Lecture 13: Debugging and Databases

STAT GR5206

Statistical Computing & Introduction to Data Science

Cynthia Rush
Columbia University

December 8, 2017

Course Notes

- Homework due December 11.
- Final Friday, December 15, 1:10pm - 4:00pm.
    - Section 002: MATH 417
    - Section 003: HAVEMEYER 309
    - Section 004: HAVEMEYER 309
    - Section 005: PUPIN 329

Databases: SQL and Querying

Databases vs. Dataframes

Database Jargon

- A **record** is a collection of **fields** (likes rows and columns).
- A **table** is a collection of records which all have the same fields with different values. These are like dataframes in `R`.
- A **database** is a collection of tables.

`R`'s dataframes are actually tables

`R` Jargon	Database Jargon
column	field
row	record
dataframe	table
types of the columns	table **schema**
bunch of related dataframes	database

So, Why Do We Need Database Software?

1. Size
   - R keeps its dataframes in memory
   - Industrial databases can be much bigger
   - Work with selected subsets
2. Speed
   - Clever people have worked very hard on getting just what you want fast
3. Concurrency
   - Many users accessing the same database simultaneously
   - Lots of potential for trouble (two users want to change the same record at once)

So, Why Do We Need Database Software?

- Databases live on a **server**, which manages them
- Users interact with the server through a **client** program
- Lets multiple users access the same database simultaneously
- **SQL** (**structured query language**) is the standard for database software
- Mostly about **queries**, which are like doing row/column selections on a dataframe in `R`

Connecting R to SQL

- SQL is its own language, independent of `R` (similar to regular expressions). But we're going to learn how to run SQL queries through `R`.
- First, install the packages `DBI`, `RSQLite`.
- Also, we need a database file: download the file `baseball.db` and save it in your working directory.

```
library(DBI)
library(RSQLite)
drv <- dbDriver("SQLite")
con <- dbConnect(drv, dbname="baseball.db")
```

- The object `con` is now a persistent connection to the database `baseball.db`.

Listing What's Available

```
dbListTables(con) # List tables in our database
dbListFields(con, "Batting") # Fields in Batting table
dbListFields(con, "Pitching") # Fields in Pitching table
```

Importing a Table as a Data Frame

```
batting <- dbReadTable(con, "Batting")
class(batting)
dim(batting)
```

- Can perform `R` operations on `batting`, since it's a data frame
- In lecture today, we'll use this route primarily to check our work in SQL; in general, want to do as much in SQL as possible, since it's more efficient and likely simpler

Check Yourself

Tasks

- Using `dbReadTable()`, grab the table named `Salaries` and save it as a data frame called `salaries`.
- Using the salaries data frame and `ddply()`, compute the payroll (total of salaries) for each team in the year 2010.
- Find the 3 teams with the highest payrolls, and the team with the lowest payroll.

SQL

SELECT

Main tool in the SQL language: `SELECT`, which allows you to perform queries on a particular table in a database. It has the form:

```
SELECT columns or computations
  FROM table
  WHERE condition
  GROUP BY columns
  HAVING condition
  ORDER BY column [ASC | DESC]
  LIMIT offset, count;
```

`WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, `LIMIT` are all optional

Pick out five columns from the table "Batting", and look at the first 10 rows:

```
dbGetQuery(con, paste("SELECT playerID, yearID, AB, H, HR",
                      "FROM Batting",
                      "LIMIT 10"))
```

```
batting[1:10, c("playerID", "yearID", "AB", "H", "HR")]
```

To reiterate: the previous call was simply to check our work, and we wouldn't actually want to do this on a large database, since it's much more inefficient to first read data into an `R` data frame, and then call `R` commands

ORDER BY

- We can use the ORDER BY option in SELECT to specify an ordering for the rows
- Default is ascending order; add DESC for descending

```
dbGetQuery(con, paste("SELECT playerID, yearID, AB, H, HR",
                      "FROM Batting",
                      "ORDER BY HR DESC",
                      "LIMIT 5"))
```

Check Yourself

Tasks

Run the following queries and determine what they're doing. Write `R` code to do the same thing on the `batting` data frame.

```
dbGetQuery(con, paste("SELECT playerID, yearID, AB, H, HR",
                      "FROM Batting",
                      "WHERE yearID >= 1990
                          AND yearID <= 2000",
                      "ORDER BY HR DESC",
                      "LIMIT 5"))
dbGetQuery(con, paste("SELECT playerID, yearID, MAX(HR)",
                      "FROM Batting"))
```

Databases: SQL Computations

Databases vs. Dataframes

R's dataframes are actually tables

R Jargon	Database Jargon
column	field
row	record
dataframe	table
types of the columns	table **schema**
collection of related dataframes	database
conditional indexing	`SELECT`, `FROM`, `WHERE`, `HAVING`
`d*ply()`	`GROUP BY`
`order()`	`ORDER BY`

SQL

SELECT

Main tool in the SQL language: `SELECT`, which allows you to perform queries on a particular table in a database. It has the form:

```
SELECT columns or computations
  FROM table
  WHERE condition
  GROUP BY columns
  HAVING condition
  ORDER BY column [ASC | DESC]
  LIMIT offset, count;
```

`WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, `LIMIT` are all optional. Importantly, in the first line of `SELECT` we can directly specify computations that we want performed.

