Working with JSON data: The case of SEC submissions data

Ian D. Gow

19 July 2024

One of the goals of *Empirical Research in Accounting: Tools and Methods* (co-authored with Tony Ding) is to provide a pathway to mastery of the contents of *R for Data Science*. That is, by working through *Empirical Research in Accounting: Tools and Methods* (along with the occasional detour to *R for Data Science*) the reader will achieved a level of proficiency at or above that provided by studying *R for Data Science*.[1]

While we do cover most of the content of *R for Data Science*, a few gaps remain: spreadsheets, dates and times, and hierarchical data are three notable ones. The omission of spreadsheets is not accidental given our focus on academic research applications.[2] While we do discuss dates extensively in *Empirical Research in Accounting: Tools and Methods*, we have no application involving date-times and thus our coverage of the the additional complexities of dates and times discussed in *R for Data Science*—such as time zones—is limited.

Regarding hierarchical data, *Empirical Research in Accounting: Tools and Methods* does touch on some key ideas. Chapter 5 and Chapter 16 discuss `unnest_wider()` and list-columns are used in a number of places, including Chapter 10. However, we do not provide coverage of these topics as systematic as that of Chapter 23 of *R for Data Science*. For example, we do not touch `unnest_longer()` at all.

The purpose of this note is three-fold. First, I hope to provide a hands-on introduction to hierarchical data that complements and builds on Chapter 23 of *R for Data Science*. Second, I aim to show how one can process JSON data provided by SEC EDGAR using submissions data as case study. Finally, I introduce what I call a **batch-processing paradigm**, which is a general approach that I believe is helpful for many significant data processing tasks.

Before proceeding, I make some brief comments on my experience working with the submissions data provided by the SEC. While the SEC continues to be a leader in providing publicly

[1] *Empirical Research in Accounting: Tools and Methods* will be published in print form by CRC Press later in 2024 and will remain free online after publication.

[2] Explaining my views on the (limited) role of spreadsheets in research probably requires a separate note, but a good starting point is Broman and Woo (2018).

available data on public companies and other entities in electronic form, I did not find the submissions data set particularly easy to work with.[3] While downloading the data submissions data in "bulk" format (`submissions.zip`) takes less than a minute, processing these data into a form that is easy to work with requires several hours.[4] At the end of this note I show how parquet files can be used to store the contents of the `submissions.zip` in a form that is easy to download and relatively painless to work with.

In writing this note, I use the packages listed below.[5] 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.

```
library(tidyverse)
library(DBI)
library(jsonlite)
library(farr)
```

JSON

JSON stands for "Javascript object notation" and is a common form of data on the internet. The SEC provides data in JSON form, including the submissions data I focus on in this note, and here I provide a brief overview of JSON.

JSON is a text-based data format with four scalar data types:

- **nulls** are similar to `NA` in R in that `null` represents the absence of data.
- **strings**, as in R, but always quoted using double quotes (`"`).
- **numbers**, as in R, but no support for `Inf`, `-Inf`, or `NaN`. Can be formatted using integer, decimal, or scientific (e.g., `2.99e8`) notation.
- **Booleans**, similar to R's logical type. Either `true` or `false`.

These scalar types can be used to form **arrays** and **objects**. An array is like an unnamed list, and is written with `[]`. For example `[2, 4, 6]` is an array containing 3 numbers, and `[null, 2, "text", true]` is an array that contains a null, a number, a string, and a Boolean. An object is like a named list, and is written with `{}`. The names ("keys" in JSON terminology) are provided as strings. For example, `{"a": 2, "b": 4}` is an object that maps `a` to 2 and `b` to 4.[6]

Note that dates or date-times are not among the scalar data types supported by JSON. Wickham, Çetinkaya-Rundel, and Grolemund (2023, 418) note that dates are "often stored as strings, and

[3] For example, Australian regulators do very little to ensure that public companies file data in a form that analysts and researchers can use and the situation in Australia is probably behind where SEC EDGAR was in 1995.
[4] Sorting out data gremlins meant that the whole task took a few days in elapsed time.
[5] Execute `install.packages(c("tidyverse", "DBI", "duckdb", "jsonlite", "farr"))` within R to install all the packages you need to run the code in this note.
[6] Python users may notice similarities between arrays and Python lists and between objects and Python dictionaries.

you'll need to use `readr::parse_date()` or `readr::parse_datetime()` to turn them into the correct data structure."

