

R four ways (plus a few)

Robert McDonald
Kellogg School, Northwestern University

2018-08-06

The babynames data

- ▶ The Social Security Administration provides state-level babynames data annually since 1910.
- ▶ One shell command creates a single file containing all state-level babynames data

```
cat *.TXT > allstates.TXT
```

- ▶ The resulting file has 5.8 million rows and no header.

File contents

- ▶ Using the head command (in Bash):

```
head -n 6 data/allstates.TXT
AK,F,1910,Mary,14
AK,F,1910,Annie,12
AK,F,1910,Anna,10
AK,F,1910,Margaret,8
AK,F,1910,Helen,7
AK,F,1910,Elsie,6
```

The fields (i.e., columns) are:

- ▶ state, a two-digit abbreviation
- ▶ sex, M or F
- ▶ year, yyyy
- ▶ name
- ▶ number of births

The tasks

- ▶ Four basic data manipulation tasks:
 1. Count the number of distinct states in the data
 2. Count the number of distinct years in the data
 3. Count the number of distinct names in the data
 4. Create a new CSV file that contains the top 10 names nationally, by sex, for each year.
- ▶ We will use Base R with and without loops, `dplyr`, and `data.table`
- ▶ We will find that `dplyr` is faster than base R, and `data.table` is faster still

Base R

R: Base R

- ▶ You will often read that looping in R is slow
- ▶ Beginning users often want to write loops
- ▶ In this problem, there are loops with few iterations (over states and sex) and many iterations (over names)
- ▶ There is a big gain to replacing the `name` loop, not so much of a gain from replacing the other loops

Using only loops

- The following code is very slow. Experienced R users will wince...

```
x = read.csv('data/allstates.TXT', header=FALSE,
             stringsAsFactors=FALSE)
names(x) <- c('state', 'sex', 'year', 'name', 'n')
print(length(table(x$state)))
print(length(table(x$year)))
print(length(table(x$name)))
top10 <- list()
sexes <- unique(x$sex)
for (i in unique(x$year)) {
  for (j in sexes) {
    tmp <- x[x$year == i & x$sex == j, ]
    names <- unique(tmp$name)
    lnames <- length(names)
    nvec <- vector(length=lnames)
    for (k in 1:lnames) {
      nvec[k] <- sum(tmp$n[tmp$name == names[k]])
    }
    tmp <- data.frame(year=i, sex=j, name=names, n=nvec)
    tmp <- tmp[order(-tmp$n), ]
    top10 <- rbind(top10, head(tmp, n=10), make.row.names=FALSE)
  }
}
write.csv(top10, file='data/babynames10Rbase_loop2.csv',
          row.names=FALSE)
print(head(top10))
```

Results: R with explicit loops

```
[1] 51
[1] 107
[1] 31014
```

	year	sex	name	n
1	1910	F	Mary	22848
2	1910	F	Helen	10479
3	1910	F	Margaret	8222
4	1910	F	Dorothy	7314
5	1910	F	Ruth	7209
6	1910	F	Anna	6433

```
user system elapsed
124.312 0.688 125.032
```


R with some loops

- ▶ We can use the `aggregate` function to replace the innermost loop, which sums individual names across states for a given year and sex
- ▶ Using `aggregate` we apply the function `sum` to the formula `n ~ year + sex + name`

```
xa <- aggregate(n ~ year + sex + name, data=x, FUN=sum)
```

- ▶ This is substantially faster

R with some loops

```
x = read.csv('data/allstates.TXT', header=FALSE,
             stringsAsFactors=FALSE)
names(x) <- c('state', 'sex', 'year', 'name', 'n')
print(length(table(x$state)))
print(length(table(x$year)))
print(length(table(x$name)))
xa <- aggregate(n ~ year + sex + name, data=x, FUN=sum)
xa <- xa[order(xa$year, xa$sex, -xa$n), ]
top10 <- data.frame()
for (i in unique(xa$year)) {
  for (j in unique(xa$sex)) {
    tmp = head(subset(xa, xa$year == i & xa$sex == j), n=10)
    top10 <- rbind(top10, tmp, make.row.names=FALSE)
  }
}
write.csv(top10, file='data/babynames10Rbase_loop.csv',
          row.names=FALSE)
print(head(top10))
```

