Reproducible data collection

Ian D. Gow

4 January 2026

An exercise I have assigned to students in the past is to go to the online supplements and datasheets page for the *Journal of Accounting Research*, pick an issue of the journal and evaluate whether one could reproduce the analysis in the associated paper using the materials made available there. Generally, the answer has been negative.[1] That said, it seems that the *Journal of Accounting Research* is still the (relative) leader among accounting-focused academic journals with regard to requiring authors to supply materials.

Tip

In writing this note, I use several packages including those listed below.[2] This note was written using Quarto and compiled with RStudio, an integrated development environment (IDE) for working with R. The source code for this note is available here and the latest version of this PDF is here.

```r
library(tidyverse)
library(DBI)
library(farr)
```

My reason for visiting the datasheets page this week was to get a sense for how people are doing data analysis these days. Are people moving from R to Python, as the latter gets stronger for data science? Or are they preferring the domain-specific strengths of R? The answer is: neither. Based on the six papers in Volume 63, Issue 1 that are not listed as "datasheet and code forthcoming" and that provide something in terms of data, all used some combination of SAS and Stata. This is probably not much different from what you would have seen around fifteen years ago.

But looking at the first paper in Volume 63, Issue 2, I see a mix of Stata and R. So, moving to the assignment I have given to students in the past, how reproducible is the analysis contained therein? I made some progress on this, but I didn't get far (to be fair, I didn't try very hard).

[1] Not always, as several of the papers covered in *Empirical Research in Accounting: Tools and Methods* are replicated there using code from the *Journal of Accounting Research* page.

[2] Execute `install.packages(c("tidyverse", "DBI", "duckdb", "farr", "arrow", "dbplyr")` within R to install all the packages you need to run the code in this note.

The first data set in the R code file is described as "Employment and Investment" in the PDF data sheet (`DLR datasheet.pdf`). The source is "the U.S. Bureau of Labor Statistics (BLS) website" and the authors say "we downloaded Zip files. We then merged these data with our sample dataset. There was no manual entry at any point in the process." However, when I go to the website, I struggled to figure out what data file covers "employment and investment".

Based on the filenames listed in the code (e.g., `2001_annual_singlefile.zip`), ChatGPT suggested that I should go to the BLS's QCEW data files page, where QCEW stands for "Quarterly Census of Employment and Wages" (not sure what happened to "investment" ... I guess it's a typo of sorts). If the authors had provided that link, it would've been very helpful for anyone trying to reproduce their analysis.

Getting to the QCEW site, I guess the authors clicked the link for each year's data—the paper used data from 2001 to 2019—and saved the `.zip` data file somewhere on one of their computers. Again, one can do better. A small function can get a single year's data and `lapply()` can do this for all years. I tend to use an environment variable (`RAW_DATA_DIR`) to indicate where I put "raw" data files like these and in this case, I put them in a directory named `bls` under that directory. Using environment variables in this way yields code that is much more reproducible than things like `D:\Users\me\my_projects\this_project` that seem very common on the JAR datasheets site. Note that the code below does not download a file if I already have it:

```r
years <- 2001:2019L
```

```r
get_bls_zip_data <- function(year, raw_data_dir = NULL,
                             schema = "bls") {

  if (is.null(raw_data_dir)) {
    raw_data_dir <- file.path(Sys.getenv("RAW_DATA_DIR"), schema)
  }

  if (!dir.exists(raw_data_dir)) dir.create(raw_data_dir, recursive = TRUE)

  url <- stringr::str_glue("https://data.bls.gov/cew/data/files/{year}",
                           "/csv/{year}_annual_singlefile.zip")
  t <- file.path(raw_data_dir, basename(url))
  if (!file.exists(t)) download.file(url, t)
  return(TRUE)
}

res <- lapply(years, get_bls_zip_data)
```