Often a JSON file contains a single top-level array that represents multiple items, e.g., multiple records or multiple results. In this case, `tibble(json)` will treat each item as a row.

A simplified version of the contents of a SEC JSON file is given below.

```
json_example <- '{
  "cik":2809,
  "entityType":"other",
  "sic":"1040",
  "insiderOwner":false,
  "description":null,
  "filings":{
    "accessionNumber":["0001104659-24-053850", "0001104659-24-053618"],
    "filingDate":["2024-04-29", "2024-04-29"]}}'
```

Here we see that the JSON stored in `json_example` comprises a single top-level JSON object. As such, we follow the advice of *R for Data Science* for such cases by wrapping the JSON in a list before putting it in a tibble.

```
json_example_df <- tibble(json = list(parse_json(json_example)))
json_example_df
```

```
# A tibble: 1 x 1
  json
  <list>
1 <named list [6]>
```

We see that `json` in `json_example_df` is a list-column that contains a single row. List-columns can be either *named* or *unnamed* and `json` is a named list-column. Wickham, Çetinkaya-Rundel, and Grolemund (2023, 406) suggest that "when the children are named, they tend to have the same names in every row. ... Named list-columns naturally unnest into columns: each named element becomes a new named column." We can apply this approach using `unnest_wider()`:

```
json_example_df |>
  unnest_wider(json)
```

```
# A tibble: 1 x 6
    cik entityType sic   insiderOwner description filings
  <int> <chr>      <chr> <lgl>        <lgl>       <list>
1  2809 other      1040  FALSE        NA          <named list [2]>
```

From the results of applying `unnest_wider()`, we see that `cik`, `entityType`, and `insiderOwner` have been correctly parsed as integer, text, and logical types, respectively. The `null` for `description` has been understood as `NA`. Finally, `filings` is a *named* list-column to which we can apply `unnest_wider()`.

```
json_example_df |>
  unnest_wider(json) |>
  select(-description) |>
  unnest_wider(filings)
```

```
# A tibble: 1 x 6
    cik entityType sic   insiderOwner accessionNumber filingDate
  <int> <chr>      <chr> <lgl>        <list>          <list>
1  2809 other      1040  FALSE        <list [2]>      <list [2]>
```

Now we see that `accessionNumber` and `filingDate` are *unnamed* list columns. Wickham, Çetinkaya-Rundel, and Grolemund (2023, 407) advises that "when each row contains an unnamed list, it's most natural to put each element into its own row with `unnest_longer()`" and we follow that advice here.

```
json_example_df |>
  unnest_wider(json) |>
  unnest_wider(filings) |>
  unnest_longer(c(accessionNumber, filingDate))
```

```
# A tibble: 2 x 7
    cik entityType sic   insiderOwner description accessionNumber
  <int> <chr>      <chr> <lgl>        <lgl>       <chr>
1  2809 other      1040  FALSE        NA          0001104659-24-053850
2  2809 other      1040  FALSE        NA          0001104659-24-053618
# i 1 more variable: filingDate <chr>
```

Readers familiar with database design principles might cringe at the output above, as we have duplicated a number of items that appear to have a one-to-one relationship with a company (here identified by `cik`)—`entityType`, `sic`, `insiderOwner`, and `description`—because of the multiplicity of the items stored in `filings`. To create the `companies` table, we simply drop the `filings` column.

```
companies <-
  json_example_df |>
  unnest_wider(json) |>
```

```
  select(-filings)

companies
```

```
# A tibble: 1 x 5
    cik entityType sic   insiderOwner description
  <int> <chr>      <chr> <lgl>        <lgl>
1  2809 other      1040  FALSE        NA
```