Results: R with some loops

```
[1] 51
```

```
[1] 107
```

```
[1] 31014
```

	year	sex	name	n
1	1910	F	Mary	22848
2	1910	F	Helen	10479
3	1910	F	Margaret	8222
4	1910	F	Dorothy	7314
5	1910	F	Ruth	7209
6	1910	F	Anna	6433

	user	system	elapsed
	28.208	0.648	28.875

Without explicit loops

- Using `by` to replace the year/sex loops yields more “R-ish” code. But it is only slightly faster

```
x = read.csv('data/allstates.TXT', header=FALSE,
             stringsAsFactors=FALSE)
names(x) <- c('state', 'sex', 'year', 'name', 'n')
print(length(table(x$state))) ## note the similarity to python
print(length(table(x$year)))
print(length(table(x$name)))
xa <- aggregate(n ~ year + sex + name, data=x, FUN=sum)
xa <- xa[order(xa$year, xa$sex, -xa$n), ]
xatop10 <- by(xa, list(xa$year, xa$sex), head, n=10)
top10 <- do.call(rbind, xatop10)
write.csv(top10, file='data/babynames10Rbase_noloop.csv',
          row.names=FALSE)
print(head(top10))
```

Results: R without explicit loops

```
[1] 51
```

```
[1] 107
```

```
[1] 31014
```

	year	sex	name	n
399169	1910	F	Mary	22848
236460	1910	F	Helen	10479
388683	1910	F	Margaret	8222
163023	1910	F	Dorothy	7314
498889	1910	F	Ruth	7209
37086	1910	F	Anna	6433

user	system	elapsed
23.984	0.284	24.273

dplyr

The dplyr approach

- ▶ The dplyr package permits data manipulation with echoes of SQL.
- ▶ There are explicit “verbs” for data manipulation tasks (sorting, filtering by row, selecting columns, grouping, summarizing, etc.)
- ▶ dplyr is very fast to code
- ▶ Compare the dplyr code to the “no-loop” base R code

dplyr and tidy

```
x <- read_csv('data/allstates.TXT',
              col_names=c('state', 'sex', 'year', 'name', 'n'),
              col_types = cols(sex = col_character())
            )
print(nrow(distinct(x, state)))
print(nrow(distinct(x, year)))
print(nrow(distinct(x, name)))
out = x %>%
  group_by(year, sex, name) %>%
  summarize(n = sum(n)) %>%
  arrange(year, sex, desc(n)) %>%
  filter(row_number(desc(n)) <= 10)
## do(head(., n=10)) ## works in place of filter
write_csv(out, path='data/babynames10Rdpdplyr.csv')
print(head(out))
```


Results: dplyr

```
[1] 51
[1] 107
[1] 31014
# A tibble: 6 x 4
# Groups:   year, sex [1]
   year  sex  name    n
  <int> <chr> <chr> <int>
1  1910    F   Mary 22848
2  1910    F   Helen 10479
3  1910    F Margaret 8222
4  1910    F  Dorothy 7314
5  1910    F    Ruth 7209
6  1910    F    Anna 6433
   user  system elapsed
7.244   0.168   7.413
```

R: dplyr with map

- The purrr functions nest and map can also be used.

```
x <- read_csv('data/allstates.TXT',
              col_names=c('state', 'sex', 'year', 'name', 'n'),
              col_types = cols(sex = col_character())
            )

print(nrow(distinct(x, state)))
print(nrow(distinct(x, year)))
print(nrow(distinct(x, name)))

out = x %>%
  group_by(year, sex, name) %>%
  summarize(n = sum(n)) %>%
  arrange(year, sex, desc(n)) %>%
  nest() %>% ## will nest on the grouping variables
  map_df(.x=.$data, .f=head, n=10)

write_csv(out, path='data/babynames10Rdplyrmap.csv')
print(head(out))
```

Results: dplyr with map

```
[1] 51
[1] 107
[1] 31014
# A tibble: 6 x 2
  name      n
  <chr> <int>
1   Mary 22848
2  Helen 10479
3 Margaret 8222
4 Dorothy 7314
5   Ruth 7209
6   Anna 6433
  user  system elapsed
7.308   0.164    7.471
```

The data.table package

data.table

- ▶ data.table is designed explicitly for manipulation of large data sets. The syntax is more abstract than in dplyr
- ▶ Like dplyr, it permits chaining commands.
- ▶ For a data table, DT, with row i, column j, grouped by by, the syntax is DT[i, j, by]

```
library(data.table)
y <- fread("data/allstates.TXT",
           col.names=c('state', 'sex', 'year', 'name', 'n'))
print(y[, uniqueN(state)])
print(y[, uniqueN(year)])
print(y[, uniqueN(name)])
y2 = y[, .(total = sum(n)), by=.(year, sex, name)][
  order(year, sex, -total)]
out = y2[, head(.SD, 10), by=.(year, sex)]
fwrite(out, file='data/babynames10RDT.csv')
print(head(out))
```