The original R code supplied by the authors had code more or less like this (I omitted the code for `bls03` through `bls17` for reasons of space):

```
bls01_in <- read.csv(unzip("Data/BLS/2001_annual_singlefile.zip"));
  bls01 <- bls_process(bls01_in); rm(bls01_in)
bls02_in <- read.csv(unzip("Data/BLS/2002_annual_singlefile.zip"));
  bls02 <- bls_process(bls02_in); rm(bls02_in)
...
bls18_in <- read.csv(unzip("Data/BLS/2018_annual_singlefile.zip"));
  bls18 <- bls_process(bls18_in); rm(bls18_in)
bls19_in <- read.csv(unzip("Data/BLS/2019_annual_singlefile.zip"));
  bls19 <- bls_process(bls19_in); rm(bls19_in)

bls_all <- rbind(bls01, bls02, bls03, bls04, bls05, bls06, bls07, bls08,
                 bls09, bls10, bls11, bls12, bls13,
                 bls14, bls15, bls16, bls17, bls18, bls19)
```

I tweaked this code a little bit without changing its basic function. First, I replaced `Data/BLS/` with something based on where I downloaded the files (`{raw_data_dir}`). Second, I put the `unzip()` and `read.csv()` calls *inside* the `bls_process()` function. Third, I replaced hard-coded file names such as `2001_annual_singlefile.zip` with something like `{year}_annual_singlefile.zip`. Finally, I replaced the code creating `bls_all` with `bls_all <- rbind(lapply(2001:2019L, bls_process))`.

I put this code in a small R file and I can call this code using `source()`. The modified "original" code I used is available here.

```
system.time(source("get_bls_data_orig.R"))
```

```
   user  system elapsed
334.403  20.244 356.040
```

So this takes about 6 minutes. This is not a big deal for the researchers, who probably worked on this project for years. But it's an additional cost for anyone attempting to replicate the analysis in the paper.

Note that the original "original" code left a lot of unzipped text files lying about on my hard drive; my modified code eliminates these (using `unlink()`). Also, quite a lot of memory is used up by having all the dataframes in memory before calling `rbind()`.

Can we do better? I think one approach is to put the processed data into a **parquet data repository** of the kind I describe here. I use the environment variable `DATA_DIR` to keep track of the location of my repository (note that this could be different for different projects) and put data files in different "schemas" within `DATA_DIR`. In this case, I will put the data under `bls`.

For my first attempt, I will use `read_csv()` from the `readr` package to read in the unzipped files and then `write_parquet()` from the `arrow` package to save the data to a parquet file in my data repository.

```r
get_bls_data <- function(year, schema = "bls", data_dir = NULL,
                         raw_data_dir = NULL) {

  if (is.null(raw_data_dir)) {
    raw_data_dir <- file.path(Sys.getenv("RAW_DATA_DIR"), schema)
  }

  if (is.null(data_dir)) data_dir <- Sys.getenv("DATA_DIR")
  data_dir <- file.path(data_dir, schema)
  if (!dir.exists(data_dir)) dir.create(data_dir, recursive = TRUE)

  filename <- str_glue("{year}_annual_singlefile.zip")
  t <- path.expand(file.path(raw_data_dir, filename))

  pq_path <- stringr::str_c("bls_all_", year, ".parquet")
  readr::read_csv(t, show_col_types = FALSE, guess_max = 100000) |>
    arrow::write_parquet(sink = file.path(data_dir, pq_path))
  return(TRUE)
}
```

Running this code, I see a bit of a performance pick-up, though it's not (yet) an apples-to-apples comparison because my code is simply transforming the zipped CSV files into parquet files, while the "original" code was creating CSV files based on subsets of the data. While it will turn out that this is still a fair comparison, I wonder if I can do better. For one thing, there is still about 3GB of RAM used in the process of loading data into R and then saving to parquet files one at a time.

```r
system.time(lapply(years, get_bls_data))
```

```
   user  system elapsed
265.484  19.658 142.119
```