To create the filings table, we retain only `cik` and `filings`. The `cik` column allows us to combine the `companies` and `filings` as necessary. As discussed above, we need to apply `parse_date()` to the string stored in `filingDate` to convert it to R's `Date` type.

```
filings <-
  json_example_df |>
   unnest_wider(json) |>
  select(cik, filings) |>
  unnest_wider(filings) |>
  unnest_longer(c(accessionNumber, filingDate)) |>
  mutate(filingDate = parse_date(filingDate))

filings
```

```
# A tibble: 2 x 3
    cik accessionNumber      filingDate
  <int> <chr>                <date>
1  2809 0001104659-24-053850 2024-04-29
2  2809 0001104659-24-053618 2024-04-29
```

SEC submissions data

According to the SEC's webpage, "each entity's current filing history is available at the following URL: `https://data.sec.gov/submissions/CIK##########.json` where the `##########` is the entity's 10-digit Central Index Key (CIK), including leading zeros." This means we will use CIKs to identify files to collect. Most data sources—including the SEC's own index files—provide CIKs as integers, but we can easily construct the 10-digit strings with leading zeros using `str_pad()` and construct the desired URL from that. To start we focus (arbitrarily) on Agnico Eagle Mines, a Canadian firm with CIK 2809. Running the following code should open a browser tab with the JSON data for submissions by Agnico Eagle Mines.

```
cik <- 2809
json_file <- str_c("CIK", str_pad(cik, width = 10, pad = "0"), ".json")
url <- paste0("https://data.sec.gov/submissions/", json_file)
```

```
browseURL(url)
```

We embed the logic for creating a URL from the file name (`json_file`) and the first couple of steps of processing the JSON in a function. The reasons for using the file name rather than the CIK and for the (as-yet-unused) `local_file` argument will become clear later.

```
get_sec_json <- function(json_file, local_file = NULL) {
  url <- paste0("https://data.sec.gov/submissions/", json_file)
  tibble(json = list(read_json(url))) |>
    unnest_wider(json)
}
```

The PCAOB requires users to provide email addresses when downloading data using a script and we can set this using the `HTTPUserAgent` option as in the R code below.[7]

```
options(HTTPUserAgent = "your_name@some_email.com")
```

Once we have set `HTTPUserAgent`, we can execute the `get_sec_json()` function:

```
get_sec_json(json_file)
```

```
# A tibble: 1 x 22
  cik   entityType sic   sicDescription       insiderTransactionForOwnerE~1
  <chr> <chr>      <chr> <chr>                                         <int>
1 2809  other      1040  Gold and Silver Ores                              0
# i abbreviated name: 1: insiderTransactionForOwnerExists
# i 17 more variables: insiderTransactionForIssuerExists <int>,
#   name <chr>, tickers <list>, exchanges <list>, ein <chr>,
#   description <chr>, website <chr>, investorWebsite <chr>,
#   category <chr>, fiscalYearEnd <chr>, stateOfIncorporation <chr>,
#   stateOfIncorporationDescription <chr>, addresses <list>, phone <chr>,
#   flags <chr>, formerNames <lgl>, filings <list>
```

From the output above, we see that there are four list columns: `tickers`, `exchanges`, `addresses`, and `filings`. For the most part the company-level data has one row per company, while the filing-level data might have thousands of rows for a given company. For this reason, I split the data processing into company-level and filing-level processes.

[7] Use your *actual* email address here.

