to TRUE (see sqrlParams). Placing <close> at the beginning of a script can be used to ensure no temporary tables are in existence (when no better mechanism is available).

When user-input is required for authentication on the opening of a new channel, connection closures should be kept to a bare minimum.

Script #13 (Returns)

```
-- This selects 1 (the result of the embedded R expression).
select <R> return(1); 2 </R>

-- This return doesn't exit the script either (only the R section).
<R>
  return(1)
  print("this won't be printed")
<do>

-- Pull from a temporary table, and save the result.
select some_columns from temp_table
<result -> a>;

-- Drop the temp table (this returns a value).
drop temp_table;

-- Return the value of interest (exit the script).
<return (a)>

-- This is unreachable.
select 1
```

Commentary on Script #13: The difference between a <return (a)> and <R>return(a)<do>, is that the former exits the SQRL script, while the latter only exits from the (embedded) R section (back into SQL), before continuing with the script.

Almost any valid R expression is allowed between the (mandatory) parentheses of a <return> tag (see script #03). Note that <return> tags are not recognised within R sections. Also, <return> tags do not imply <do>, and it is an error to attempt a <return> within partially-formed (unsubmitted) SQL. Because x <- y returns invisibly, so too do <R> x <- y <do> and <return (x <- y)> (see example script #16).

The next example shows how <return> tags become much more useful when combined with conditional expressions.

Script #14 (If, Else If, Else)

```
-- Submit this when we have a 9-digit code.
<if (nchar(code) == 9)>
  select category from item_category
  where long_item_code = <R> code </R>
  <result -> k> k <- as.integer(k) <do>
```

```
-- Submit this when we have a 6-digit code.
<else if (nchar(code) == 6)>
  select category from item_category
  where short_item_code = <R> code </R>
  <result -> k> k <- as.integer(k) <do>

-- Exit here when we have neither
-- (because the next query would fail).
<else>
  <return (NULL)>
</if>

-- Obtain all items in the same category,
-- in the original (long or short) format.
select
  category,
  <if (nchar(code) == 9)>
    long_item_code
  <else>
    short_item_code
  </if>
from item_category
where category = <R> k </R>
```

Commentary on Script #14: In this example, we have a table of (say) stock items, each of which has both a long (9 digit) and short (6 digit) identity code, and is assigned to some kind of category (with another integer identifier). The script takes a single integer argument, *code*, which might be in either the long or short format.

The `<if>` and `<else if>` tags are used to submit an appropriate query, according to the type of identity code supplied. In the event that the code is of an unrecognised type, the `<else>` tag is used, with `<return>`, to exit cleanly (without submitting any query). Provided that the code is of a recognised type, a second query is submitted, wherein `<if>` and `<else>` control which column is selected.

In the final query, the appropriate column could instead have been named from within `<R>` and `</R>` tags, but the earlier conditional submission and conditional return cannot easily be achieved in that way. Essentially arbitrary R is allowed between the (mandatory) parentheses of an `<if>` or `<else if>` tag (see script #03), but the final result of evaluating that R must be either `TRUE` or `FALSE`. None of these tags are recognised within R sections. To enable their intra-statement application (as in the final query of the example script), none of these tags imply `<do>`. The space in `else if` is optional (i.e., `elseif` is equally valid).

Script #15 (Caching)

```
-- If we already have a cached copy of the
-- data, return that.
<if (exists(.cached, .GlobalEnv))>
  <return (get(.cached, .GlobalEnv))>
</if>
```

```
-- Otherwise, submit the query and cache the
-- result before returing it.
select * from table
<result -> x>
  assign(.cached, x, .GlobalEnv)
  x
```

Commentary on Script #15: In this example, the result of a large query is cached locally, with <if> and <return> logic being used to return that cached result should the script be run again.

This is most useful when the data changes infrequently, the query takes a long time to run, and the script is stored as a procedure in the library (see script #18).

Script #16 (While Loops)

```
<R>
  batch <- split(ID, ceiling(seq_along(ID) / 1000))
  x <- NULL
  i <- 1
<do>

-- Pull and accumulate results, a thousand at a time.
<while (i <= length(batch))>
  select idnumber, name from identity_lookup
  where idnumber in (<R> batch[[i]] </R>)
  <result -> y>
    i <- i + 1
    x <- rbind(x, y)
  <do>
</while>
```

Commentary on Script #16: In this example, we have a script with a single argument; ID, a vector of integer codes. Supposing that vector might be too long for the SQL in operator, a <while> loop is used to pull the results in batches.

