OPP-SQL-2023

Saatvik Kher

2023-11-15

Contents

This document details the data scraping and upload process for the Stanford Open Policing Project SQL Server.

Scraping

We parse the Stanford Open Policing Project website and download all the RDS files (files ending in ".rds").

```
url <- "https://openpolicing.stanford.edu/data/"
doc <- htmlParse(readLines(url), asText=TRUE)
links <- xpathSApply(doc, "//a/@href")
free(doc)
rm(doc)
all_links <- as.character(links[grep('_.*rds', links)])
```

We define a regex function to parse the urls and extract filenames for all the datasets.

```
fileToDfName <- function(input_string){
  str_extract(input_string, regex("([A-Za-z]+(_[A-Za-z]+)+).*([0-9]+(_[0-9]+)+)"))
}

filenames <- map_chr(all_links, fileToDfName)
filenames_data <- str_c("data/", filenames, ".rds", sep="")
```

This chunk downloads every RDS file to a folder called "data" in your current working directory.

```
downloadFile <- function(input_link) {
    tic()
    tmpDF <- tryCatch(
                expr = {
                  name <- fileToDfName(input_link)
                  filename <- str_c("poster/", name, ".rds", sep = "")
                  download.file(input_link, filename)
                },
                error = function(e){
                    print(str_c("Caught an error in", input_link, e, sep = " "))
                },
                warning = function(w){
                    print(str_c("Caught an error in", input_link, w, sep = " "))
                },
                finally = {
                    cat()
                }
              )
    cat("Time for ", fileToDfName(input_link), ": ", sep="")
    toc(log = TRUE)
    return(tmpDF)
}


tic()
map_dfr(all_links, downloadFile)
toc()
```

Note: 9 RDS files were corrupted. They had to be downloaded as CSVs and converted to RDS. The GitHub repository maintainers were informed of this. These files are:

- co_aurora_2023_01_26

- ct_statewide_2020_04_01
- fl_statewide_2020_04_01
- ga_statewide_2020_04_01
- il_statewide_2020_04_01
- ky_louisville_2023_01_26
- la_new_orleans_2020_04_01
- md_statewide_2020_04_01
- tx_statewide_2020_04_01

Query-Table

Now that all the files have been downloaded locally, we can start building a query table. The query-table contains information on the `state`, `city`, `uploadDate`, `num_rows`, `earliest_date`, `latest_date` as well as the coverage of every variable in the SOPP dataset. An *NA* indicates 0% coverage.

```
coverageRow <- function(input_link) {
    tic()
    # We use a try-catch block to catch any errors while reading the RDS.
    tmpDF <- tryCatch(
                expr = {
                  name <- fileToDfName(input_link)
                  tmpDF <- readRDS(input_link)
                },
                error = function(e){
                    print(str_c("Caught an error in", input_link, e, sep = " "))
                    tmpDF <- tibble(state_city_uploadDate = fileToDfName(input_link))
                },
                warning = function(w){
                    print(str_c("Caught an error in", input_link, w, sep = " "))
                    tmpDF <- tibble(state_city_uploadDate = fileToDfName(input_link))
                },
                finally = {
                    cat()
                }
            )
    cat("Time for ", fileToDfName(input_link), ": ", sep="")
    toc(log = TRUE)

    # This is the case where the RDS file was read correctly
    # Hacky way to differentiate correctly read files from corrupt RDS files
    if (ncol(tmpDF) > 1) {
      name <- fileToDfName(input_link)
      row <- ((1 - colMeans(is.na(tmpDF))) * 100) %>%
        as_tibble_row() %>%
        mutate(state_city_uploadDate = name,
               num_rows = nrow(tmpDF),
               earliest_date = min(tmpDF$date, na.rm = TRUE),
               latest_date = max(tmpDF$date, na.rm = TRUE))

      rm(tmpDF)
      return(row)
    }
    return(tmpDF)
}
```

```
tic()
query_table <- map_dfr(filenames_data, coverageRow)
toc()
```

We save the table as a CSV file called "OPP_querytable.csv" in the current working directory.

```
tic()
query_table %>%
  replace(is.na(.), 0) %>%
  write_csv("OPP_querytable.csv")
cat("Time to Write CSV: ")
toc()
```

This is the code to parse and clean the `state_city_uploadDate` filename.

```
query_table %>%
  mutate(state =  toupper(str_sub(state_city_uploadDate, end=2)),
         city = str_to_title(str_replace(str_sub(state_city_uploadDate, 4, -12), pattern = "_", " ")),
         upload_date = as_date(str_sub(state_city_uploadDate, -10))) %>%
  relocate(state, city, upload_date, num_rows, earliest_date, latest_date) %>%
  select(-state_city_uploadDate) %>%
  write_csv("OPP_querytable.csv")
```