Examples

To calculate the average number of homeruns, and average number of hits:

```
dbGetQuery(con, paste("SELECT AVG(HR), AVG(H)",
                      "FROM Batting"))
```

We can replicate this simple command on an imported data frame:

```
mean(batting$HR, na.rm = TRUE)
mean(batting$H, na.rm = TRUE)
```

GROUP BY

We can use the GROUP BY option in SELECT to define aggregation groups

```
dbGetQuery(con, paste("SELECT playerID, AVG(HR)",
                      "FROM Batting",
                      "GROUP BY playerID",
                      "ORDER BY AVG(HR) DESC",
                      "LIMIT 5"))
```

Note: the order of commands here matters; try switching the order of GROUP BY and ORDER BY above, and you'll get an error.

WHERE

We can use the `WHERE` option in `SELECT` to specify a subset of the rows to use (pre-aggregation/pre-calculation)

```
dbGetQuery(con, paste("SELECT yearID, AVG(HR)",
                      "FROM Batting",
                      "WHERE yearID >= 1990",
                      "GROUP BY yearID",
                      "ORDER BY AVG(HR) DESC",
                      "LIMIT 5"))
```

Check Yourself

Tasks

Run the following query and determine what it is doing. Write `R` code to do the same thing on the `batting` data frame. Hint use `daply()`.

```
dbGetQuery(con, paste("SELECT teamID, AVG(HR)",
                      "FROM Batting",
                      "WHERE yearID >= 1990",
                      "GROUP BY teamID",
                      "ORDER BY AVG(HR) DESC",
                      "LIMIT 5"))
```

AS and HAVING

We can use AS in the first line of SELECT to rename computed columns

```
dbGetQuery(con, paste("SELECT yearID, AVG(HR) as avgHR",
                      "FROM Batting",
                      "GROUP BY yearID",
                      "ORDER BY avgHR DESC",
                      "LIMIT 5"))
```

We can use the HAVING option in SELECT to specify a subset of the rows to display (post-aggregation/post-calculation)

```
dbGetQuery(con, paste("SELECT yearID, AVG(HR) as avgHR",
                      "FROM Batting",
                      "WHERE yearID >= 1990",
                      "GROUP BY yearID",
                      "HAVING avgHR >= 4.5",
                      "ORDER BY avgHR DESC"))
```

Tasks

Recompute the payroll for each team in 2010, but now with `dbGetQuery()` and an appropriate SQL query. In particular, the output of `dbGetQuery()` should be a data frame with two columns, the first giving the team names, and the second the payrolls, just like your output from `daply()` before. (Hint: your SQL query here will have to use `GROUP BY`.)

Databases: Join

Databases vs. Dataframes

R's dataframes are actually tables

R Jargon	Database Jargon
column	field
row	record
dataframe	table
types of the columns	table **schema**
collection of related dataframes	database
conditional indexing	`SELECT`, `FROM`, `WHERE`, `HAVING`
`d*ply()`	`GROUP BY`
`order()`	`ORDER BY`
`merge()`	`INNER JOIN` or just `JOIN`

JOIN

Sometimes we need to combine information from many tables.

patient_last	patient_first	physician_id	complaint
Morgan	Dexter	37010	insomnia
Soprano	Anthony	79676	malaise
Swearengen	Albert	NA	healthy
Garrett	Alma	90091	nerves
Holmes	Sherlock	43675	addiction

physician_last	physician_first	physicianID	plan
Meridian	Emmett	37010	UPMC
Melfi	Jennifer	79676	BCBS
Cochran	Amos	90091	UPMC
Watson	John	43675	VA

JOIN

- Suppose we want to know which doctors are treating patients for insomnia.
- Complaints are in one table and physicians in another.
- In `R`, we use `merge()` to link the tables by `physicianID`.
- Here `physicianID` or `physician_id` is acting as the key or the identifier.