Essentially arbitrary R is allowed between the (mandatory) parentheses of a <while> tag (see script #03), but the final result of evaluating it must be either TRUE or FALSE. As with the <if> family, <while> and </while> tags are not recognised within R sections, and do not imply <do>. Ordinarily, loops over SQL should be avoided, or used only as a last resort, but there are use-cases (see script #18), including for the insertion of rows.

Referring back to script #13, the final result of script #16 (that of x <- rbind(x, y)) is invisible. If visible output is required, <R>x or <return (x)> could be appended to the script.

The parser is simple, and does not verify or enforce correct nesting structure. Unintuitive output may appear when nesting violations occur.

Script #17 (Procedures)

```
<proc "random patients">
  select patient_number
    from patient_details
```

```
    order by rand()
    limit <R> N </R>
  <result -> a>
    a[, 1L]
</proc>

select days_in_hospital
from patient_history
where patient_number in
  (<R> sqrl("random patients", N = 200) </R>)
```

Commentary on Script #17: It is possible to define reusable SQRL procedures within (between) <proc> and </proc> tags. Here, a (parameterised) procedure is employed as a nested-query alternative to the sequential feedback mechanism of script #09.

Each procedure must be named (in its <proc> tag), with quotes (either single or double) being mandatory about that name. The end of a procedure definition acts in the same way as the end of a script (as an implied <do>). Note that </proc> tags are only recognised within R sections, when (as is the case in this example) the R section is within a procedure definition (i.e., under a <proc> tag). Opening <proc> tags are never recognised within R sections.

Script #18 (Libraries)

```
<proc "add na.strings">
  -- Takes one argument, 'add' (a character vector),
  -- and adds its strings to the na.strings parameter.
  <R>
    sqrl(na.strings = unique(c(sqrl("na.strings"), add)))
  <do>
</proc>

<proc "drop if exists">
  /* Takes one argument, 'tables', being a
     character vector of tables to be dropped. */
  -- Force as.is to be TRUE (for the query).
  <with>
    as.is = TRUE
  </with>
  -- Pull details of temporary tables.
  help volatile table
  <result -> v>
  <do>
  -- Exit here when the above query did not return
  -- a data frame (when no volatile table exists).
  <if (class(v) != class(data.frame()))>
    <return (invisible())>
  </if>
  -- Extract the names of all volatile tables
  -- in existence, retain only those (unique)
  -- tables to be dropped that actually exist,
```

```
  -- and initialise the iterator.
  <R>
    volatiles <- as.character(v[, "Table SQL Name"])
    tables <- unique(tables[tables %in% volatiles])
    i <- 0L
  <do>
  -- Drop each requested table (that exists).
  <while (i <- i + 1L; i <= length(tables))>
    drop table <R> tables[i] </R> <do>
  </while>
  -- Return invisible NULL.
  <return (invisible())>
</proc>
```

Commentary on Script #18: The previous example, script #17, exhibited a procedure defined as a utility within a larger SQRL script. Such definitions are not persistent, with the procedures vanishing upon the conclusion of their parent script.

However, as shown in sqrlUsage, it is possible to construct a persistent library of procedures. As is the case above, SQRL scripts intended for libraries must consist only of procedure definitions; no other SQL or R is allowed. This example script defines two procedures (both parameterised). The first, "add na.strings", adds strings to the existing *na.strings* vector (refer to sqrlParams). The second, "drop if exists", implements that capability for Teradata SQL (which doesn't).

Following sqrlUsage, let's say the above script is recorded in a file '`library.sqrl`', and that we have a **SQRL** interface function called `owl`. The library is then established with `owl(library = "library.sqrl")`, and the procedures are called with (for instance) `owl("add na.strings", add = c("N/A", "-"))`, and `owl("drop if exists", tables = c("tableA", "tableB"))`.

Ultimately, procedures do not confer any capability beyond that of SQRL files. Procedure libraries merely allow the consolidation of multiple files into one. They also offer another slight advantage in that when you change working directory, they come along.

