Interacting with Databases from R and Shiny

Barbara Borges Ribeiro

WSDS 2017

Content

PART I

- DB best practices in *R*
  - `DBI` standardizes how to interact with a DB
  - `odbc` is a DBI backend for any DB with an ODBC driver
  - use `dplyr` syntax to talk to a DB
  - use `pool` (my package) to connect to a DB from Shiny

PART II

- Shiny app demo: a CRUD app using `DBI`, `RSQLite`, `dplyr`, and `pool`
- Connect to a *SQLite* database from Shiny
- **C**reate, **R**ead, **U**pdate and **D**elete data from database
- See updated information using `reactivePoll`

Databases, and its many flavors

- **Relational databases** $<-$ *We'll focus on these*
- **NoSQL/object oriented databases**

Databases, and its many flavors

- **Relational**-ish **databases**
  - RDBMSs (relational database management systems) store data in columns and rows, which in turn make up tables. A table in RDBMS is like a spreadsheet.
  - Use *SQL*.
  - MySQL, PostgreSQL, SQLite
  - Apache Hive and Cloudera Impala for distributed systems (relational-like)
  - *R* packages: `DBI`, `odbc`, `dplyr` (and `dbplyr`), `pool`

Databases, and its many flavors

- **NoSQL/object oriented databases**
  - These do not follow the table/row/column approach of RDBMS. Good for working with large amounts of data that do not require structure. Less concerned with storing them in ordered tables than they are with simply making them available for fast access.
  - MongoDB, CouchDB, HBase, Cassandra

DBI (theory)

- `DBI` defines the generic **D**ata**B**ase **I**nterface for R. The idea is to standardize how to interact with a database from *R* (connect, disconnect, read, write and mutate data safely from *R*).
- The connection to individual DBMS is provided by other packages that import `DBI` (`DBI`-compliant backends) and implement the methods for the generics defined in `DBI`.
- Current goal: ensure maximum portability and exchangeability and reduce the effort for implementing a new DBI backend (through the `DBItest` package and the DBI specification)

DBI (practice)

```
con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")

DBI::dbWriteTable(con, "iris", iris)

DBI::dbGetQuery(con, "SELECT count() FROM iris")
#>   count()
#> 1     150

DBI::dbDisconnect(con)
```

DBI (practice) – *SQL* injections edition!

```
sql <- "SELECT * FROM X WHERE name = ?name"
sqlInterpolate(ANSI(), sql, name = "Hadley")
#> <SQL> SELECT * FROM X WHERE name = 'Hadley'

# This is safe because the single quote has been double es
sqlInterpolate(ANSI(), sql, name = "H'); DROP TABLE--;")
#> <SQL> SELECT * FROM X WHERE name = 'H''); DROP TABLE--;
```

DBI (practice) – *SQL* injections edition!