Results: data.table

```
[1] 51
```

```
[1] 107
```

```
[1] 31014
```

	year	sex	name	total
1:	1910	F	Mary	22848
2:	1910	F	Helen	10479
3:	1910	F	Margaret	8222
4:	1910	F	Dorothy	7314
5:	1910	F	Ruth	7209
6:	1910	F	Anna	6433

	user	system	elapsed
	1.896	0.080	1.971

Reading and writing data files

read.csv vs read_csv vs fread

- ▶ In each comparison we have used the “native” function for reading and writing in that particular environment
- ▶ read.csv and write.csv are in base R
- ▶ read_csv and write_csv are in dplyr
- ▶ fread and fwrite are in data.table

Comparison: Reading

- Note that `read_csv` without column types will throw an error because it infers that the variable `sex` is logical

```
system.time(x <- read.csv('data/allstates.TXT',
                          header=FALSE))

  user  system elapsed
 7.124   0.296   7.447

system.time(
  x <- read_csv('data/allstates.TXT',
                progress=FALSE,
                col_names=c('state', 'sex', 'year', 'name', 'n'),
                col_types = cols(sex = col_character())
  )
)

  user  system elapsed
 2.276   0.080   2.361

system.time(x <- fread("data/allstates.TXT",
                      col.names=c('state', 'sex', 'year', 'name', 'n'))
)

  user  system elapsed
 1.008   0.012   1.019
```

Writing files

- ▶ We can choose
 1. Writing CSV or Rdata files
 2. If Rdata: Writing compressed or uncompressed
 3. If CSV: Using one of three functions

Comparison: Writing

```
system.time(write.csv(x, file='/tmp/save1.CSV'))
  user  system elapsed
15.876   0.292  16.261
system.time(write_csv(x, path='/tmp/save2.CSV'))
  user  system elapsed
 3.624   0.112   3.796
system.time(fwrite(x, file='/tmp/save3.CSV'))
  user  system elapsed
 0.624   0.084   0.231
system.time(save(x, file='/tmp/save.Rdata'))
  user  system elapsed
 6.064   0.008   6.089
system.time(save(x, file='/tmp/save2.Rdata', compress=FALSE))
  user  system elapsed
 2.004   0.144   2.156
system.time(saveRDS(x, file='/tmp/save.RDS', compress=FALSE))
  user  system elapsed
 1.984   0.128   2.177
cat(system('ls -al /tmp/save*', intern=TRUE), sep='\n')
-rw-rw-r-- 1 rmcd rmcd 206677054 Nov  4 13:11 /tmp/save1.CSV
-rw-rw-r-- 1 rmcd rmcd 114367569 Nov  4 13:11 /tmp/save2.CSV
-rw-rw-r-- 1 rmcd rmcd 238673419 Nov  4 13:11 /tmp/save2.Rdata
-rw-r--r-- 1 rmcd rmcd 114367569 Nov  4 13:11 /tmp/save3.CSV
-rw-r--r-- 1 rmcd rmcd 114367569 Nov  3 17:59 /tmp/save.csv
-rw-rw-r-- 1 rmcd rmcd 17278437  Nov  4 13:11 /tmp/save.Rdata
-rw-rw-r-- 1 rmcd rmcd 238673393 Nov  4 13:11 /tmp/save.RDS
```

Reading the Rdata files back in

```
system.time(load('/tmp/save.Rdata'))
  user  system elapsed
 4.080   0.032   4.115
system.time(load('/tmp/save2.Rdata'))
  user  system elapsed
 2.760   0.060   2.819
system.time(y <- readRDS(file='/tmp/save.RDS'))
  user  system elapsed
 2.752   0.052   2.805
```

- The relative times depend on both CPU and disk speeds

Conclusion about reading and writing

- ▶ Use `fread` and `fwrite` if the file is not small
- ▶ When writing files using `save`, set `compress=FALSE`
- ▶ In the previous examples:
 - ▶ differences in file *reading* speed would have been substantial
 - ▶ differences in file *writing* speed would have been small, because the output file was small
- ▶ `dplyr` using `fread` and `fwrite` runs in under 5 seconds

SQL

Creating an SQL Connection

- ▶ It is possible to use dplyr with an SQL connection
- ▶ SQL databases have their own passwords
 - ▶ Password security becomes an issue when creating scripts. Two solutions are the keyringr package, which reads your local keyring, and the getPass package, which will prompt you for the password when making a connection.)

```
conn = DBI::dbConnect(RPostgreSQL::PostgreSQL(),  
                      user=username,  
                      password=pw,  
                      dbname='babynames_by_state',  
                      host='localhost'  
                      )
```

