Data curation: The case of Call Reports

Ian D. Gow

29 January 2026

I recently (Gow 2026) proposed an extended version of model of the data science "whole game" of in R for Data Science (Wickham, Çetinkaya-Rundel, and Grolemund 2023). In Gow (2026), I used Australian stock price data to illustrate the data curation process and in this note, I use "call report" data for United States banks as an additional illustration of my proposed extension.

My extension of the data science "whole game"—depicted in Figure 1 below—adds a *persist* step to the original version, groups it with *import* and *tidy* into a single process, which I call *Curate*. As a complement to the new *persist* step, I also add a *load* step to the *Understand* process.[1]

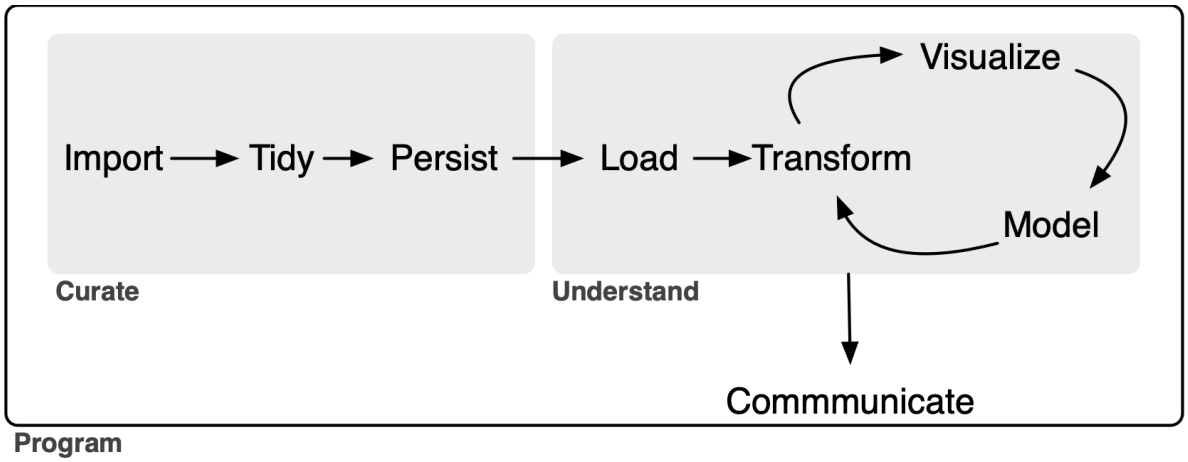


Figure 1: A representation of the data science workflow

In this note, as in Gow (2026), I focus on the data curation (*Curate*) process. My rationale for separating *Curate* from *Understand* is that I believe it clarifies certain best practices in the curation of data. the notion of a service-level agreement to delineating roles and responsibilities in the preparation and analysis of data. As will be seen, my conception of *Curate* will encompass some tasks that are included in the *transform* step (part of the *Understand* process) in R for

[1] As will be seen this, *load* step will not generally be an elaborate one. The inclusion of a separate *load* step serves more to better delineate the distinction between the *Curate* process and the *Understand* process.

Data Science. The current version of this note uses daily data on Australian stock prices as an application.

While I will argue that even the sole analyst who will perform all three processes can benefit from thinking about *Curate* separate from *Understand*, it is perhaps easiest to conceive of the *Curate* and *Understand* processes as involving different individuals or organizational units of the "whole game" of a data analysis workflow. In Gow (2026), I used the idea of a service-level agreement to delineate the division of responsibilities between the *Curate* and *Understand* teams. In effect, I will act as a self-appointed, single-person, unpaid *Curate* team and I imagine potential users of call report data as my *Understand* clients.

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 PDF version is available here.

First, in R install the `pak` package using `install.packages("pak")`. Then, use `pak` to install my package using `pak::pak("iangow/ffiec.pq")`.

```
library(ffiec.pq)
library(tidyverse)
library(farr)
library(dbplyr)
library(DBI)
```

1 Call Reports

According to its website, the Federal Financial Institutions Examination Council (FFIEC) "is an interagency body ... focused on promoting consistency in examination activities [related to United States financial institutions]. The FFIEC"does not regulate financial institutions [and its] jurisdiction is limited to prescribing uniform principles, standards, and report forms for the federal examination of financial institutions and making recommendations to promote uniformity in the supervision of financial institutions."

One of the services provide by the FFEIC is its "Central Data Repository's Public Data Distribution web site. Through this site you can obtain Reports of Condition and Income (Call Reports) for most FDIC-insured institutions. The information available on this site is updated to reflect the most recent data for both prior and current periods. The earliest data provided is from March 31, 2001."

Many academic studies use Call Report data. For example, Kashyap, Rajan, and Stein (2002, 52) states "our data come from the 'Call Reports,' the regulatory filings that all commercial banks having insured deposits submit each quarter. The Call Reports include detailed information on the composition of bank balance sheets and some additional data on off-balance-sheet items. These data are reported at the level of the individual bank."

While raw data are offered by the FFIEC, some work is required to arrange the data into a form amenable to further analysis. In other words, some curation of the data is required.

Several commercial data providers of the FFIEC data in a curated form, but accessing those requires an institutional or individual subscription. It appears that the Federal Reserve Bank of Chicago provided a curated data set on its website, but no longer does so.

Other services provide "API wrappers" that allow users to access the data via the FFIEC API, but this approach is best suited to real-time feeds of small amounts of data. Collation of the data into a single repository using this approach seems unfeasible.