Script #19 (Stop)

```
select 1

-- This ends the script.
<stop>

-- This is unreachable.
select 2
```

Commentary on Script #19: Lastly, `<stop>` tags act as an early end-of-script (with its implied `<do>`). They apply within R sections, as well as SQL, and even on the inside of a `FALSE` conditional block (i.e., the `<stop>` still functions in `<if (FALSE)><R><stop>`). They are intended as a troubleshooting aid.

Summary of Tags

`<R>`**:** Begins an R section (leaves SQL). Once begun, only `</R>`, `<do>`, `;`, `<stop>`, `EOS`, and `</proc>` are recognised (and can end the section). The `</proc>` case is only recognised when the initiating `<R>` tag is within a procedure definition (beneath a `<proc>` tag).

`</R>`: Ends an R section, causing that section to be evaluated. The result of that evaluation is pasted back into the surrounding SQL.

`<do>`: If inside an R section, ends and evaluates that section (without pasting the result back into SQL). If outside of an R section, causes the preceding SQL (which may be blank) to be submitted.

`<result -> name>`: Submits the preceding SQL, and assigns the result of that submission to the R variable `name`. Any syntactically valid R name, or "NULL", is allowed (in place of `name`). An R section begins immediately after the tag. That section concludes with any of the tags listed (above) for `<R>`, besides `</R>`.

`<close>`: Closes the ODBC channel (connection) to the data source.

`<return (Rexp)>`: Evaluates `Rexp`, which can be any R expression, and returns the resulting value (exits the script). The parentheses are mandatory.

`<with>`: Begins a special R section, for assigning temporary values to **RODBC/SQRL** parameters. Once begun, only `</with>`, `<stop>`, `EOS`, and `</proc>` are recognised (and can end the section). The `</proc>` case is only recognised when the initiating `<with>` tag lies within a procedure definition (beneath a `<proc>` tag). Only the *aCollapse*, *as.is*, *buffsize*, *dec*, *errors*, *lCollapse*, *max*, *na.strings*, *nullstring*, *ping*, *pwd*, *retry*, *scdo*, *stringsAsFactors*, and *verbose* parameters permit temporary value assignments.

`</with>`: Ends an R section begun by `<with>`. Causes the section to be evaluated, and assigns temporary parameter values accordingly.

`<if (Rexp)>`: Evaluates `Rexp`, which can be any R expression. If that expression evaluated to `TRUE`, the (arbitrary) script beneath the tag is acted upon. If that expression evaluated to `FALSE`, the (arbitrary) script beneath the tag is ignored (except for any `<stop>` tags). The parentheses are mandatory.

`<else if (Rexp)>`: Acts as `<if (Rexp)>`, when the (mandatory) previous `<if ((Rexp)>`, and all the (multiple, optional) previous `<else if (Rexp)>` tags, evaluated to `FALSE`. Otherwise (when any of those evaluated to `TRUE`), acts as `<if (FALSE)>`. The parentheses are mandatory.

`<else>`: Acts as `<else if (TRUE)>`.

`</if>`: Marks the end of an `<if (Rexp)>` - `<else if (Rexp)>` - `<else>` flow-control structure.

`<while (Rexp)>`: Acts in the manner of `<if (Rexp)>`. The parentheses are mandatory.

`</while>`: Marks the end of a `<while (Rexp)>` flow-control loop. If the previous `<while (Rexp)>` evaluated to `TRUE`, the parser returns to that `<while (Rexp)>` tag, and re-evaluates the R expression. Otherwise (if the previous `<while (Rexp)>` evaluated to `FALSE`), no action is performed (the parser continues from this point).

`<proc "name">`: Marks the beginning of the definition of a procedure called *name*. Any character string can be used in place of *name*. The quotation marks are mandatory (but can be singles or doubles).

`</proc>`: Marks the end of a procedure definition (acts as an *EOS* for that procedure).

`<stop>`: Acts as *EOS*. Applies even from within a `FALSE` conditional block. Intended for debugging, only.

`;`: When the *scdo* parameter is at its default value of `TRUE` (see `sqrlParams`), the SQRL parser attempts to identify SQL statement terminating semicolons, procedural language block terminating semicolons, and any extra semicolons within R sections. When detected, these all act as `<do>`.

EOS: The end of the script (*EOS*), acts as `<do>`. As necessary, it also acts as `</if>` and `</proc>` (it does not act as `</while>`).