JOIN

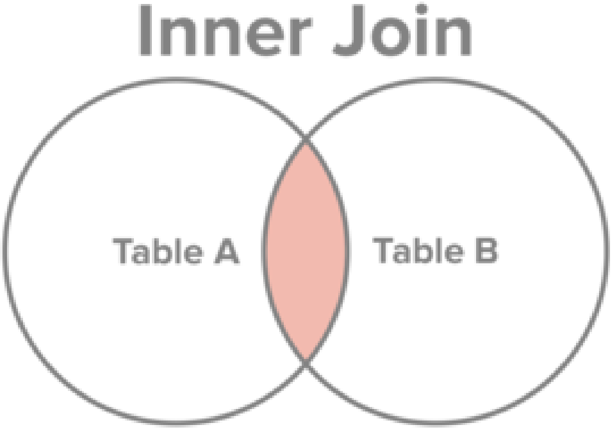
In all we've seen so far with `SELECT`, the `FROM` line has just specified one table. But sometimes we need to combine information from many tables. Use the `JOIN` option for this

There are 4 options for `JOIN`:

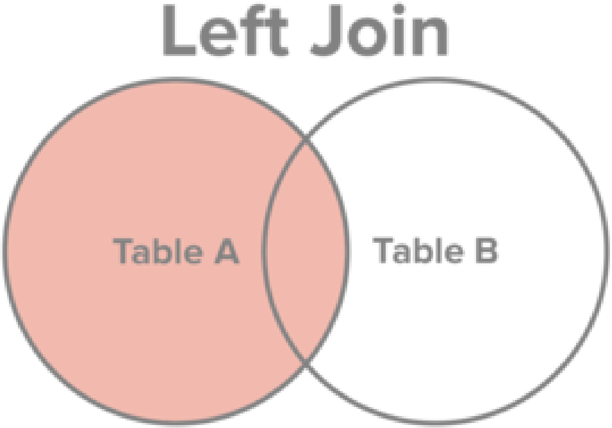
1. `INNER JOIN` or just `JOIN`: retain just the rows each table that match the condition.
2. `LEFT OUTER JOIN` or just `LEFT JOIN`: retain all rows in the first table, and just the rows in the second table that match the condition.
3. `RIGHT OUTER JOIN` or just `RIGHT JOIN`: retain just the rows in the first table that match the condition, and all rows in the second table.
4. `FULL OUTER JOIN` or just `FULL JOIN`: retain all rows in both tables

Fields that cannot be filled in are assigned NA values

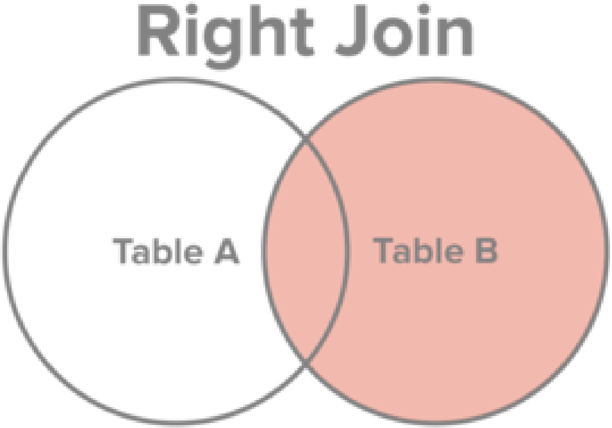
INNER JOIN



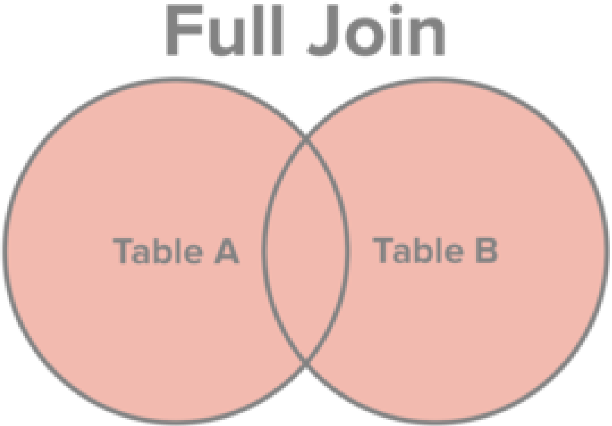
LEFT JOIN



RIGHT JOIN



FULL JOIN



Suppose we want to find the average salaries of the players with the top 10 highest homerun averages. We need to combine the two tables.

```
dbGetQuery(con, paste("SELECT *",
                      "FROM Salaries",
                      "ORDER BY playerID",
                      "LIMIT 8"))

dbGetQuery(con, paste("SELECT yearID, teamID, lgID,
                      playerID, HR",
                      "FROM Batting",
                      "ORDER BY playerID",
                      "LIMIT 7"))
```

Examples

We can use a JOIN on the pair: yearID, playerID.

```
dbGetQuery(con, paste("SELECT yearID, playerID, salary, HR",
                      "FROM Batting JOIN Salaries
                            USING(yearID, playerID)",
                      "ORDER BY playerID",
                      "LIMIT 7"))
```