Regardless of the availability of paid curation services, in this note I will imagine that such sources either do not exist or are unavailable to my hypothetical *Understand* clients and illustrate how one can curate Call Reports data from the FFIEC source.

1.1 Getting the raw data

The FFIEC [Bulk Data Download site] provides the Call Report data in two forms. The first is as zipped tab-delimited data files, one for each quarter. The second is as zipped XBRL data files, one for each quarter. At the time of writing, getting the complete data archive amounts to point-and-clicking to download each of the 99 files for each format.

The FFIEC data sets are not as amenable to automated downloading as those offered by other government agencies such as the SEC (see my earlier note on XBRL data), the PCAOB (see my note on Form AP data), or even the Federal Reserve itself (I used data from the MDRM site in preparing this note). However, a group of individuals has contributed the Python package `ffiec_data_collector` that we can use to collect the data.[2]

To install this Python package, you first need to install Python and then install the `ffiec_data_collector` using a command like `pip install ffiec_data_collector`.

As discussed in Appendix E of Gow and Ding (2024), I organize my raw and processed data in a repository comprising a single parent directory and several sub-directories corresponding to various data sources and projects. For some data sets, this approach to organization facilitates switching code from using (say) data sources provided by Wharton Research Data Services (WRDS) to using local data in my data repository. I will adopt that approach for the purposes of this note.

As the location of my "raw data" repository is found in the the environment variable `RAW_DATA_DIR`, I can identify that location in Python easily. The following code specifies the download directory as the directory `ffiec` within my raw data repository.[3]

[2] I discovered this package after writing most of this note. In my case, I pointed-and-clicked to get many of the files and Martien Lubberink of Victoria University of Wellington kindly provided the rest.

[3] I recommend that readers follow a similar approach if following along with note, as it makes subsequent steps easier to implement. A reader can simply specify `os.environ['RAW_DATA_DIR'] = "/Users/igow/Dropbox/raw_data"`, substituting a location where the data should go on his or her computer.

```
import os
from pathlib import Path
print(os.environ['RAW_DATA_DIR'])
```

```
/Users/igow/Dropbox/raw_data
```

```
download_dir = Path(os.environ['RAW_DATA_DIR']) / "ffiec"
```

Having specified a location to put the downloaded files, it is a simple matter to adapt a script provided on the package website to download the raw data files.

```
import ffiec_data_collector as fdc
import time

downloader = fdc.FFIECDownloader(download_dir=download_dir)

periods = downloader.select_product(fdc.Product.CALL_SINGLE)

results = []
for period in periods[:4]:

    print(f"Downloading {period.yyyymmdd}...", end=" ")
    result = downloader.download(
        product=fdc.Product.CALL_SINGLE,
        period=period.yyyymmdd,
        format=fdc.FileFormat.TSV
    )
    results.append(result)

    if result.success:
        print(f"  ({result.size_bytes:,} bytes)")
    else:
        print(f"  Failed: {result.error_message}")

    # IMPORTANT: Be respectful to government servers
    # Add delay between requests to avoid overloading the server
    time.sleep(1)  # 1 second delay - adjust as needed
```

```
Downloading 20250930...   (5,686,532 bytes)
Downloading 20250630...   (6,231,006 bytes)
Downloading 20250331...   (5,704,421 bytes)
Downloading 20241231...   (6,605,092 bytes)
```

```
# Summary
successful = sum(1 for r in results if r.success)
print(f"\nCompleted: {successful}/{len(results)} downloads")
```

```
Completed: 4/4 downloads
```

Note that the code above downloads just the most recent four files available on the site. Remove `[:4]` from the line `for period in periods[:4]:` to download *all* files. Note that the package website recommends using `time.sleep(5)` in place of `time.sleep(1)` to create a five-second delay and this may be a more appropriate choice if you are downloading all 99 files using this code. Note that the downloaded files occupy about 800 MB of disk space, so make sure you have that available if running this code.

1.2 Processing the data

With the raw data files on hand, the next task it to process these into files useful for analysis. For reasons I will discuss below, I will process the data into Parquet files. The Parquet format is described in *R for Data Science* (Wickham, Çetinkaya-Rundel, and Grolemund 2023, 393) as "an open standards-based format widely used by big data systems." Parquet files provide a format optimized for data analysis, with a rich type system. More details on the parquet format can be found in Chapter 22 of Wickham, Çetinkaya-Rundel, and Grolemund (2023) and every code example in Gow and Ding (2024) can be executed against Parquet data files created using my `db2pq` Python library as described in Appendix E of that book.

The easiest way to run the code I used to process the data is to install the `ffiec.pq` R package I have made available on GitHub.

The easiest way to run the package is to set the locations for the downloaded raw data files from above and for the processed data using the environment variables `RAW_DATA_DIR` and `DATA_DIR`, respectively.