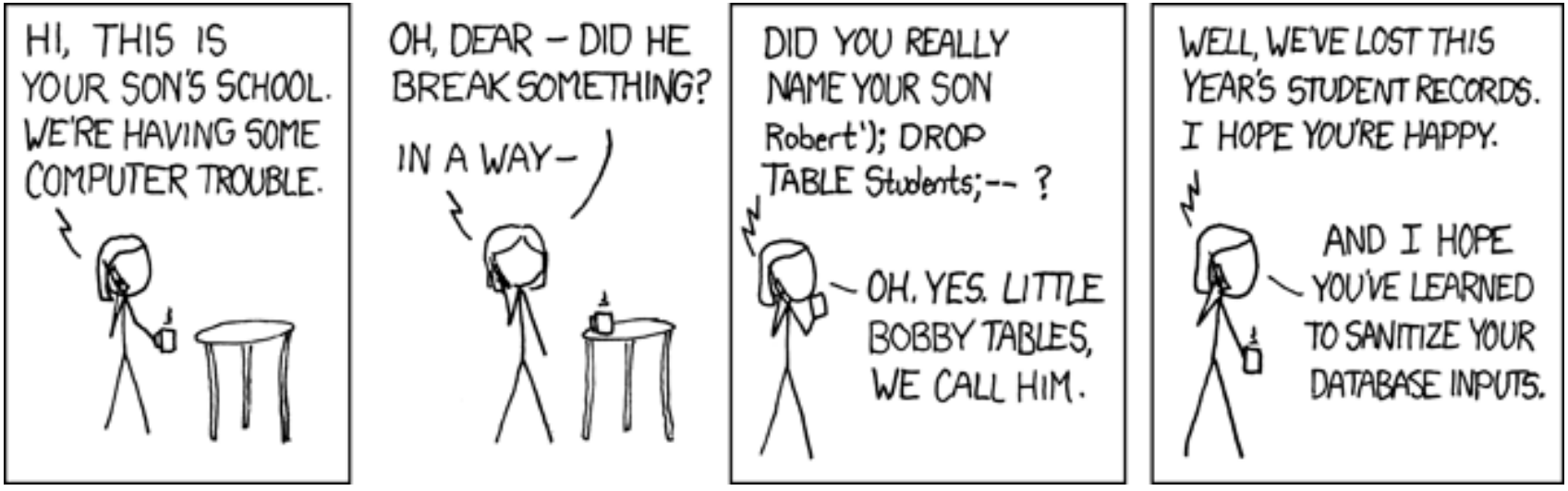

Figure 1: bobby-tables

odbc (theory)

- ODBC (Open Database Connectivity) is a specification for a database API. This API is independent of any one DBMS, operating system or programming language. The functions in the ODBC API are implemented by developers of DBMS-specific drivers. (source)
- The `odbc` package provides a DBI compliant backend for any database with an ODBC driver (although anyone can write a driver, most of these tend to be paid, enterprise products). This allows for an efficient, easy to setup connection to any database with ODBC drivers available (RStudio Server Pro will soon bundle several of these drivers including Microsoft SQL Server, Oracle, MySQL, PostgreSQL, SQLite, Cloudera Impala, Apache Hive and others).

odbc (practice)

```
con <-  DBI::dbConnect(odbc::odbc(),
  Driver = "{postgresql}",
  Server = "postgresdemo.cfd8mtk93q6a.us-west-2.rds.amazona
  Port = 5432,
  Database = "postgresdemo",
  UID = "guest",
  PWD = "guest"
)

DBI::dbGetQuery(con, "SELECT * FROM city LIMIT 2;")
#>   id     name countrycode district population
#> 1  1    Kabul         AFG    Kabol    1780000
#> 2  2 Qandahar         AFG Qandahar     237500

DBI::dbDisconnect(con)
```

dplyr (theory)

- **Idea**: use `dplyr` syntax to talk to databases (no *SQL* involved for the end user).
- `dplyr` (and the brand-new `dbplyr`) wrap and extend a lot of `DBI` methods, so that you can use `dplyr` + *R* directly to interact with your database (instead of `DBI` + *SQL*, which is what `dplyr` does for you)
- With the recent revamp, you can a LOT with `dplyr` (reading and transforming data, writing tables, querying the database), but `DBI` still allows you to do more (the price you pay. . .)
- **Bottom line**: Especially if you're already familiar with the `dplyr` verbs (mainly, `filter()`, `select()`, `mutate()`, `group_by()`, and `summarise()`), using `dplyr` to interact with databases is a great idea.

dplyr (practice)

```
library(dplyr)
con <- DBI::dbConnect(RMySQL::MySQL(),
  dbname = "shinydemo",
  host = "shiny-demo.csa7qlmguqrf.us-east-1.rds.amazonaws.c
  username = "guest",
  password = "guest"
)
con %>% tbl("City") %>% head(2)
#> # Source:   lazy query [?? x 5]
#> # Database: mysql 5.5.5-10.0.17-MariaDB
#> #   [guest@shiny-demo (...) amazonaws.com:/shinydemo]
#>      ID     Name CountryCode District Population
#>   <dbl>    <chr>       <chr>    <chr>      <dbl>
#> 1     1    Kabul         AFG    Kabol    1780000
#> 2     2 Qandahar         AFG Qandahar     237500

DBI::dbDisconnect(con)
```

pool (theory)