Note that here we're missing one of David Aardsma's records from the Batting table (i.e., the JOIN discarded 1 record) We can replicate this using merge() on imported data frames:

```
merged <- merge(x = batting, y = salaries,
                by.x = c("yearID","playerID"),
                by.y = c("yearID","playerID"))

names <- c("yearID", "playerID", "salary", "HR")
merged[order(merged$playerID)[1:8], names]
```

For demonstration purposes, we use a `LEFT JOIN` on the pair: `yearID`, `playerID`:

```
dbGetQuery(con, paste("SELECT yearID, playerID, salary, HR",
                      "FROM Batting LEFT JOIN Salaries
                            USING(yearID, playerID)",
                      "ORDER BY playerID",
                      "LIMIT 7"))
```

- Now we can see that we have all 6 of David Aardsma's original records from the Batting table (i.e., the `LEFT JOIN` used them all, and just filled in an `NA` value when it was missing his salary)
- Currently, `RIGHT JOIN` and `FULL JOIN` are not implemented in the `RSQLite` package

Examples

Now, as to our original question (average salaries of the players with the top 10 highest homerun averages):

```
dbGetQuery(con, paste("SELECT playerID, AVG(HR), AVG(salary)",
                      "FROM Batting JOIN Salaries
                            USING(yearID, playerID)",
                      "GROUP BY playerID",
                      "ORDER BY Avg(HR) DESC",
                      "LIMIT 10"))
```

Check Yourself

Tasks

- Using the `Fielding` table, list the 10 worst (highest) number of error (`E`) committed by a player in one season, only considering years 1990 and later. In addition to the number of errors, list the year and player ID for each record.
- By appropriately merging the `Fielding` and `Salaries` tables, list the salaries for each record that you extracted in the last question.

Debugging

Debugging

- **Bug** is the original name for glitches and unexpected defects in code: dates back to at least Edison in 1876.
- Debugging is a the process of locating, understanding, and removing bugs from your code.
- Why should we care to learn about this?
  1. The truth: you're going to have to debug, because you're not perfect and so you can't write perfect code.
  2. Debugging is frustrating and time-consuming, but essential.
  3. Writing code that makes it easier to debug later is worth it, even if it takes a bit more time (lots of our design ideas support this).
  4. Simple things you can do to help: use lots of comments, use meaningful variable names!

How?

- Debugging is (largely) a process of differential diagnosis. Stages of debugging:
    1. Reproduce the error: can you make the bug reappear?
    2. Characterize the error: what can you see that is going wrong?
    3. Localize the error: where in the code does the mistake originate?
    4. Modify the code: did you eliminate the error? Did you add new ones?

Reproduce the bug

Step 0: make it happen again

- Can we produce it repeatedly when re-running the same code, with the same input values?
- And if we run the same code in a clean copy of `R`, does the same thing happen?

Characterize the bug

Step 1: figure out if it's a pervasive/big problem

- How much can we change the inputs and get the same error?
- Or is it a different error?
- And how big is the error?

Localize the bug

Step 2: find out exactly where things are going wrong

- This is most often the hardest part!
- Today, we'll learn how to understand errors, using `print()`.
- There are many more sophisticated debugging tools like `traceback()` or the `R` tool `browser()` which lets you interactively debug. Unfortunately don't have time for these.

Localizing the Bug

Sometimes error messages are easier to decode, sometimes they're harder; this can make locating the bug easier or harder.

```
my.plotter <- function(x, y, my.list = NULL) {
  if (!is.null(my.list))
    plot(my.list, main = "A plot from my.list!")
  else
    plot(x, y, main = "A plot from x, y!")
}

my.plotter(x = 1:8, y = 1:8)
my.plotter(my.list = list(x = -10:10, y = (-10:10)^3))

my.plotter() # Easy to understand error message
my.plotter(my.list = list(x = -10:10, Y = (-10:10)^3))
```

Who called `xy.coords()`? (Not us, at least not explicitly!) And why is it saying `x` is a list? (We never set it to be so!)

Localizing the Bug

Let's modify the function by calling `print()` at various points, to print out the state of variables, to help localize the error.

```
my.plotter <- function(x, y, my.list = NULL) {
  if (!is.null(my.list)) {
    print("Here is my.list:")
    print(my.list)
    print("Now about to plot my.list")
    plot(my.list, main = "A plot from my.list!")
  }
  else {
    print("Here is x:"); print(x)
    print("Here is y:"); print(y)
    print("Now about to plot x, y")
    plot(x, y, main = "A plot from x, y!")
  }
}

my.plotter(my.list = list(x = -10:10, Y = (-10:10)^3))
my.plotter(x = "hi", y = "there")
```