I already have these set:

```
Sys.getenv("RAW_DATA_DIR")
```

```
[1] "/Users/igow/Dropbox/raw_data"
```

```
Sys.getenv("DATA_DIR")
```

```
[1] "/Users/igow/Dropbox/pq_data"
```

If you don't have these set, you can set them within R using commands like the following. You should set `RAW_DATA_DIR` to match what you used above in Python and you should set `DATA_DIR` to be where you want to put the processed files. The processed files will occupy about 2 GB of disk space, so make sure you have room for these there.

```
Sys.setenv(RAW_DATA_DIR="/Users/igow/Dropbox/raw_data")
Sys.setenv(DATA_DIR="/Users/igow/Dropbox/pq_data")
```

Having set these environment variables, we can load my package and run a single command `ffiec_process()` without any arguments to process *all* the raw data files. This takes a bit over three minutes for me:

```
results <- ffiec_process() |> system_time()
```

```
   user  system elapsed
206.273  93.055 195.449
```

So what have we just done? Some answers come from inspection of the data frame `results` returned by the `ffiec_process()`.

```
results
```

```
# A tibble: 3,674 x 9
   type     kind  date       parquet   zipfile n_parts repairs inner_files ok
   <chr>    <chr> <date>     <chr>     <chr>     <dbl> <list>  <list>      <lgl>
 1 por      por   2001-03-31 por_2001~ FFIEC ~       1 <chr>   <chr [1]>   TRUE
 2 schedule ci    2001-03-31 ci_20010~ FFIEC ~       1 <chr>   <chr [1]>   TRUE
 3 schedule ent   2001-03-31 ent_2001~ FFIEC ~       1 <chr>   <chr [1]>   TRUE
 4 schedule gi    2001-03-31 gi_20010~ FFIEC ~       1 <chr>   <chr [1]>   TRUE
 5 schedule gl    2001-03-31 gl_20010~ FFIEC ~       1 <chr>   <chr [1]>   TRUE
 6 schedule leo   2001-03-31 leo_2001~ FFIEC ~       1 <chr>   <chr [1]>   TRUE
 7 schedule narr  2001-03-31 narr_200~ FFIEC ~       1 <chr>   <chr [1]>   TRUE
 8 schedule rc    2001-03-31 rc_20010~ FFIEC ~       1 <chr>   <chr [1]>   TRUE
 9 schedule rca   2001-03-31 rca_2001~ FFIEC ~       1 <chr>   <chr [1]>   TRUE
10 schedule rcb   2001-03-31 rcb_2001~ FFIEC ~       1 <chr>   <chr [1]>   TRUE
# i 3,664 more rows
```

Each row in `results` represents a Parquet file created by `ffiec_process()`.

```
results |>
  count(type)
```

```
# A tibble: 2 x 2
  type         n
  <chr>    <int>
1 por         99
2 schedule  3575
```

Let's use DuckDB.

The `results` data frame is also stored as `ffiec_process_data` in the `ffiec` directory.

No doubt you are thinking "OMG! There are 3674 files? How can I use these?" It turns out that this structure is quite easy to work with given the right tools.

1.3 Using the data with R

Most of the files are listed as having `type` equal to `"schedule"` and the rest have `type` equal to `"por"`. The latter files represent the "Panel of Reporters" (POR) data and they provides details on the financial institutions filing in the respective quarter. **TODO:** Need to mention that the data are quarterly.

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

```
results <-
  load_parquet(db, "ffiec_process_data", "ffiec") |>
  system_time()
```

```
   user  system elapsed
  0.003   0.000   0.003
```

To start with let's focus on the last available date, which is 2025-09-30 at the time of writing.

We can access the latest POR file using the `load_parquet()` function from my `farr` package.

```
por <- load_parquet(db, "por_20250930", "ffiec")
por
```

```
# A query:  ?? x 13
# Database: DuckDB 1.4.4-dev6 [root@Darwin 25.3.0:R 4.5.2/:memory:]
   IDRSSD fdic_certificate_number occ_charter_number ots_docket_number
    <int> <chr>                  <chr>               <chr>
 1     37 10057                  <NA>                16553
 2    242 3850                   <NA>                <NA>
 3    279 28868                  <NA>                2523
 4    354 14083                  <NA>                <NA>
 5    457 10202                  <NA>                <NA>
 6    505 6959                   1253                <NA>
 7   1155 17639                  <NA>                <NA>
 8   1351 9392                   <NA>                <NA>
 9   1454 19184                  15359               <NA>
10   1669 3384                   13541               <NA>
# i more rows
# i 9 more variables: primary_aba_routing_number <chr>,
#   financial_institution_name <chr>, financial_institution_address <chr>,
#   financial_institution_city <chr>, financial_institution_state <chr>,
#   financial_institution_zip_code <chr>,
#   financial_institution_filing_type <chr>,
#   last_date_time_submission_updated_on <dttm>, date <date>
```

The remaining files with `type` equal to `"schedule"` represent files on various schedules and the applicable schedule is indicated by the column `kind`.

If we were experts in Call Report data, we might know that domestic total assets is reported as item `RCFD2170` on Schedule RC for banks reporting on a consolidated basis and as item `RCON2170` for banks reporting on an unconsolidated basis.[4] We can load the latest Schedule RC data by combining the schedule as lower case (`rc`) with an underscore followed by the date in `yyyymmdd` form. **TODO: Guide readers to this point.**

We can make a balance sheet table using the following code.

```
rc_20250930 <- load_parquet(db, "rc_20250930", "ffiec")

bs_data <-
  rc_20250930 |>
  mutate(total_assets = coalesce(RCFD2170, RCON2170)) |>
  select(IDRSSD, date, total_assets) |>
  system_time()
```

```
   user  system elapsed
  0.005   0.000   0.005
```

[4] From inspection of the data, we can confirm that banks report one or the other.