Note

The *verbose* parameter toggles extended output when running scripts (see sqrlParams). This includes the display of intermediate values.

See Also

sqrlUsage

---

sqrlSource *Define New Data Sources*

Description

Defines (registers) new data sources and creates the interface functions for communicating with them. For DSNs, this process occurs automatically when **SQRL** is loaded, thereby making the manual use of this function unnecessary for those sources. The function can also redefine or delete (deregister) existing sources.

Usage

```
sqrlSource(...)
```

Arguments

...	A name and definition for the source (refer to the details section, below).

Details

The arguments must contain (at least) a name and definition for the source. In simplest form, these could be given as either (*name*, *definition*) or (*name* = *definition*), where both *name* and *definition* are single character strings. In decreasing order of precedence, the definition can be the path of a configuration file (containing a connection string or DSN, as per sqrlConfig), the name of an existing **SQRL** source (to copy all settings from), an ODBC connection string (as a character vector of components, or as a single string containing the equals character; =), or as the name of a DSN.

When clarity is required, the keywords `config`, `copy`, `connection` and `dsn` can be used to explicitly specify a configuration file, existing source, connection string or DSN, respectively. If the definition is given as multiple terms, and none of these four keywords are present, or if one of the named terms does not correspond to the name of an **RODBC/SQRL** parameter, then the terms are assumed to be components of a connection string. If, instead, the definition is given as multiple terms and at least one of these four keywords is present, and when all of the remaining terms appear to be **RODBC/SQRL** parameters, then those remaining terms will be treated as such (rather than as connection string components). The examples (below) should illustrate these statements.

Whichever form of definition is employed, the new interface name (which defaults to the source name) must not conflict with that of any function on the R search path (or else an error will be thrown).

Redefinition of an existing source is allowed, provided it is closed at the time.

When the source name is "remove", the definition is interpreted as a list of sources to be deregistered. This precludes the use of "remove" as a source name. Alternatively, redefining a source to NULL also deregisters the source.

Value

An invisible list of the new source's parameter values.

Note

Source definitions are not checked for validity (specified connection strings need not be correct, specified DSNs need not exist).

Connection strings may include placeholders; "<dsn>", "<driver>", "<uid>", and "<pwd>", to be replaced with the corresponding parameter values on the opening of a channel. These placeholders are case sensitive (see sqrlParams).

In 'Rgui.exe', the ODBC driver may, via **RODBC**, prompt for missing connection details (username, password, etc.). In other R applications, those details will need to be complete (no prompting occurs).