My next approach does all the processing using DuckDB. I unzip the data, then use DuckDB's `read_csv()` piped into a direct save to parquet files, again saved in the same location.[3]

```r
get_bls_data_duckdb <- function(year, schema = "bls", data_dir = NULL) {

  if (is.null(raw_data_dir)) {
    raw_data_dir <- file.path(Sys.getenv("RAW_DATA_DIR"), schema)
  }
```

[3]So the files created in the previous step will be overwritten here.

```
  if (is.null(data_dir)) data_dir <- Sys.getenv("DATA_DIR")
  data_dir <- file.path(data_dir, schema)
  if (!dir.exists(data_dir)) dir.create(data_dir, recursive = TRUE)

  filename <- str_glue("{year}_annual_singlefile.zip")
  t <- path.expand(file.path(raw_data_dir, filename))
  csv_file <- unzip(t)

  pq_file <- stringr::str_glue("bls_all_{year}.parquet")
  pq_path <- path.expand(file.path(data_dir, pq_file))
  db <- DBI::dbConnect(duckdb::duckdb())

  sql <- stringr::str_glue("COPY (SELECT * FROM read_csv('{csv_file}')) ",
                           "TO '{pq_path}' (FORMAT parquet)")

  res <- DBI::dbExecute(db, sql)
  DBI::dbDisconnect(db)
  unlink(csv_file)
  res
}
```

How does this code do? Well, much faster! Also, much less RAM used (more like hundreds of megabytes than gigabytes).

```
system.time(lapply(2001:2019L, get_bls_data_duckdb))
```

```
   user  system elapsed
130.071  16.977  36.488
```

I said before that the comparison was a bit oranges-to-apples because I've skipped some steps from the "original" code. First, the original code applied some fields (e.g., `industry_code < 100` and `own_code %in% c(0, 5)`). I apply these filters in the code below.

First, I connect to a fresh DuckDB database (a single line of code). Second, I use `load_parquet()` from the `farr` package with a wildcard (`"bls_all_*"`) to load *all* the BLS data into a single table. I then apply the filters. As you can see, this step is *fast*. One reason it is so fast is because the `dbplyr` package I am using to connect to DuckDB uses **lazy evaluation**, so it doesn't actually realize any data in this step.

```
db <- DBI::dbConnect(duckdb::duckdb())

bls_data <-
```

```
  load_parquet(db, "bls_all_*", schema = "bls") |>
  mutate(industry_code =
           case_when(industry_code == "31-33" ~ "31",
                     industry_code == "44-45" ~ "44",
                     industry_code == "48-49" ~ "48",
                     .default = industry_code),
         industry_code = as.integer(industry_code)) |>
  filter(industry_code < 100) |>
  # and for now just keep the ownership codes for
  # - (a) total employment levels -- own_code = 0; and
  # - (b) total private sector employment -- own_code = 5
  filter(own_code %in% c(0, 5)) |>
  system_time()
```

```
   user  system elapsed
  0.138   0.006   0.151
```

I can take a quick peek at a sample of the data:

```
bls_data
```

```
# A query:  ?? x 38
# Database: DuckDB 1.4.4-dev6 [root@Darwin 25.3.0:R 4.5.2/:memory:]
   area_fips own_code industry_code agglvl_code size_code  year qtr
   <chr>        <dbl>         <int>       <dbl>     <dbl> <dbl> <chr>
 1 01000            0            10          50         0  2001 A
 2 01000            5            10          51         0  2001 A
 3 01000            5            11          54         0  2001 A
 4 01000            5            21          54         0  2001 A
 5 01000            5            22          54         0  2001 A
 6 01000            5            23          54         0  2001 A
 7 01000            5            31          54         0  2001 A
 8 01000            5            42          54         0  2001 A
 9 01000            5            44          54         0  2001 A
10 01000            5            48          54         0  2001 A
# i more rows
# i 31 more variables: disclosure_code <chr>, annual_avg_estabs <dbl>,
#   annual_avg_emplvl <dbl>, total_annual_wages <dbl>,
#   taxable_annual_wages <dbl>, annual_contributions <dbl>,
#   annual_avg_wkly_wage <dbl>, avg_annual_pay <dbl>, lq_disclosure_code <chr>,
#   lq_annual_avg_estabs <dbl>, lq_annual_avg_emplvl <dbl>,
#   lq_total_annual_wages <dbl>, lq_taxable_annual_wages <dbl>, ...
```