Let's say we're interested in the top 10 banks by total assets on this date. This is easy enough to produce.

```
top_5 <-
  bs_data |>
  window_order(desc(total_assets)) |>
  mutate(ta_rank = row_number()) |>
  filter(ta_rank <= 5) |>
  system_time()
```

```
   user  system elapsed
  0.066   0.001   0.068
```

```
top_5 |> collect()
```

```
# A tibble: 5 x 4
  IDRSSD date       total_assets ta_rank
   <int> <date>            <dbl>   <dbl>
1 852218 2025-09-30   3813431000       1
2 480228 2025-09-30   2651090000       2
3 476810 2025-09-30   1844189000       3
4 451965 2025-09-30   1767105000       4
5 504713 2025-09-30    679293260       5
```

However, it's hard for us to know who these banks if we haven't committed the table of `IDRSSD` values to memory. Fortunately, we can get the information we need from `por`.

```
top_5_names <-
  top_5 |>
  inner_join(por |>
               select(IDRSSD, date, financial_institution_name),
             join_by(IDRSSD, date)) |>
  arrange(ta_rank) |>
  select(IDRSSD, financial_institution_name, ta_rank) |>
  rename(bank = financial_institution_name) |>
  compute() |>
  system_time()
```

```
   user  system elapsed
  0.032   0.001   0.032
```

```
top_5_names |> collect()
```

```
# A tibble: 5 x 3
  IDRSSD bank                                        ta_rank
   <int> <chr>                                         <dbl>
1 852218 JPMORGAN CHASE BANK, NATIONAL ASSOCIATION       1
2 480228 BANK OF AMERICA, NATIONAL ASSOCIATION           2
3 476810 CITIBANK, N.A.                                  3
4 451965 WELLS FARGO BANK, NATIONAL ASSOCIATION          4
5 504713 U.S. BANK NATIONAL ASSOCIATION                  5
```

Already you might be feeling frustrated. "I don't want to access data for just one quarter. Do I have to make tables for every quarter like this?"

The answer is "no".

In principle the `load_parquet()` function can take **glob** entries, such as this:

```
rc <- load_parquet(db, "rc_*", "ffiec")
```

But, while that appears to work, we will quickly run into trouble because the items that appear on any given schedule can vary over times. Fortunately, we can use the function `ffiec_scan_pqs()` to combine all schedules into a single table that we can use easily.

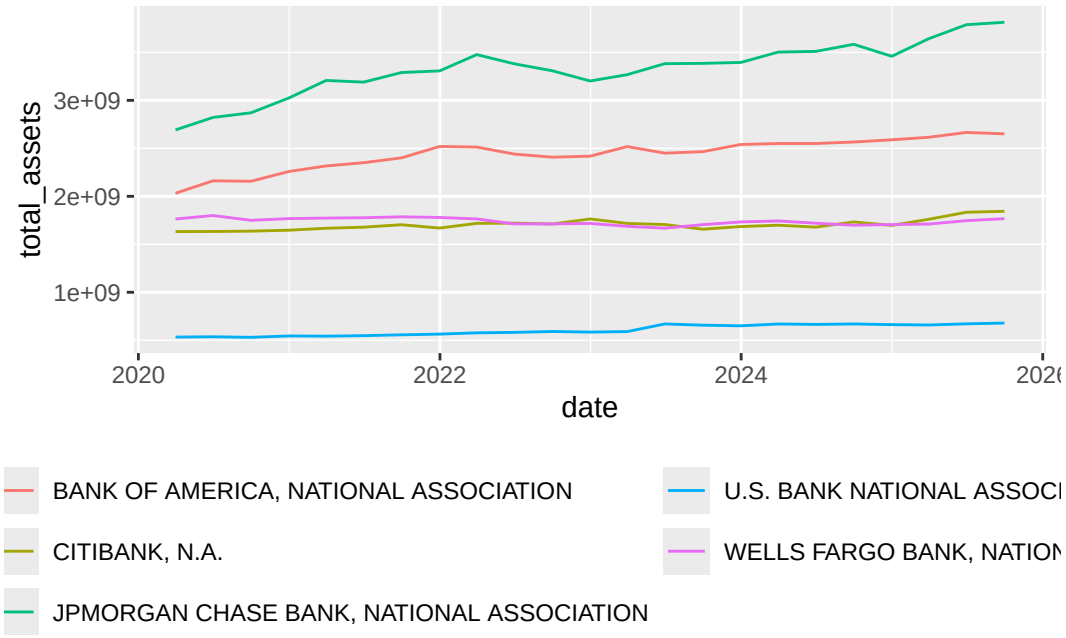
```
rc <- ffiec_scan_pqs(db, schedule = "rc") |> system_time()
```

```
   user  system elapsed
  0.036   0.028   0.031
```

```
bs_panel_data <-
  rc |>
  mutate(total_assets = coalesce(RCFD2170, RCON2170)) |>
  select(IDRSSD, date, total_assets) |>
  inner_join(top_5_names, join_by(IDRSSD)) |>
  collect() |>
  system_time()
```

```
   user  system elapsed
  0.074   0.036   0.059
```

```r
bs_panel_data |>
  filter(date >= "2020-01-01") |>
  ggplot(aes(x = date, y = total_assets, colour = bank)) +
  geom_line() +
  theme(legend.position = "bottom") +
  guides(colour = guide_legend(nrow = 3))
```



1.4 Using the data with Python

2 The boring details

2.1 Reading the data

Each quarter's zip file (`zipfile`) actually contains dozens of text files (`.txt`) in TSV ("tab-separated values") form. The TSV is a close relative of the CSV ("comma-separated values") and the principles applicable to one form apply to the other.