See Also

sqrlConfig

Examples

```
# Define a new source, 'A', by a connection string. Alternatively,
# the string could be replaced with the name of a DSN, the path of
# a config file, or the name of an existing source (to be copied).
# This particular connection string would be for a GNU/Linux system
# upon which the unixODBC driver alias 'MariaDB' has been defined,
# in addition to the 'MDB' alias for the server address. In general,
# there should be no space between 'Driver=' and the driver name.
sqrlSource(A = "Driver=MariaDB;Server=MDB;User=zarkov;Password=zenith")

# Redefine source 'A', by a connection string given in sections.
# This is for a GNU/Linux system without a unixODBC driver alias.
sqrlSource("A", "dbcname=planet;uid=zakharov;pwd=$tdwallet(planet)",
                "driver=/opt/teradata/client/16.10/lib64/tdata.so")

# Define a new source, by way of named connection-string components.
# This example is for a Windows-system client, and uses the '<pwd>'
# placeholder (it remains to set a value for the pwd parameter before
# connecting to the ODBC source).
sqrlSource("jumbo",
           driver = "PostgreSQL ANSI(x64)",
           server = "localhost",
           port = 5432,
           uid = "admin",
           pwd = "<pwd>")

# Define another source, as a vector of connection string
```

```
# components, along with some RODBC/SQRL parameter values.
sqrlSource("mydb", believeNRows = FALSE, autoclose = TRUE,
           connection = c(Driver = "SQLite3 ODBC Driver",
                          Database = "C:/mydatabase.db",
                          Timeout = 10000,
                          StepAPI = 1))

# Define another source, from a DSN (rather than a connection
# string), while also setting an RODBC/SQRL parameter value.
sqrlSource("ak", dsn = "Akademgorodok", as.is = TRUE)

# Redefine the source. This time, the dsn term is treated as a
# connection string component (because the server term must be).
sqrlSource("ak", dsn = "Akademgorodok", server = "Novosibirsk")

# Define another source, 'Huma', by a list of components, which
# includes an explicit source name, a vector of connection string
# components, a config list of RODBC/SQRL parameter values, and
# one more RODBC/SQRL parameter value outside of that config list.
# This list format is convenient for programmatic source creation.
sqrlSource(list(name = "Huma",
                connection = c(DRIVER = "Firebird/InterBase(r) driver",
                               DBNAME = "C:\\Database\\myDB.fdb",
                               UID = "MCSSITE",
                               PWD = "mcssite"),
                config = list(as.is = TRUE, scdo = FALSE),
                visible = TRUE))

# Define a source called 'source'. Although the name 'source' clashes
# with that of the base::source function, this definition is allowed
# because we simultaneously set an interface, 'S', that does not clash.
# Note the mixture of named and unnamed connection string components.
sqrlSource(source = list(connection = c("DSN=Source", UID = "me"),
                         interface = "S"))

# Another source, defined by a list of named connection string
# components (without setting any other parameter values).
sqrlSource(sf = list(driver = "{SnowflakeDSIIDriver}",
                     server = "xyz.eu-central-1.snowflakecomputing.com",
                     uid = "me", pwd = "guess", port = 443))

# Create a configuration file, and define a new source, 'sage',
# from it. Compare with the very first example, and with the 'mydb'
# example (both above). Once prepared, using a file is simple.
config.file <- tempfile()
writeLines(c("connection = c(driver   = 'SQLite3 ODBC Driver',",
             "               database = 'D:/mydb.db')",
             "autoclose = TRUE; believeNRows = TRUE"),
           config.file)
sqrlSource(sage = config.file)

# Create a (partial) configuration file (defining values for
# communications settings, but not the data source itself).
```

```
writeLines(c("autoclose = TRUE",
             "readOnlyOptimize = TRUE",
             "visible = FALSE"),
           config.file)

# Define a new source, 'papango', as a copy of the existing source,
# 'Huma', then apply the above configuration file over that, and
# then set values for the dsn and interface parameters over those.
# In this example, the value of the 'visible' parameter inherited
# from Huma is overwritten by the value from the config file.
# Setting the dsn parameter likewise erases the value of the
# connection parameter inherited from Huma, while the inherited
# uid and pwd values survive. Alternatively, we could have set
# 'connection = "dsn=Aythya"' in place of 'dsn = "Aythya"', and
# this would ignore the inherited uid and pwd values.
sqrlSource("papango",
           copy = "Huma",
           config = config.file,
           dsn = "Aythya",
           interface = "P")

# Review all defined sources.
sqrlSources()

# Review the configuration of the papango source.
P("config")

# Review details of how we connect to the source.
P("source")

# Clean-up (various methods of source removal).
sqrlSource("remove", "jumbo")
sqrlSource(remove = c("mongo", "papango"))
sqrlSource(Huma = NULL)
unlink(config.file)
```

`sqrlSources` *List Data Sources and their Interfaces*

Description

Returns a summary of defined data sources. These will consist of system and user DSNs, plus any additional sources defined via `sqrlSource`.

Usage

```
sqrlSources(...)
```

Arguments

...	An optional character string. If set to one of "all", "user", or "system", then a call is made to RODBC::odbcDataSources (with the corresponding *type* value) to re-examine that class of data source names (DSNs) and import all those found. If set to "remove", then all currently defined sources are deregistered.

Value

Returns a data frame of data source details.

Note

The return frame may have zero rows, if no data sources are defined.

Sources need only to have been defined; they need not actually exist.

DSNs with "Access", "dBASE", or "Excel" in their names are not automatically imported. They can be manually added via sqrlSource.

