SQRL Features and Usage

Mike Lee

September 21, 2022

Whatever your nominal role, if it comes with access to an organisation's database, then the occasional explore-pull-summarise request is almost inevitable. The aim of SQRL is to turn those around as quickly as possible, letting you get back to the fun stuff.

SQRL handles (RODBC) communications in the background; silently opening connections and applying preferences, without bothering the user. If you've been handed a multi-statement SQL script, SQRL can run that original file. Blocks of R can be added to the same script; in order to parameterise a query, or transform the resulting data. SQRL also provides means for living with DBMS limitations, administrator's restrictions, and unreliable networks.

1 Quick Start

Upon loading, SQRL generates a dedicated interface function to each ODBC DSN found on your system. Supposing one of those DSNs is called 'arx', it could then be queried like so;

```
> library(SQRL)
> arx("use database; select * from table")
```

Arguments fed to `arx(...)` are parsed and interpreted. In the example above, the single argument looks like a pair of SQL statements. That being the case, a connection is opened, the first statement is submitted, then the second, and the result of that query returned. The connection is left open for reuse. All communications with the ODBC drivers are made through calls to RODBC functions.

There is a computational overhead to the parsing, of course, but it is negligible in comparison to the time it takes to process a query. In return, less time, effort, and typing is required from the user. For instance, if 'query.sql' should happen to be the path of a readable file, then

```
> arx("query.sql")
```

submits the statements appearing within it. By and large, SQRL attempts to be flexible; supporting different and minimalist styles of input, and deducing what you mean it to do. For the order of precedence, see

```
> help(sqrlUsage)
```

2 Multi-Statement Scripts

If this

```
1   /* This script has three statements. */
2   use database;
3   insert into table1
4     select * from table2;
5   drop table table2;
```

is the content of file 'sequence.sql', then

```
> arx("sequence.sql")
```

submits each statement in turn, and returns the result of the final operation. This is convenient when you've inherited a script prepared in a SQL IDE.

While the parser does handle basic procedural language (PL), it may sometimes trip over more esoteric DBMS-specific extensions. Should that occur, the manual details a robust-parsing fallback mode (requiring tweaks to the original script), and an unparsed verbatim alternative (for single statements, only).

3 Parameterisaton of Queries

SQRL's parser supports a hybrid syntax wherein arbitrary blocks of R can be embedded within SQL script. Special tags mark the beginning and end of any such R inclusions, in much the same way as Sweave isolates R within LaTeX. Here is a listing of one such script;

```
1   -- SQL with R in it.
2   select * from database.table
3   where column > <R> pnorm(x) </R>
```

If this is the content of a file 'query.sqrl', then it could be submitted like so;

```
> arx("query.sqrl", x = 3)
```

While hybrid scripts are parameterised at the user's end, the data source receives only standard unparameterised SQL; after all R blocks have been replaced by the results of their evaluation. This provides an alternative to constructing queries from collections of strings and variables pasted together within the body of an R script, and allows your text editor to apply SQL syntax highlighting rules to predominantly SQL scripts. It is also useful when you don't have the permissions to create parameterised queries, macros, or functions on the database server itself.

An extended discussion of the hybrid syntax, with examples, is provided in the reference manual and help system;

```
> help(sqrlScript)
```

4 Integrated Post-Processing

For any number of reasons, SQL-queried data is frequently cleansed, converted, or otherwise transformed in R, prior to reaching its destination as input to a model or analysis. SQRL's parser allows any such processing to be tacked onto the end of a SQL script;

```
/* A SQL query followed by R post-processing. */
select A, B, C from table

-- Submit the query, assign its result to 'x', and switch to R.
<result -> x>

# Manipulate the result in R.
x$A <- as.factor(x$A)
x$B <- as.Date(x$B, "%d/%m/%Y")
x$C <- (x$C - mean(x$C)) / sd(x$C)
x -- Only this final value is returned.
```

If this is in file 'pull.sqrl', then;

```
> arx("pull.sqrl")
```

returns the data already cleaned and formatted. This helps separate data-preparation script from data-analysis script. Alternatively, simple analyses can be placed entirely within the post-processing section.