I have seen code that just imports from these individual files (what I call `inner_file`) in some location on the user's hard drive. This approach is predicated on the user having downloaded the zip files and unzipped them. While we have downloaded all the zip files—assuming you followed the steps outlined in Section 1.1—I don't want to be polluting my hard drive with thousands of `.txt` files that I won't use after reading them once.

Instead, R allows me to do simply say

```
con <- unz(zipfile, inner_file)
```

The resulting `con` object is a temporary read-only connection to a single text file (`inner_file`) stored inside the zip file `zipfile`. The object allows R to stream the file's contents directly from the zip archive, line by line, without extracting it. Given `con`, the core function used to read the data into R has the following basic form, where `read_tsv()` comes from the `readr` library, part of the Tidyverse.

```
df <- read_tsv(
  con,
  col_names = cols,
  col_types = colspec,
  skip = skip,
  quote = "",
  na = c("", "CONF"),
  progress = FALSE,
  show_col_types = FALSE
)
```

2.1.1 Handling embedded newlines and tabs

Experienced uses of the `readr` library might wince a little at the `quote = ""` argument above. What this means is that the data are not quoted.

Wickham, Çetinkaya-Rundel, and Grolemund (2023, 101) points out that "sometimes strings in a CSV file contain commas. To prevent them from causing problems, they need to be surrounded by a quoting character, like `"` or `'`. By default, `read_csv()` assumes that the quoting character will be `"`."

Adapting this to our context and expanding it slightly, I would say: "sometimes strings in a TSV file contain tabs (`\t`) and newline characters (`\n`).[5] To prevent them from causing problems, they need to be surrounded by a quoting character, like `"` or `'`." While this is a true statement, the TSV files provided on the FFIEC Bulk Data website are *not* quoted, which means that tabs and newlines characters **embedded** in strings *will* cause problems.

The approach taken by the `ffiec.pq` package is to attempt to read the data using a call like that above, which I term the "fast path" (in part because it is indeed fast). Before making that call, the code has already inspected the first row of the file to determine the column names (stored in

[5] Tabs and newlines are what are sometimes called **invisibles** because there presence is not apparent from viewing their usual representation as text (for example, a tab might look the same as a series of spaces). The `\t` and `\n` representations are quite standard ways of making these characters visible to humans.

`cols`) and used those column names to look up the appropriate type for each column (stored in `colspecs`). Any anomaly caused by embedded newlines or embedded tabs will almost certainly cause this first `read_tsv()` call to fail. But if there are no issues, then we pretty much have the data as we want them and can return `df` to the calling function.[6] Fortunately, over 95% of files can be read successfully on the "fast path".

It turns out that if the "fast path" read fails, the most likely culprit is **embedded newlines**. Let's say the table we're trying to read has seven columns and the text field that is the fourth field in the file contains, in some row of the data, an embedded newline, because the data submitted by the reporting financial institution contained `\n` in that field. Because `read_tsv()` processes the data line by line and lines are "delimited" by newline characters (`\n`), it will see the problematic line as terminating part way through the fourth column and, because `cols` tells `read_tsv()` to expect seven columns, `read_tsv()` will issue a warning.

When a warning occurs on the "fast path", the read function in `ffiec.pq` moves to what I call (unimaginatively) the "slow path". A "feature" (it turns out) of the TSV files provided on the FFIEC Bulk Data website is that each line ends with not just `\n`, but `\t\n`. This means we can assume that any `\n` not preceded by `\t` is an embedded newline, not a line-terminating endline.[7] So I can read the data into the variable `txt` using `readLines()` and use a regular expression to replace embedded newlines with an alternative character. The alternative I use is a space and I use the `gsub()` function to achieve this: `gsub("(?<!\\t)\\n", " ", txt, perl = TRUE)` The regular expression here is `(?<!\\t)\\n` is equivalent to `(?<!\t)\n` in Perl or `r"(?<!\t)\n"` in Python.[8] In words, the regular expression literally means "any newline character that is not immediately preceded by a tab" and the `gsub()` function will replace such characters with spaces (`" "`) because that is the second argument to `gsub()`.

This fix addresses almost all the problematic files. "Almost all" means "all but two". The issue with the remaining two files is (as you might have guessed) **embedded tabs**. Unfortunately, there's no easy "identify the embedded tabs and replace them" fix, because there's no easy way to distinguish embedded tabs from delimiting tabs. However, we can detect the presence of embedded tabs because there will one too many tabs in any affected row.

For one of the two "bad" files, there is only one text field, so once we detect the presence of the embedded tab, we can assume that the extra tab belongs in that field: Problem solved. For the other "bad" file, there are several text fields and, while there is just one bad row, we cannot be sure which text field has the embedded tab. The `ffiec.pq` package just assumes that the embedded tab belongs in the last field and moves on. This means that the textual data for one row (i.e., one financial institution) for one schedule for one quarter cannot be guaranteed to be completely correct.[9] Such is life.

---

[6] In practice, there's a little clean-up to be done before returning `df`, as I will explain shortly.

[7] Unfortunately, the `read_tsv()` function does not allow use to specify an alternative to the default for line-terminating characters.

[8] There are extra backslashes in the R version to specify that the backslashes represent backslashes to be passed along to `gsub()`, not special characters to be interpreted by R itself.

[9] This is not a completely insoluble problem in that we could inspect the XBRL data to determine the correct form of the data in this case. This is "above my pay grade" in this setting. (I'm not being paid to do this!)

Even without addressing the issue, given that only two files are affected, it's possible to measure the "damage" created by embedded tabs.