See Also

sqrlInterface, sqrlSource

Examples

```
# Review defined sources.
sqrlSources()

## Not run:
# Sample sqrlSources() output:

   name interface open                     driver
1 chaos     chaos    N       PostgreSQL ANSI(x64)
2 order      <NA>    N MySQL ODBC 5.3 ANSI Driver

# Here, there are two data sources; 'order' and 'chaos'.
# The interface to 'chaos' is a function of the same name.
# No interface has yet been defined for 'order' (use of
# that name is prevented due to its conflicting with the
# base::order function). Neither source (channel) is open.

## End(Not run)

# Remove all sources.
sqrlSources("remove")

# Reload user DSNs.
sqrlSources("user")
```

sqrlUsage *How to Use the Interface Functions*

Description

This material does not describe a single function, but (rather) how to use **SQRL** interfaces (once created). These interface functions do not have their own static help files, since their names are not known at build time.

Details

Once you have a named interface (created either automatically, on loading of the **SQRL** namespace, or manually, via `sqrlSource`), it can be used to communicate with the associated data source. Connection handles and communication parameters are managed under the hood.

The following sections provide usage examples for an interface called `owl`. The names of your own interface functions can be discovered by calling `sqrlSources()`.

Opening and Closing

```
# Open a connection to the data source.
owl()

# Alternative method (explicit form).
owl("open")

# This is fine (the channel survives).
rm(list = ls(all.names = TRUE))

# Check if the connection is open.
owl("isopen")

# Open a connection and confirm status.
owl()$isopen

# Close the connection.
owl("close")

# Close the connection when not in use.
owl(autoclose = TRUE)
```

Opening connections in the above way isn't usually necessary, since this occurs automatically, as and when required.

When necessary, the `isopen` command 'pings' the data source, to reliably establish whether or not the connection really is open (including after a network outage or remote closure).

With `autoclose = TRUE`, `owl("isopen")` should always return `FALSE`, whereas `owl()$isopen` will often return `TRUE`. This is because the latter command attempts to open a connection, with its return value being the connectivity status immediately after that attempt (before *autoclose* takes effect

and closes the connection). This provides a test of data source reachability and responsiveness, regardless of the *autoclose* setting.

Submitting Queries

```
# Submit a query.
owl("select 1")

# Submit another query.
owl("select '", sample(letters, 1), "' from dual")

# Submit a multi-statement query.
owl("use necronomicon; select top ",
    sample(6, 1), " shoggoths from pit")

# Filtering against a set of values.
owl("select columnA from database.table ",
    "where columnB in (", c(2, 3, 5), ")")

# Supplying a query as a list of components.
owl(list("select ", c("'red'", "'blue'")))

# A parameterised query (SQL with R in it).
owl("select <R> a + b </R>", a = 0, b = 1)

# Explicit form.
owl(query = list("select ", "<R>sin(x)</R>"), x = 0)

# Verbatim query.
owl(verbatim = "select 1 as N")

# Recall the result of the last successful query.
owl("result")
```

If necessary, a connection channel will be opened automatically. The connection will remain open afterwards, unless *autoclose* is TRUE.

When a query is supplied as components, the pieces are pasted together without any intervening whitespace. To facilitate their use with the SQL in operator, any atomic vectors are collapsed to comma-separated values, beforehand. This (default) behaviour can be altered with the *aCollapse* and *lCollapse* parameters (as described in sqrlParams).

Using the *query* keyword overrides the order of precedence (as detailed below). Whereas *query* arguments go through the usual **SQRL** concatenation, parsing, and R-substitution process, *verbatim* arguments are submitted directly and without alteration. Consequently, the *verbatim* command accepts only a single character string, which cannot be parameterised via embedded R.

As **SQRL** aims to be flexible on input formatting, the above examples can be extrapolated. For instance, the explicit *query* could have been a single string.

When a query returns no data (as would "use database"), the interface function returns invisibly.

If a query should fail due to an unexpectedly lost connection, one further attempt will be made to reconnect and resubmit (provided the *retry* parameter is TRUE). Unless user input is required for authentication, this should go unnoticed. If temp tables were in use, then these will have been dropped along with the original connection, and an error may still occur.

Submitting Queries from File

```
# Submit a query from file.
owl("my/file.sql")

# Equivalent alternative forms.
owl("my/", "file.sql")
owl(c("my/", "file.sql"))

# Explicit alternative forms.
owl(file = "my/file.sql")
owl(file = c("my/", "file.sql"))
```

Using the *file* keyword overrides the order of precedence (as detailed below). In its absence, unnamed arguments are treated as a file path when they point to a readable file.