5 Set and Forget Communications

Whether opening a connection, submitting a query, or fetching results, communicating with ODBC drivers involves passing the values of several control parameters every time. Since this might be bothersome when one or more of those values is non-default, SQRL keeps a record of the settings for each data source, and applies them behind the scenes. For example,

```
> arx(believeNRows = FALSE, dec = ",")
```

sets new values for two of the parameters. All subsequent queries to that source,

```
> arx("select * from table")
```

for instance, inherit those values. Any other sources use their own values.

Some communications settings cannot be changed while the connection to the data source is open. Their values are applied at the time of opening, rather than with each query, and so can only be altered beforehand.

Some features of SQRL work best when opening a new connection does not involve manual authentication. Usernames and passwords can be stored for automatic use on data sources where the user would otherwise be prompted for input. Passwords are held with a very modest level of security.

For a complete list of control parameters, see

```
> help(sqrlParams)
```

6 Protected Connection Handles

SQRL interface functions, communications settings, and connection handles are stored outside of the global environment. That being the case,

```
> rm(list = ls(all = TRUE))
```

is fine; settings are preserved, connections remain open, temp tables persist, and you shouldn't see any warnings about the closing of unused handles.

7 Automatic Connection Closure

By default, connections remain open after use. To manually close one, use

```
> arx("close")
```

This is allowed even when the connection is not open (is already closed). A new connection will be opened the next time a query is submitted. Communications settings can be changed in the meanwhile.

To have the connection close automatically after each sequence of operations, set

```
> arx(autoclose = TRUE)
```

With that done,

```
> arx("use database; select * from table")
```

opens a connection, submits the first statement, then the query, closes the connection, and returns the result of the query. This can be courteous when your administrator has set a short timeout.

8 Recovery from Lost Connections

If a query should happen to fail, and that failure appears to have been caused by an unexpectedly dropped connection, then a single attempt will be made to reconnect and resubmit. This occurs automatically, and will usually go unnoticed (unless manual input is required for authentication, see section 5). This can be helpful when working over an unreliable network, or when your administrator has set a short timeout.

Suspect connections are tested with a simple and trusted 'ping' query. This is automatically determined the first time a connection is opened. It can be manually redefined if that initial value proves unsatisfactory;

```
> arx(ping = "begin null; end;")
```

Note that any temp tables will not have survived the initial connection loss. Even though a connection is reestablished, the query will subsequently fail if it uses one of those temps. Automatic recovery attempts can be disabled with

```
> arx(retry = FALSE)
```

9 Local Exceptions from Remote Failures

When a query fails in an unrecoverable way, the corresponding driver and RODBC messages are raised to an error (instead of being returned as a normal character vector). Consequently, exceptions on the remote server immediately halt local execution in R. This stops the script at the point of failure, before knock-on effects cause other, potentially cryptic, problems.

To disable this behaviour, and revert to character notifications, use

```
> arx(errors = FALSE)
```

10 Visible Connection Status

For a continuous display of connection status, use

```
> arx(visible = TRUE)
```

This authorises changes to R's global 'prompt' option. If a connection to arx is open, the prompt should now read

```
a>
```

This can be of use when juggling multiple sources. If we have three, 'arx', 'box' and 'crate', we can make them all visible with

```
> sqrlAll(visible = TRUE)
```

If the prompt then reads

```
ac>
```

we can see that arx and crate and are open, while box is closed.

On Windows (only), in R.exe, Rterm.exe, or Rgui.exe (MDI or SDI, but not in RStudio), the window title also changes. While a connection is open, it might read something like

```
R Console (64-bit) (arx)
```

While a query is running, this becomes

```
R Console (64-bit) (arx)*
```

and while results are being fetched,

```
R Console (64-bit) (arx)+
```

This distinguishes server query time from network data transfer time, and is potentially helpful in diagnosing a bottleneck.