Company-level data

I embed code all to collect company-level data in a simple function `get_company_data()`. I use `get_sec_json()` to get the JSON, then drop `filings` before processing the remaing columns as needed. While `cik` was supplied as a number in the example above, the actual data on the SEC site present this as a string and I convert it to integer using `as.integer()` (and do likewise for `sic`). Similarly, the two fields that are fundamentally Boolean are supplied as either `0` or `1` rather than as `true` or `false` and I convert these to logical type using `as.logical()`.

```
get_company_data <- function(cik, local_file = NULL) {
  json_file <- str_c("CIK", str_pad(cik, width = 10, pad = "0"), ".json")

  get_sec_json(json_file, local_file) |>
    select(-filings) |>
    mutate(across(c(cik, sic), as.integer),
           across(ends_with("Exists"), as.logical))
}
```

We can apply our function to `cik`, which we set to 2809 above and store the result in `company_data`.

```
company_data <- get_company_data(cik)
```

We then split the `company_data` data frame into three data frames:

The first data frame (`companies`) contains all the columns that are not list-columns, which is most of the company-level data.

```
companies <-
  company_data |>
  select(-tickers, -exchanges, -addresses)

companies
```

```
# A tibble: 1 x 18
    cik entityType   sic sicDescription       insiderTransactionForOwnerE~1
  <int> <chr>      <int> <chr>               <lgl>
1  2809 other       1040 Gold and Silver Ores FALSE
# i abbreviated name: 1: insiderTransactionForOwnerExists
# i 13 more variables: insiderTransactionForIssuerExists <lgl>,
#   name <chr>, ein <chr>, description <chr>, website <chr>,
#   investorWebsite <chr>, category <chr>, fiscalYearEnd <chr>,
#   stateOfIncorporation <chr>, stateOfIncorporationDescription <chr>,
#   phone <chr>, flags <chr>, formerNames <lgl>
```

We can confirm that these type conversions discussed above behave as expected using the following code:

```
companies |> select(cik, sic, ends_with("Exists"))
```

```
# A tibble: 1 x 4
    cik   sic insiderTransactionForOwnerExists insiderTransactionForIssue~1
  <int> <int> <lgl>                           <lgl>
1  2809  1040 FALSE                           FALSE
# i abbreviated name: 1: insiderTransactionForIssuerExists
```

The second data frame is `addresses`. Rather than the more typical named list of unnamed lists, the `addresses` list-column in `company_data` contains an unnamed list of named lists, which requires a reversal of the usual `unnest_wider()`-then-`unnest_longer()` paradigm we saw above. It appears that most filers have a mailing address and a business address.

```
addresses <-
  company_data |>
  select(cik, addresses) |>
  unnest_longer(addresses, indices_to = "address_type") |>
  unnest_wider(addresses) |>
  select(cik, address_type, everything())

addresses
```

```
# A tibble: 2 x 8
    cik address_type street1           street2 city  stateOrCountry zipCode
  <int> <chr>        <chr>             <chr>   <chr> <chr>          <chr>
1  2809 mailing      145 KING STREET ~ SUITE ~ TORO~ A6             M5C 2Y7
2  2809 business     145 KING STREET ~ SUITE ~ TORO~ A6             M5C 2Y7
# i 1 more variable: stateOrCountryDescription <chr>
```

The third data frame is `tickers`, which contains data from the `tickers` and `exchanges` list-columns. While not represented explicitly in the JSON structure, it appears that the unnamed lists in these list-columns have the same length as they represent pairs (i.e., each ticker relates to a specific exchange).

```
tickers <-
  company_data |>
  select(cik, tickers, exchanges) |>
  unnest_longer(c(tickers, exchanges)) |>
```

```
  rename(ticker = tickers,
         exchange = exchanges)

tickers
```

```
# A tibble: 1 x 3
    cik ticker exchange
  <int> <chr>  <chr>
1  2809 AEM    NYSE
```

I initially tried keeping the `addresses` and `tickers` data as list-columns. However, for reasons discussed below, I end up putting the data in a DuckDB database and encountered problems with these columns when doing so. The unnamed list of named lists structure of the original `addresses` data was rejected by DuckDB. The issue with the `tickers-exchanges` data is that occasionally the exchange associated with a ticker is `NULL` and DuckDB rejected such data. Splitting out the `addresses` and `tickers` into separate tables resolves these data issues without any loss of data.