In the above examples, `list(...)` works just as well as `c(...)`. Either way, the path components are pasted together without any intervening whitespace (the path not being a literal query).

Submitting Parameterised Queries from File

```
# Submit a parameterised query from file.
owl("my/file.sqrl", day = 1, month = "May")

# Supplying the arguments in a list.
owl("my/file.sqrl", list(day = 1, month = "May"))

# Supplying the arguments in an explicitly named list.
owl("my/file.sqrl", args = list(day = 1, month = "May"))

# Supplying both the query and its arguments in a list.
owl(list(file = "my/file.sqrl", day = 1, month = "May"))
owl(list(file = c("my/", "file.sqrl"),
         args = list(day = 1, month = "May")))
```

To be clear, the phrase "parameterised query" is not meant in the sense of prepared or parameterised statements (as per package **RODBCext**). In **SQRL**, parameter substitution occurs within R (locally), with the resulting string being passed to the ODBC driver as an ordinary query. Refer to `sqrlScript` for the details.

The use of the *args* keyword is optional when all list members have syntactically valid names (in the sense of `base::make.names`). Any such lists are automatically interpreted as collections of named arguments (and are unpacked to those collections). Query arguments called, for instance, *file*, *proc*, or *query*, may need to be wrapped in *args* to ensure they are treated as intended, and not as query (script) specifiers.

In keeping with **SQRL**'s intended flexibility around input formatting, any of the file path specification methods of the previous section could also be used here.

Stored Procedures

```
# Import procedures from file.
owl(library = "my/library.sqrl")

# List procedure names.
owl("library")

# List procedure definitions.
owl("Library")

# Run a named procedure.
owl("my procedure")

# Equivalent explicit form.
owl(proc = "my procedure")

# Run a parameterised procedure.
owl("Cropp River Rainfall", date = Sys.Date() - 1)

# Empty the library.
owl(library = NULL)
```

As detailed below, procedures top the order of precedence. Consequently, the *proc* keyword is an entirely optional transparency device. In its absence, unnamed arguments are treated as the name of a procedure when they name a procedure within the library.

The library-file procedure-definition format is described in sqrlScript. The path to such a file (i.e., the value of the *library* keyword) can be supplied in any of the file-path formats of the previous sections (that is, as a list or vector of components).

Any of the file-path and/or query-argument specification formats seen in the previous two sections can equally be used with procedure names. The only difference is to replace any *file* keyword with the *proc* keyword.

Querying Metadata

```
# List all tables.
owl("tables")

# List all tables within a database (schema).
owl("tables", "mydatabase")

# Get information on the columns of a particular table.
owl("columns", "mydatabase.table")

# Get information on the primary keys of a particular table.
owl("primarykeys mydatabase.table")

# Get information on source data types.
owl("typeinfo")
```

The tables, columns, primarykeys and typeinfo commands are simple (reduced functionality) wrappers about **RODBC**'s sqlTables, sqlColumns, sqlPrimaryKeys, and sqlTypeInfo, respectively. These features are dependent upon the support of your DBMS and driver. For some sources, the *believeNRows* parameter may need to be FALSE. Metadata queries bypass SQRL's parser.

Reviewing Settings

```
# Get the associated source definition.
owl("source")

# Get the value of one named parameter.
owl("uid")

# Alternative method (pings the source).
owl()$uid

# List the values of all parameters.
owl("config")

# List a subset of parameter values.
owl("settings")
```

The settings subset is intended for restoring **RODBC** and/or **SQRL** parameter values at the end of a script that changed some. An example of this is given in sqrlScript. It can also be used to transfer parameter values between sources.

Setting Parameters

```
# Set a parameter value.
owl(stringsAsFactors = FALSE)

# Set multiple parameter values.
owl(max = 1000, na.strings = c("NA", "-", ""))

# Set multiple values from a list.
owl(list(case = "toupper", scdo = FALSE))

# Set values from a list (explicit form).
owl(config = list(visible = TRUE, autoclose = TRUE))

# Import values from source 'wol'.
owl(config = wol("settings"))

# Import values from a configuration file.
owl(config = "my/config/file.txt"))

# Import one value from a file.
owl(pwd = c("path/", "to/", "file", ".txt"))

# Reset parameters to their default values.
```

```
owl(reset = c("errors", "nullstring"))
```

The *driver* and *dsn* parameters accept file paths as their values. For all other parameters, values are extracted from within any specified files.