To change the prompt and title indicators, use

```
> arx(prompt = "A", wintitle = "[ARX]")
```

11 Result Retention

If you've just done this;

```
> arx("my_long_query.sql")
```

and forgot to assign the output, fear not. The result of the last successful job is stored as

```
> arx("result")
```

When memory is low, the stored result can be deleted with

```
> arx(result = NULL)
```

12 Automatic Concatenation

Queries and file paths may be expressed as components. These are automatically pasted into a single character string. By default, list elements are collapsed together with empty strings, while atomic vectors are collapsed with commas. For example, if we define two atomic vectors;

```
> a <- c("columnA", "columnB")
> b <- c(1, 2, 3)
```

then

```
> arx("select ", a, " from table",
+ " where columnC in (", b, ");")
```

results in

```
"select columnA,columnB from table where columnC in (1,2,3);"
```

being submitted to the data source.

Note that if `b` were a character vector, it might still be necessary to put single quotes about each of its elements;

```
> sq <- function(x) paste0("'", x, "'")
> arx("select * from table where column in (", sq(b), ")")
```

This is difficult to automate, because the elements might already be quoted, or quotes might not be wanted. The latter occurs when the vector specifies, say, column names (as is the case with `a`, above), or when it holds character representations of long integers.

The collapsing rules can be changed by

```
> arx(aCollapse = ", ", lCollapse = "\n")
```

13 A Longer Script, using Feedback

This script;

```
1   -- Use these communications settings.
2   <with>
3     as.is = TRUE
4   </with>
5
6   -- Define an R function, for later use.
7   <R>
8     asdate <- function(d) {format(d, "'%Y-%m-%d'")}
9   <do>
10
11  -- A parameterised SQL query (template).
12  select columnA from table1
13    where columnB = <R> asdate(date) </R>
14
15  -- Submit the above query, assign its result to 'x',
16  -- and transform that data in R.
17  <result -> x>
18    x <- unique(x$columnA %/% 10000)
19  <do>
20
21  -- Submit another query, incorporating the
22  -- (transformed) intermediate results.
23  select * from table2
24    where columnC in (<R> x </R>)
```

in file 'script.sqrl', could be run with

```
> arx("script.sqrl", date = Sys.Date() - 1)
```

The first section, lines 1 through 4, forces the script to run with the specified communications settings. Original settings will be restored afterwards. The second section, lines 6 through 9, evaluates R script into a temporary working environment. The global environment is unaffected. The third section, lines 11 through 19, constructs a query from an argument, `date`, assigns the result of the query to `x` (in the temporary environment), and manipulates that result. The final section, lines 21 through 24, constructs a second query from the (transformed) intermediate result, and returns the result of that query.

This example is simple enough it could have been done with a join, but circumstances might be that the intermediate data manipulation is too complex for SQL, or involves more data from R, or from another data source, or you don't have enough server memory quota, or permissions for a temp table, or can't nest queries, or can't upload data, or want to reduce load on the server (at the expense of increased network traffic), or are fighting an unfamiliar DBMS, or some other administrative restriction. More advanced features of the hybrid syntax allow for conditional submissions, loops, early returns, and stored procedures.

```
> help(sqrlScript)
```

14 Verbose Mode

For debugging, verbose mode is enabled by

```
> arx(verbose = TRUE)
```

15 Data Sources

To see what sources are available, call

```
> sqrlSources()
```

New sources can be defined by specifying a DSN, or connection string, or from a configuration file containing the string and communications settings;

```
> sqrlSource(foo = "config.txt")
```

Detailed examples are provided in the manual;

```
> help(sqrlSource)
> help(sqrlConfig)
```

16 Help

Being user and system dependent, the names of the interface functions, like `arx`, are unknown at package build time. Consequently, they do not appear within the SQRL reference manual or the standard R help system. For examples of general usage, try

```
> help(sqrlUsage)
```

Alternatively,

```
> arx("help")
```

provides a smaller set of examples, along with configuration details specific to interface `arx`.

For a list of all communications settings, use

```
> arx("config")
```

and for one specific setting,

```
> arx("believeNRows")
```

(for instance).

For the current connection status (open or closed), try

```
> arx("is open")
```

This may ping the source for confirmation. The space is optional.

17 Ending Sessions

To close the connection to data source 'arx', use

```
> arx("close")
```

To close the connections to all SQRL data sources, use

```
> sqrlAll("close")
```

To close the connections to all SQRL data sources, remove their interface functions, and unload the SQRL library, use

```
> sqrlOff()
```

After this, no further communication is possible with any data source (via SQRL, until it is reloaded).