R, using a connection to an SQL database

- ▶ A database connection can be used with either SQL or R.
- ▶ dplyr code works with the remote database

```
names.tbl <- tbl(conn, 'names')
distinct(names.tbl, state) %>% count
distinct(names.tbl, year) %>% count
distinct(names.tbl, name) %>% count
tmp <- names.tbl %>%
  group_by(year, sex, name) %>%
  summarize(total=sum(n)) %>%
  arrange(year, sex, -total) %>%
  filter(row_number() <= 10)
print(head(collect(tmp)))
```


Results: SQL via dplyr

- It's hard to assess the relative speed because the remote SQL engine and network play a role

```
# A tibble: 6 x 4
# Groups:   year, sex [1]
  year  sex    name total
<int> <chr>  <chr> <dbl>
1  1910    F    Mary 22848
2  1910    F   Helen 10479
3  1910    F Margaret 8222
4  1910    F  Dorothy 7314
5  1910    F    Ruth 7209
6  1910    F    Anna 6433
  user  system elapsed
0.260   0.000  22.657
```

The dplyr query

- Use `show_query()` to examine the query constructed by dplyr

```
show_query(tmp)
```

```
<SQL>
```

```
SELECT "year", "sex", "name", "total"
```

```
FROM (SELECT "year", "sex", "name", "total", row_number() OVER (PARTITION BY "y
```

```
FROM (SELECT *
```

```
FROM (SELECT "year", "sex", "name", SUM("n") AS "total"
```

```
FROM "names"
```

```
GROUP BY "year", "sex", "name") "mdtmabreug"
```

```
ORDER BY "year", "sex", -"total") "pvuhrhznzug") "liekblzjas"
```

```
WHERE ("zzz3" <= 10.0)
```

Manipulation using SQL

- ▶ Access the SQL connection by setting `connection=conn` in the the chunk options.

```
select count(distinct state) from names;
```

Table 1: 1 records

count
51

Manipulation using SQL, continued

```
select count(distinct year) from names;
```

Table 2: 1 records

count
107

```
select count(distinct name) from names;
```

Table 3: 1 records

count
31014

Direct SQL

- ▶ The following chunk is pure SQL. The result of the statement will be assigned to the data frame `babynames10sql`, specified in the chunk options as `output.var='babynames10sql'`.

```
-- name the output with chunk option "output.var='babynames10sql'"
create temp table tmp as
select * from
(
  select name, year, sex, SUM(n),
  ROW_NUMBER () OVER (
  PARTITION BY year, sex
  order by year, sex, sum(n) desc
  )
from names
group by year, sex, name
order by year, sex, sum desc
) as foo
where row_number <= 10;
select year, sex, name, sum from tmp;
```

Back to R to look at the results...

► Now we're using R again.

```
head(babynames10sql)
```

	year	sex	name	sum
1	1910	F	Mary	22848
2	1910	F	Helen	10479
3	1910	F	Margaret	8222
4	1910	F	Dorothy	7314
5	1910	F	Ruth	7209
6	1910	F	Anna	6433

```
write_csv(babynames10sql, path='data/babynames10sql.csv')
```

Command line

Command line

- ▶ You can use the command line to do some of this.
- ▶ The works in Linux, OS X, and Windows with either git-bash or the Linux Subsystem for Windows.

```
## Number of states
```

```
cut -d, -f1 data/allstates.TXT | uniq | wc -l  
51
```

```
## Number of years
```

```
cut -d, -f3 data/allstates.TXT | uniq | sort | uniq | wc -l  
107
```

```
## Number of names
```

```
cut -d, -f4 data/allstates.TXT | uniq | sort | uniq | wc -l  
31014
```

- ▶ Finally, this is included just for fun: sort the entire original file from the command line, using multiple sort keys

```
## Sort the entire original data set by name, n (descending), year,  
## state. "-t ," specifies delimiter, and the others are sort  
## keys. See "man sort"
```

```
sort -t , -k 4,4d -k 5,5nr -k 3,3n -k 1,1d allstates.TXT > out.csv
```


Conclusions

- ▶ Use `dplyr` or `data.table` (especially for large data)
- ▶ For large files, save uncompressed
- ▶ For CSV files, `data.table::fread` and `data.table::fwrite` are outstanding
- ▶ Some loops are okay, but using loops for everything kills performance and takes too much time to code
- ▶ Learn to use the command line
 - ▶ If you are using Linux or OS X, you have what you need
 - ▶ If you are using Windows, you will need to install either `git-bash` or the Linux Subsystem for Windows (only for Windows 10)