Assigning *visible* TRUE authorises modification of the global prompt option. When running 'R.exe', 'Rterm.exe' or 'Rgui.exe' on a Windows operating system, this also authorises modification of theR window title.

Calling reset on its own, as in owl("reset"), does nothing.

Further alternative input formats appear in the examples section of sqrlParams.

Changing the Interface

```
# Change the interface.
owl(interface = "O")

# Change it back.
O(interface = "owl")
```

Changing the interface is just a particular case of setting a parameter.

If the proposed new interface name already belongs to some other function within the R search path, then the change request will be denied (unless that name is remove, in which case the current interface function will be deleted).

A successful change deletes the previous interface.

Listing Data Sources

```
# See the data sources and their interfaces.
owl("sources")
```

This is equivalent to calling sqrlSources().

Getting Help

```
# Get help on 'owl' (alternative forms).
owl("help")
owl("?")

# Obtain help, in specific formats.
owl("help text")
owl("help html")
```

The above calls will attempt to provide help tailored for the specific interface, and will fall back to these notes (help(sqrlUsage) or ?sqrlUsage) should that fail.

Removing the Source

```
# Deregister the associated source.
owl("remove")
```

This closes any open connection to the data source, deletes the interface function (owl), and deregisters the source from **SQRL**.

Order of Precedence

When unnamed arguments are supplied, such as in `owl("something")`, **SQRL** interprets those arguments with the following order of precedence:

1. Procedure names (in the *library*),
2. File paths (of SQL or SQRL scripts),
3. Special words ("close", "config", etc.),
4. Parameter names (optionally followed by values),
5. Literal SQRL script (including pure SQL).

Hence, if a file called (say) '`use database`' should exist, then `owl("use database")` submits the content of that file (rather than the apparent SQL command). Such conflicts can be resolved by assigning the unnamed arguments to the appropriate keyword (*file*, *proc*, or *query*). In this case, the new command would be `owl(query = "use database")`.

See Also

`sqrlAll`, `sqrlConfig`, `sqrlParams`, `sqrlScript`

Examples

```
# Some default values, for demonstration purposes.
x <- 1; y <- 2

# Define a new SQRL source and its interface. The DSN need
# not exist, but this will fail (by design) if a function
# called 'owl' already exists on the R search path.
sqrlSource("owl", dsn = "HundredAcreWood")

# This (ordinarily pointless) SQRL script contains no SQL,
# which allows it to run even when the DSN does not exist
# (there being no need to open a connection). In this case,
# the x and y variables are implicitly inherited.
owl("<R> x * y")

# Explicitly assign x, while inheriting y.
owl("<R> x * y", x = 2)

# An alternative arrangement of arguments.
owl(x = 3, query = "<R> x * y", y = 1)

# Write the script to file.
myfile <- tempfile()
writeLines("<R> x * y", myfile)

# Run the script from file, inheriting variables.
owl(myfile)

# Run the script from file, with explicit arguments.
# (These are all equivalent.)
```

```
owl(myfile, x = 2, y = 3)
owl(myfile, list(x = 2, y = 3))
owl(myfile, args = list(x = 2, y = 3))
owl(file = myfile, x = 2, y = 3)
owl(file = myfile, list(x = 2, y = 3))
owl(file = myfile, args = list(x = 2, y = 3))
owl(list(file = myfile, x = 2, y = 3))
owl(list(file = myfile, args = list(x = 2, y = 3)))

# With the file path specified as components.
owl(dirname(myfile), "/", basename(myfile), x = 2, y = 3)
owl(file = c(dirname(myfile), "/", basename(myfile)),
    args = list(x = 2, y = 3))

# Construct a library file (of procedure definitions).
mylibraryfile <- tempfile()
writeLines(c("<proc 'proc-a'> <R> x * y </proc>",
             "<proc 'proc-b'> <R> x + y </proc>"),
           mylibraryfile)

# Import procedures from file (to owl's library).
owl(library = mylibraryfile)

# Run the imported procedures.
owl("proc-a")
owl("proc-b", x = 2, y = c(3, 4))

# Review the last result.
owl("result")

# Clean-up.
unlink(c(myfile, mylibraryfile))
owl("remove")
```

Index