Let's look at the earlier file, which turns out to affect the Schedule RIE data for The Traders National Bank (`IDRSSD` of `490937`) for June 2004.[10] Traders National Bank in Tennessee was "Tullahoma's second oldest bank", until changing its name to Wellworth Bank (seemingly after some mergers), and it listed total assets of $117,335,000 in the affected Call Report.

Let's look at the textual data we have in our Parquet files for this case:[11]

```
rie <- ffiec_scan_pqs(db, schedule = "rie", schema = "ffiec")

rie |>
  filter(IDRSSD == "490937", date == "2004-06-30") |>
  select(IDRSSD, starts_with("TEXT")) |>
  pivot_longer(!IDRSSD) |>
  filter(!is.na(value)) |>
  collect() |>
  system_time()
```

```
   user  system elapsed
  0.794   0.521   0.696
```

```
# A tibble: 4 x 3
  IDRSSD name     value
   <int> <chr>    <chr>
1 490937 TEXT4464 ATM Fees, Exam Fees, Dues & Chairitable Cont
2 490937 TEXT4467 Telephone, BanClub, Federal Res Fees, NSF and Other Loss
3 490937 TEXT4468 Courier, Audit Tax, Deff Comp, Other
4 490937 TEXT4469 ns Exp, Bus Dev, P
```

We can compare this with what we see in the Call Report in Figure 2, where we see that the value of `TEXT4468` should be something like `"Courier, Audit Tax, Deff Comp, Other\tns Exp, Bus Dev, P"`. The embedded tab has split this into `"Courier, Audit Tax, Deff Comp, Other"` for `TEXT4468` and `"ns Exp, Bus Dev, P"` for `TEXT4469`, which should be `NA`. If the values for `TEXT4468`and `TEXT4469` for Traders National Bank in June 2004 are important to your analysis, you could fix this "by hand" easily enough.

Looking at the later file, there were embedded tabs in two rows of Schedule NARR for December 2022. I compared the values in the Parquet file with those in the Call Reports for the two affected

[10] Note that it would be somewhat faster to use `rie <- load_parquet(db, "rie_20040630", "ffiec")`, but this code doesn't take too long to run.

[11] Only the textual data will have embedded tabs and, because such data is arranged after numerical data, only text data will be affected by embedded tabs.

(TEXT4464) ATM Fees, Exam Fees, Dues & Chairitable Cont

(TEXT4467) Telephone, BanClub, Federal Res Fees, NSF and Other Loss

(TEXT4468) Courier, Audit Tax, Deff Comp, Other ns Exp, Bus Dev, P

Figure 2: Extract from June 2004 Call Report for The Traders National Bank

banks and the values in the Parquet file match perfectly.[12] Because Schedule NARR ("Optional Narrative Statement Concerning the Amounts Reported in the Consolidated Reports of Condition and Income") has just one text column (`TEXT6980`), the fix employed by the `ffiec.pq` package will work without issues.[13]

2.1.2 Handling missing-value sentinels

Users familiar with both `readr` and Call Report data might also have noticed the use of `na = c("", "CONF")` in the call to `read_tsv()` above. The default value for this function is `na = c("", "NA")` means that empty values and the characters `NA` are treated as missing values. As I saw no evidence that the export process for FFIEC Bulk Data files used `"NA"` to mark `NA` values, I elected not to treat `"NA"` as `NA`. However, a wrinkle is that the reporting firms some times populate text fields—but not numerical fields—with the value `"NA"`.[14] While the most sensible interpretation of such values is as `NA`, without further investigation it is difficult to be sure that `"NA"` is the canonical form in which firms reported `NA` values rather than `"N/A"` or `"Not applicable"` or some other variant.

This approach seems validated by the fact that I see the value `"NR"` in text fields of PDF versions of Call Reports and these values show up as empty values in the TSV files, suggesting that `"NR"`, not `"NA"` is the FFIEC's canonical way of representing `NA` values in these files, while `"NA"` is literally the text value `"NA"`, albeit perhaps one intended *by the reporting firm* to convey the idea of `NA`. Users of the FFIEC data created by the `ffiec.pq` package who wish to use textual data should be alert to the possibility that values in those fields may be intended by the reporting firm to convey the idea of `NA`, even if they are not treated as such by the FFIEC's process for creating the TSV files.

The other value in the `na` argument used above is `"CONF"`, which denotes that the the reported value is confidential and therefore not publicly disclosed. Ideally, we might distinguish between

[12] See here for the gory details. Interestingly, the WRDS Call Report data have the same issue with the earlier case and have incorrect data for one of the banks in the latter case. This seems to confirm that WRDS uses the TSV data itself in creating its Call Report data sets.

[13] Recall that the "fix" assumes that embedded tab belongs in last available text column.

[14] If `"NA"` appeared in a numeric field, my code would report an error. As I detected no errors in importing the data, I know there are no such values.

`NA`, meaning "not reported by the firm to the FFIEC" or "not applicable to this firm" or things like that, from `"CONF"`, meaning the FFIEC has the value, but we do not. Unfortunately, the value `"CONF"` often appears in numeric fields and there is no simple way to ask `read_tsv()` to record the idea that "this value is confidential", so I just read these in as `NA`.[15]

I say "no simple way" because there are probably workarounds that allow `"CONF"` to be distinguished from true `NA`s. For example, I could have chosen to have `read_tsv()` read all numeric fields as character fields and then convert the value `CONF` in such fields to a **sentinel value** such as `Inf` (R's way of saying "infinity" or ∞).[16] This would not be terribly difficult, but would have the unfortunate effect of surprising users of the data who (understandably) didn't read the manual and starting finding that the mean values of some fields are `Inf`. Perhaps the best way to address this would allow the user of `ffiec.pq` to *choose* that behaviour as an option, but I did not implement this feature at this time.