I next make a nifty function that takes a lazy dataframe inside DuckDB (such as `bls_data`) and saves it as a parquet file inside the data repository used to store the parquet files above.

```r
duckdb_to_parquet <- function(df, name, schema, data_dir = NULL) {

  db <- dbplyr::remote_con(df)

  if (is.null(data_dir)) data_dir <- Sys.getenv("DATA_DIR")
  data_dir <- file.path(data_dir, schema)

  table_sql <- dbplyr::sql_render(df)
  pq_path <- path.expand(file.path(data_dir,
                                   stringr::str_c(name, ".parquet")))

  sql <- stringr::str_glue("COPY ({table_sql}) ",
                           "TO '{pq_path}' (FORMAT parquet)")

  DBI::dbExecute(db, sql)
  tbl(db, stringr::str_glue("read_parquet('{pq_path}')"))
}
```

The original code created three data sets (all saved as CSV files): `bls_state`, `bls_county`, and `bls_national`. The following code chunks creates each of these in turn. Note that the `duckdb_to_parquet()` function returns a DuckDB table based on the created parquet file, so it can be examined and used in subsequent analysis. As can be seen, none of these steps takes much time. Hence the apples-to-apples aspect of the performance comparison can be restored by adding up the time taken for all the steps (under a minute given that the zipped CSV files are already on my hard drive).

```r
bls_state <-
  bls_data |>
  filter(agglvl_code > 49, agglvl_code < 60) |>
  duckdb_to_parquet(name = "bls_state", schema = "bls") |>
  system_time()
```

```
   user  system elapsed 
  3.827   0.782   0.769 
```

```r
bls_state
```

```
# A query:  ?? x 38
# Database: DuckDB 1.4.4-dev6 [root@Darwin 25.3.0:R 4.5.2/:memory:]
```

```
   area_fips own_code industry_code agglvl_code size_code  year qtr
   <chr>        <dbl>         <int>       <dbl>     <dbl> <dbl> <chr>
 1 01000            0            10          50         0  2001 A
 2 01000            5            10          51         0  2001 A
 3 01000            5            11          54         0  2001 A
 4 01000            5            21          54         0  2001 A
 5 01000            5            22          54         0  2001 A
 6 01000            5            23          54         0  2001 A
 7 01000            5            31          54         0  2001 A
 8 01000            5            42          54         0  2001 A
 9 01000            5            44          54         0  2001 A
10 01000            5            48          54         0  2001 A
# i more rows
# i 31 more variables: disclosure_code <chr>, annual_avg_estabs <dbl>,
#   annual_avg_emplvl <dbl>, total_annual_wages <dbl>,
#   taxable_annual_wages <dbl>, annual_contributions <dbl>,
#   annual_avg_wkly_wage <dbl>, avg_annual_pay <dbl>, lq_disclosure_code <chr>,
#   lq_annual_avg_estabs <dbl>, lq_annual_avg_emplvl <dbl>,
#   lq_total_annual_wages <dbl>, lq_taxable_annual_wages <dbl>, ...
```

```r
bls_county <-
  bls_data |>
  filter(agglvl_code > 69, agglvl_code < 80) |>
  duckdb_to_parquet(name = "bls_county", schema = "bls") |>
  system_time()
```

```
   user  system elapsed
 10.166   0.774   1.129
```

```r
bls_county
```

```
# A query:  ?? x 38
# Database: DuckDB 1.4.4-dev6 [root@Darwin 25.3.0:R 4.5.2/:memory:]
   area_fips own_code industry_code agglvl_code size_code  year qtr
   <chr>        <dbl>         <int>       <dbl>     <dbl> <dbl> <chr>
 1 01001            0            10          70         0  2001 A
 2 01001            5            10          71         0  2001 A
 3 01001            5            11          74         0  2001 A