- **Problem**: how to interact with a database from Shiny?
  - Per session, there is only a single R process and potentially multiple users.
  - Also, establishing connections takes time and they can go down at any time.
  - So, you don't want a fresh connection every for every user action (because that's slow), and you don't want one connection per app (because that's unreliable)...
- The `pool` package allows you to manage a shared pool of connections for your app, giving you both speed (good performance) and reliability (connection management).

pool (theory)

- `pool` is mainly important when in a Shiny app (or another interactive app with an R backend), but it can be used in other situations with no problem.
- `pool` integrates seamlessly with both `DBI` and `dplyr` (the only noticeable differences are in the create/connect and close/disconnect functions).
- Will be released to CRAN very soon!

pool (practice)

```
pool <- pool::dbPool(RMySQL::MySQL(),
  dbname = "shinydemo",
  host = "shiny-demo.csa7qlmguqrf.us-east-1.rds.amazonaws.c
  username = "guest",
  password = "guest"
)
pool %>% tbl("City") %>% head(2)
#> # Source:   lazy query [?? x 5]
#> # Database: mysql 5.5.5-10.0.17-MariaDB
#> #   [guest@shiny-demo (...) amazonaws.com:/shinydemo]
#>      ID     Name CountryCode District Population
#>   <dbl>    <chr>       <chr>    <chr>      <dbl>
#> 1     1    Kabul         AFG    Kabol    1780000
#> 2     2 Qandahar         AFG Qandahar     237500

pool::poolClose(pool)
```

Resources

- All packages mentioned here are open-source and available on Github
- `DBI`, `dplyr`: https://db.rstudio.com/
- `pool` (and general DB + `shiny`): http://shiny.rstudio.com/articles/ (Databases section)

- https://blog.rstudio.org/
- These slides + app source code: https://github.com/bborgesr/useR2017

Shiny app

Shiny app: skeleton

```
library(shiny)
library(shinydashboard)
library(dplyr)
library(pool)

pool <- dbPool(RSQLite::SQLite(), dbname = "db.sqlite")

tbls <- reactiveFileReader(500, NULL, "db.sqlite",
  function(x) db_list_tables(pool)
)

ui <- dashboardPage(...)
server <- function(input, output, session) {...}
shinyApp(ui, server)
```

Shiny app: create table (adapted!)

```
actionButton("create", "Create table"),    # ui
textInput("tableName", "Table name"),
numericInput("ncols", "Number of columns"),
uiOutput("cols")

output$cols <- renderUI({                          # server
  input$tableName
  cols <- vector("list", input$ncols)
  for (i in seq_len(input$ncols)) {
    textInput(paste0("colName", i), "Column name"),
    selectInput(paste0("colType", i), "Column type",
      c(Integer = "INT", Character = "VARCHAR"))
  }
  cols
})
```

Shiny app: create table (cont)

```
# finalCols is a list. E.g:
# list(ID = "INT", item = "VARCHAR", count = "INT")
observeEvent(input$create, {
  db_create_table(pool, input$tableName, finalCols)
})
```

Shiny app: read table (adapted!)

```
selectInput("tableName", "Table name", NULL), # ui
checkboxGroupInput("select", "Choose columns to read"),
selectInput("filter", "Choose column to filter on", NULL),
checkboxGroupInput("vals", "Choose values to include"),
tableOutput("res")

observeEvent(tbls(), {                                  # server
  updateSelectInput(session, "tableName", choices = tbls()
})
observe({
  cols <- db_query_fields(pool, input$tableName)
  updateCheckboxGroupInput(session, "select", choices = co
})
output$res <- renderTable({
  pool %>% tbl(input$tableName) %>% select(input$select) %
    filter(input$filter %in% input$vals)
})
```

Shiny app: in action!