In addition to these missing values, I discovered in working with the data that the FFIEC often used specific values as **sentinel values** for `NA`. For example, `"0"` is used for some fields, while `"00000000"` is used to mark dates as missing, and `"12/31/9999 12:00:00 AM"` is used for timestamps. I recoded such sentinel values as `NA` in each case.

3 The service-level agreement

Nothing from here up has been updated from original note

It is perhaps helpful to think of dividing the data science workflow into processes with different teams being responsible for the different processes. From this perspective, the *Curate* team manufactures data that are delivered the *Understand* team. While I won't discuss transfer pricing (i.e., how much the *Understand* team needs to pay the *Curate* team for the data), we might consider the analogy of a service-level agreement between the two teams.

One template for a service-level agreement would specify that data from a particular source will be delivered to the *Understand* team with the following conditions:

1. The data will be presented as a set of tables in a modern storage format.
2. The division into tables will adhere to a pragmatic version of good database principles.
3. The **primary key** of each table will be identified and validated.
4. Each variable (column) of each table will be of the correct type.
5. There will be no manual steps that cannot be reproduced.
6. A process for updating the curated data will be established.
7. The entire process will be documented in some way.
8. Some process for version control of data will be maintained.

[15] This is the kind of situation where SAS's approach to coding missing values would be helpful.

[16] In a sense, this would be doing the opposite of what the Python package pandas did in treating `np.NaN` as the way of expressing what later became `pd.NA`; I'd be using `Inf` to distinguish different kinds of missing values.

3.1 Storage format

In principle, the storage format should fairly minor detail determined by the needs of the *Understand* team. For example, if the *Understand* team works in Stata or Excel, then perhaps they will want the data in some kind of Stata format or as Excel files. However, I think it can be appropriate to push back on notions that data will be delivered in form that involves downgrading the data or otherwise compromises the process in a way that may ultimately add to the cost and complexity of the task for the *Curate* team. For example, "please send the final data as an Excel file attachment as a reply email" might be a request to be resisted because the process of converting to Excel can entail the degradation of data (e.g., time stamps or encoding of text).[17] Instead it may be better to choose a more robust storage format and supply a script for turning that into a preferred format.

One storage format that I have used in the past would deliver data as tables in a (PostgreSQL) database. The *Understand* team could be given access data from a particular source organized as a **schema** in a database. Accessing the data in this form is easy for any modern software package. One virtue of this approach is that the data might be curated using, say, Python even though the client will analyse it using, say, Stata.[18]

3.2 Good database principles

I included the word "pragmatic" because I think it's not necessary in most cases to get particularly fussy about database normalization. That said, it's probably bad practice to succumb to requests for One Big Table that the *Understand* team might make. It is reasonable to impose some obligation to merge tables that are naturally different tables on the client *Understand* team.

3.3 Primary keys

The *Curate* team should communicate the primary key of each table to the *Understand* team.[19] A primary key of a table will be a set of variables that can be used to uniquely identify a row in that table. In general a primary key will have no missing values. Part of data curation will be confirming that a proposed primary key is in fact a valid primary key.

3.4 Data types

Each variable of each table should be of the correct type. For example, dates should be of type `DATE`, variables that only take integer values should be of `INTEGER` type.[20] Date-times should

[17] I discuss some of the issues with Excel as a storage format below.

[18] One project I worked on involved Python code analysing text and putting results in a PostgreSQL database and a couple of lines of code were sufficient for a co-author in a different city to load these data into Stata.

[19] Sometimes there will be more than one primary key for a table.

[20] Here I used PostgreSQL data types, but the equivalent types in other formats should be fairly clear.

generally be given with `TIMESTAMP WITH TIME ZONE` type. Logical columns should be supplied with type `BOOLEAN`.

Note that there is an interaction between this element of the service-level agreement and the storage format. If the data are supplied in a PostgreSQL database or as parquet files, then it is quite feasible to prescribe the data types of each variable. But if the storage format is Excel files (not recommended!) or CSV files, then it is difficult for the data curator to control how each variable is understood by the *Understand* team.[21]

In some cases, it may seem unduly prescriptive to specify the types in a particular way. For example, a logical variable can easily be represented as `INTEGER` type (`0` for `FALSE`, `1` for `TRUE`). Even in such cases, I think there is merit in choosing the most logical type (no pun intended) because of the additional information it conveys about the data. For example, a logical type should be checked to ensure that it only takes two values (`TRUE` or `FALSE`) plus perhaps `NULL` and that this checking has occurred is conveyed by the encoding of that variable as `BOOLEAN`.

3.5 No manual steps

When data vendors are providing well-curated data sets, much about the curation process will be obscure to the user. This makes some sense, as the data curation process has elements of trade secrets. But often data will be supplied by vendors in an imperfect state and significant data curation will be performed by the *Curate* team working for or within the same organization as the *Understand* team.

Focusing on the case where the data curation process transforms an existing data set—say, one purchased from an outside vendor—into a curated data set in sense used here, there are a few ground rules regarding manual steps.