Alternate ways to summarize for coverage

Here is a method to use aggregate functions to query missingness by year. We could alternatively build the query-table using an aggregate function like this.

```
sumNA <- function(x) {
  (1 - mean(is.na(x))) * 100
}

tmpDF %>%
  mutate(year = year(date)) %>%
  group_by(year) %>%
  summarise(across(everything(), sumNA))
```

Uploading to the SQL Server

Setup

We first establish a connection to the server.

```
con <- dbConnect(
  MySQL(), host = "traffic.st47s.com", user = "loading",
  password = Sys.getenv("SQL_LOADING"), dbname = "traffic")
```

As an additional step to help monitor the upload process, we sort the RDS files in order of size.

```
filenames_sorted <- query_table %>% bind_cols(link = filenames_data) %>%
  select(num_rows, link) %>%
  arrange(num_rows) %>%
  pull(link)
```

Upload

This chunk will read every RDS file from the "data/" folder and upload it to the server.

```
uploadLinksToDatabase <- function(filename){
  tic()
  tryCatch(expr = {
    tmpDF <- readRDS(filename)
    tmpDF <- convertLogicalToInt(tmpDF)
    name <- fileToDfName(filename)
    dbWriteTable(con, name, tmpDF, overwrite = TRUE, row.names = FALSE)
    rm(tmpDF)
    },
    error = function(e){
      print(str_c("Caught an error in", filename, e, sep = " "))
      },
    warning = function(w){
      print(str_c("Caught an error in", filename, w, sep = " "))
      },
    finally = cat())
  cat("Time to upload ", fileToDfName(filename), ": ", sep="")
  toc(log = TRUE)
}



tic()
map_dfr(filenames_sorted, uploadLinksToDatabase)
cat("Total Time: ")
toc()
```

Fixing Data Types

The default datatype for the date and time columns is *text*. We modify these types in the server.

```
# Extract tables with date and time variables
filenames_dt <- query_table %>%
  bind_cols(filename = filenames) %>%
  filter(!is.na(time) & !is.na(date)) %>%
  pull(filename)

# Extract table with only date variable not time
filenames_d <- query_table %>%
  bind_cols(filename = filenames) %>%
  filter(is.na(time) & !is.na(date)) %>%
  pull(filename)
```

To modify data types for `filenames_dt`:

```
tic()
for (table in filenames_dt) {
  cat(table, " ", sep="")
  dbGetQuery(con, str_c("ALTER TABLE ", table, " MODIFY COLUMN date Date, MODIFY COLUMN time Time", sep=
}
cat("Time to modify datatypes (dt)")
toc()
```

To modify data types for `filenames_d`:

```
tic()
for (table in filenames_d) {
  cat(table, " ", sep="")
  dbGetQuery(con, str_c("ALTER TABLE ", table, " MODIFY COLUMN date Date", sep=""))
}
cat("Time to modify datatypes (d)")
toc()
```

This concludes the data upload process and the server is now ready.

Queries

We can list all 88 tables in the database as follows:

```
dbListTables(con)
```

To check the size of the database:

```
dbGetQuery(con, "SELECT table_schema AS 'Database',
ROUND(SUM(data_length + index_length) / 1024 / 1024 / 1024, 2) AS 'Size (GB)'
FROM information_schema.TABLES
GROUP BY table_schema;") %>%
  filter(Database == "traffic")
```

Alternate linux shell command (root access required):

```
ls -lh /var/lib/mysql/traffic
```

To query the size of each individual table:

```
dbGetQuery(con, "SELECT
     table_schema as `Database`,
     table_name AS `Table`,
     round(((data_length + index_length) / 1024 / 1024), 2) `Size in MB`
FROM information_schema.TABLES
ORDER BY (data_length + index_length) DESC;")
```

To look at the variables and datatypes in a table, for example *ar_little_rock_2020_04_01*:

```
dbGetQuery(con, "describe ar_little_rock_2020_04_01;") %>%
  select(Field, Type)
```

To get a 10% random sample w/ seed = 47:

```
dbGetQuery(con, "SELECT * FROM ar_little_rock_2020_04_01 WHERE rand(47) <= 0.1")
```

Useful Linux Commands

Might need root access to see the entire output

Disk usage from */*

```
du -hx --max-depth=1 /
```

Disk usage from */home*

```
du -hx --max-depth=1 /home
```

Check Memory usage (RAM not disk):

```
free -h
```

Logs Check how much storage logs are taking

```
journalctl --disk-usage
```

This is long and unwieldy

```
journalctl
```

Instead, you can check the logs of the current boot:

```
journalctl -b
```

Or specify a time range

```
journalctl --since "1 hour ago"
journalctl --since "2 days ago"
journalctl --since "2015-06-26 23:15:00" --until "2015-06-26 23:20:00"
```

MySQL Logs (the f is for follows):

```
journalctl -u mysql.service -f
```

Vaccum logs (only keep the previous 2 days)

```
journalctl --vacuum-time=2d

Other options:
--vacuum-size=BYTES   Reduce disk usage below specified size
--vacuum-files=INT    Leave only the specified number of journal files
--vacuum-time=TIME    Remove journal files older than specified time
```

Long Term Solution

You can write a `cron` job to run the vacuum cleaner at the desired frequency. Alternatively, you can change the `SystemMaxUse` attribute in */etc/systemd/journald.conf*:

```
vim /etc/systemd/journald.conf    Scroll to SystemMaxUse and set to desired space
```

It is currently set to 500MB.