 4 01001            5            21          74         0  2001 A
 5 01001            5            22          74         0  2001 A
 6 01001            5            23          74         0  2001 A
 7 01001            5            31          74         0  2001 A
 8 01001             5             42           74          0  2001 A
 9 01001             5             44           74          0  2001 A
10 01001             5             48           74          0  2001 A
# i more rows
# i 31 more variables: disclosure_code <chr>, annual_avg_estabs <dbl>,
#   annual_avg_emplvl <dbl>, total_annual_wages <dbl>,
#   taxable_annual_wages <dbl>, annual_contributions <dbl>,
#   annual_avg_wkly_wage <dbl>, avg_annual_pay <dbl>, lq_disclosure_code <chr>,
#   lq_annual_avg_estabs <dbl>, lq_annual_avg_emplvl <dbl>,
#   lq_total_annual_wages <dbl>, lq_taxable_annual_wages <dbl>, ...
```

```
bls_national <-
  bls_data |>
  filter(agglvl_code == 10) |>
  duckdb_to_parquet(name = "bls_national", schema = "bls") |>
  system_time()
```

```
   user  system elapsed
  1.088   0.088   0.210
```

```
bls_national
```

```
# A query:  ?? x 38
# Database: DuckDB 1.4.4-dev6 [root@Darwin 25.3.0:R 4.5.2/:memory:]
   area_fips own_code industry_code agglvl_code size_code  year qtr
   <chr>        <dbl>         <int>       <dbl>     <dbl> <dbl> <chr>
 1 US000            0            10          10         0  2001 A
 2 US000            0            10          10         0  2002 A
 3 US000            0            10          10         0  2003 A
 4 US000            0            10          10         0  2004 A
 5 US000            0            10          10         0  2005 A
 6 US000            0            10          10         0  2006 A
 7 US000            0            10          10         0  2007 A
 8 US000            0            10          10         0  2008 A
 9 US000            0            10          10         0  2009 A
10 US000            0            10          10         0  2010 A
11 US000            0            10          10         0  2011 A
12 US000            0            10          10         0  2012 A
13 US000            0            10          10         0  2013 A
14 US000            0            10          10         0  2014 A
15 US000            0            10          10         0  2015 A
16 US000            0            10          10         0  2016 A
```

```
17 US000             0            10          10          0  2017 A
18 US000             0            10          10          0  2018 A
19 US000             0            10          10          0  2019 A
# i 31 more variables: disclosure_code <chr>, annual_avg_estabs <dbl>,
#   annual_avg_emplvl <dbl>, total_annual_wages <dbl>,
#   taxable_annual_wages <dbl>, annual_contributions <dbl>,
#   annual_avg_wkly_wage <dbl>, avg_annual_pay <dbl>, lq_disclosure_code <chr>,
#   lq_annual_avg_estabs <dbl>, lq_annual_avg_emplvl <dbl>,
#   lq_total_annual_wages <dbl>, lq_taxable_annual_wages <dbl>,
#   lq_annual_contributions <dbl>, lq_annual_avg_wkly_wage <dbl>, ...
```

The upshot of all this is that I think this demonstrates how it is possible to have completely reproducible data steps that are also researcher-friendly and take very little computing resources to process. For example, all the BLS data created using my version of the code can be found here.[4]

```
dbDisconnect(db)
```

[4]Note that this is as far as I got in looking at the reproducibility of the steps in the original paper.