"First, *the original data files should not be modified in any way*." Correct. I don't even touch the files after downloading them. I do make some corrections to the `item_name` variable in the `ffiec_items` package, but these "manual steps [are] extensively documented and applied in a transparent, automated fashion." The code for these steps can be found on the GitHub page for the `ffiec.pq` package.

3.6 Documentation

"The process of curating the data should be documented sufficiently well that someone else could perform the curation steps should the need arise." I regardin having a package that does the work satisfies this requirement.

[21] SAS and Stata are somewhat loose with their "data types". In effect SAS has just two data types—fixed-width character and floating-point numeric—and the other types are just formatting overlays over these. These types can be easily upset depending on how the data are used.

3.7 Update process

If a new zip file appears on the FFIEC Bulk Data website, you can download it using the process outlined in Section 1.1. Just changing the `[:4]` to `[0]` and the script downloads the latest file.

Then run

```
library(ffiec.pq)
library(tidyverse)
library(farr)
library(dbplyr)
library(DBI)
```

```
library(ffiec.pq)
zipfiles <- ffiec_list_zips()

results <-
  zipfiles |>
  filter(date == max(date)) |>
  select(zipfile) |>
  pull() |>
  ffiec_process() |>
  system_time()
```

3.8 Data version control

Welch (2019) argues that, to ensure that results can be reproduced, "the author should keep a private copy of the full data set with which the results were obtained." This imposes a significant cost on the *Understand* team to maintain archives of data sets that may run to several gigabytes or more and it would seem much more efficient for these obligations to reside with the parties with the relevant expertise.

Unfortunately, even when data vendors provide curated data sets, they generally provide little in the way of version control. For example, there is no evidence that Wharton Research Data Services (WRDS), perhaps the largest data vendor in academic business research, provides any version control for its datasets, even though it should have much greater expertise for doing this than the users of its services.

Nonetheless, some notion of version control of data probably has a place in data curation, even if this is little more than archiving of various versions of data supplied to research teams.

4 The service-level agreement revisited

I now return to our service-level agreement (SLA) to take stock of where we are after the above. Given that much of the focus above was on data types, I do not revisit that here and instead focus on those elements of the SLA that I did not address above.

4.1 Storage format

We have chosen to use parquet files for our output. **?@tbl-pq-files** provides some data on the parquet files we have produced for our hypothetical client (the *Understand* team). Assuming that the client is a group of colleagues at an institution with access to SIRCA, we (the *Curate* team) might just send a link to the Dropbox folder where we have stored the parquet files.

4.2 Primary keys

4.3 Good database principles

In general, I think one wants to be fairly conservative in considering database principles with a data library. If the data are workable and make sense in the form they come in, then it may make most sense to keep them in that form.

The SIRCA ASX EOD data are organized into four tables with easy-to-understand primary keys and a fairly natural structure. At some level, the two primary tables are `si_au_ref_names` and `si_au_prc_daily`.[22] These two tables are naturally distinct, with one about companies and the other about daily security returns.

While there might be merit in splitting `si_au_prc_daily` into separate tables to reduce its size, it is actually quite manageable in its current form.

4.4 No manual steps

There are no manual steps in creating the parquet files except for the initial download of the CSV files from SIRCA. While some data vendors allow users to download files using scripts (e.g., the scripts I have here for WRDS), this does not appear to be an option for SIRCA. But once the data have been downloaded, the subsequent steps are automatic.

While some of the checks and data-cleaning had manual elements (e.g., identifying the near-duplicate with `Gcode=="rgwb1"` in `si_au_ref_names`), the resulting code implements the fix in an automated fashion. So long as the SIRCA data remain unchanged, the fix will continue to work.

[22] It seems possible that `si_au_retn_mkt` and `si_au_ref_trddays` are generated from `si_au_prc_daily`.

4.5 Documentation

A important principle here is that the code for processing the data is documentation in its own right. Beyond that the document you are reading now is a form of documentation. If the goal of this document were to provide details explaining the process used to produce the final data sets, then it might make sense to edit this document to reflect that different purpose, but in many ways I hope this document already acts as good documentation.

4.6 Update process

In some ways, the update process is straightforward: when new CSV files become available, download them into `RAW_DATA_DIR` and run the script. However, it would probably be necessary to retrace some of the steps above to ensure that no data issues have crept in (e.g., duplicated keys). It may make sense to document the update process as part of performing it the first time.

4.7 Data version control

I achieve a modest level of data version control by using Dropbox, which offers the ability to restore previous versions of data files. As discussed earlier, version control of data is a knotty problem.

References

Gow, Ian D. 2026. "Data Curation and the Data Science Workflow." https://papers.ssrn.com/sol3/papers.cfm?abstract_id=6149488.

Gow, Ian D., and Tongqing Ding. 2024. *Empirical Research in Accounting: Tools and Methods*. Chapman & Hall/CRC. https://doi.org/10.1201/9781003456230.

Kashyap, Anil K., Raghuram Rajan, and Jeremy C. Stein. 2002. "Banks as Liquidity Providers: An Explanation for the Co-Existence of Lending and Deposit-Taking." *Journal of Finance* 57 (3): 33–63. https://doi.org/10.1111/1540-6261.00415.

Welch, Ivo. 2019. "Editorial: An Opinionated FAQ." *Critical Finance Review* 8 (1-2): 19–24. https://doi.org/10.1561/104.00000077.

Wickham, Hadley, Mine Çetinkaya-Rundel, and Garrett Grolemund. 2023. *R for Data Science*. Sebastopol, CA: O'Reilly Media. https://books.google.com/books?id=TiLEEAAAQBAJ.