References

E. Pierson, C. Simoiu, J. Overgoor, S. Corbett-Davies, D. Jenson, A. Shoemaker, V. Ramachandran, P. Barghouty, C. Phillips, R. Shroff, and S. Goel. "A large-scale analysis of racial disparities in police stops across the United States". Nature Human Behaviour, Vol. 4, 2020.

Appendix

A number of discrepancies were noted between the generated query-table and the information on the OPP website. This chunk compares the "number of stops" per table parsed from the website with our query-table. The result includes tables where the number of stops are not updated/correctly displayed on the website. The `web_nums` string is the result of parsing the website.

```
query_table <- read_csv("OPP_querytable.csv")
```

```
## Rows: 88 Columns: 229
## -- Column specification --------------------------------------------------------
```

```
## Delimiter: ","
## chr    (2): state, city
## dbl  (224): num_rows, raw_row_number, date, time, lat, lng, subject_age, sub...
## date   (3): upload_date, earliest_date, latest_date
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
web_nums <- ("13,641
480,599
157,015
3,494,153
87,876
189,685
365,924
5,418,400
133,405
90,523
382,844
905,070
152,833
46,268
31,778,515
41,629
257,606
1,870,731
3,112,852
18,435
1,175,339
16,383
7,297,538
2,849,017
1,906,772
2,441,335
104,814
2,108,098
12,748,173
265,019
1,030,376
146,560
6,921
512,088
3,416,238
854,759
3,587,052
800,302
675,156
10,038,706
758,412
921,216
1,598,453
326,024
486,998
600,031
```

```
856,400
20,286,645
452,560
82,535
330,132
9,031,494
259,822
195,100
3,845,334
111,089
737,282
74,093
7,962,169
315,281
128,157
7,225,577
945,107
438,248
1,058,518
1,865,096
509,671
8,983,807
435,895
3,092,351
3,828,141
112,526
483,255
159,840
2,045,972
546,101
249,043
1,301,103
27,426,840
5,006,847
36,842
283,285
319,959
11,333,425
271,912
331,450
1,058,816
172,948")

(web_nums %>%
  str_split(pattern = "\n")) %>%
  unlist() %>%
  tibble() %>%
  mutate(web_numbers = as.numeric(str_replace_all(`.`, ",", ""))) %>%
  select(-`.`) %>%
  bind_cols(query_numbers = query_table$num_rows, state = query_table$state, city = query_table$city, u
  mutate(match = web_numbers == query_numbers) %>%
  filter(!match) %>%
  mutate(difference = web_numbers - query_numbers) %>%
```

```
arrange(-difference) %>%
select(-match) %>%
kable()
```

web_numbers	query_numbers	state	city	upload_date	difference
2849017	2818240	FL	Tampa	2020-04-01	30777
152833	152834	CA	San Jose	2020-04-01	-1
3112852	3112853	CO	Statewide	2020-04-01	-1
111089	111090	NV	Henderson	2020-04-01	-1
5418400	5418402	CA	Los Angeles	2020-04-01	-2
133405	133407	CA	Oakland	2020-04-01	-2
146560	146562	KY	Louisville	2023-01-26	-2
737282	737285	NV	Statewide	2020-04-01	-3
8983807	8983810	SC	Statewide	2020-04-01	-3
36842	36845	VT	Burlington	2023-01-26	-3
18435	18439	CT	Hartford	2020-04-01	-4
512088	512092	LA	New Orleans	2020-04-01	-4
159840	159845	TX	Garland	2020-04-01	-5
509671	509681	RI	Statewide	2020-04-01	-10
921216	921228	MT	Statewide	2023-01-26	-12
1058816	1058902	WI	Statewide	2020-04-01	-86
382844	383027	CA	San Diego	2020-04-01	-183
195100	195298	NJ	Camden	2020-04-01	-198
172948	173311	WY	Statewide	2020-04-01	-363
3828141	3829082	TN	Statewide	2020-04-01	-941
3494153	3498159	AZ	Statewide	2020-04-01	-4006
3587052	3669665	MD	Statewide	2020-04-01	-82613
1058518	1143017	OR	Statewide	2020-04-01	-84499