Filing-level data

We can now shift our focus to the filing-level data.

```
get_sec_json(json_file) |>
  select(filings)
```

```
# A tibble: 1 x 1
  filings
  <list>
1 <named list [2]>
```

Noting that it's a named list, we look at the results of applying `unnest_wider()` to `filings`:

```
get_sec_json(json_file) |>
  select(filings) |>
  unnest_wider(filings)
```

```
# A tibble: 1 x 2
  recent            files
  <list>            <lgl>
1 <named list [14]> NA
```

Here we see that there are two fields embedded in `filings`: `recent` and `files`. According to the SEC's webpage, the "JSON data structure [of the submissions data] contains metadata such as current name, former name, and stock exchanges and ticker symbols of publicly-traded companies. The object's property path contains at least one year's of filing or to 1,000 (whichever is more) of the most recent filings in a compact columnar data array. If the entity has additional filings, `files` will contain an array of additional JSON files and the date range for the filings each one contains."

In the current case, `files` contains only `NA`, suggesting that there are less than 1,000 filings in `recent` (so `recent` represents *all* filings for this issuer). We put `files` aside for now and focus on `recent`. As `recent` is a named list-column, the natural next step is to apply `unnest_wider()` to this column.

```
get_sec_json(json_file) |>
  select(filings) |>
  unnest_wider(filings) |>
  select(-files) |>
  unnest_wider(recent)
```

```
# A tibble: 1 x 14
  accessionNumber filingDate   reportDate acceptanceDateTime act    form
  <list>          <list>       <list>     <list>             <list> <list>
1 <list [716]>    <list [716]> <list>     <list [716]>       <list> <list>
# i 8 more variables: fileNumber <list>, filmNumber <list>, items <list>,
#   size <list>, isXBRL <list>, isInlineXBRL <list>,
#   primaryDocument <list>, primaryDocDescription <list>
```

Seeing that all remaining columns are unnamed list-columns, we now apply `unnest_longer()`.

```
get_sec_json(json_file) |>
  select(filings) |>
  unnest_wider(filings) |>
  select(-files) |>
  unnest_wider(recent) |>
  unnest_longer(everything())
```

```
# A tibble: 716 x 14
  accessionNumber      filingDate reportDate acceptanceDateTime act   form
  <chr>                <chr>      <chr>      <chr>              <chr> <chr>
1 0001104659-24-075579 2024-06-27 "2024-06-~ 2024-06-27T14:16:~ 34    6-K
2 0001104659-24-073878 2024-06-21 "2024-06-~ 2024-06-21T15:13:~ 34    6-K
3 0001104659-24-071293 2024-06-13 ""         2024-06-13T16:27:~ 33    F-3D
```

```
4 0001104659-24-070324 2024-06-11 ""          2024-06-11T14:38:~ 33    F-X
5 0001104659-24-070316 2024-06-11 ""          2024-06-11T14:22:~ 33    F-10~
# i 711 more rows
# i 8 more variables: fileNumber <chr>, filmNumber <chr>, items <chr>,
#   size <int>, isXBRL <int>, isInlineXBRL <int>, primaryDocument <chr>,
#   primaryDocDescription <chr>
```

This seems to work, as have a rectangular table of filing-level data. However, we now need to consider a case where `files` is not `NA`. Ashland Inc, a chemicals company with CIK of 7694, is such a case.

```
cik <- 7694
json_file <- str_c("CIK", str_pad(cik, width = 10, pad = "0"), ".json")
```

Retracing our steps above, we see that `recent` takes the same form as we saw for Agnico Eagle Mines, but that `files` is now an unnamed list-column. Applying `unnest_longer()`, we get a named list-column. Applying `unnest_wide()` to that named list-column, we get the following.

```
get_sec_json(json_file) |>
  select(filings) |>
  unnest_wider(filings) |>
  select(files) |>
  unnest_longer(files) |>
  unnest_wider(files)
```

```
# A tibble: 1 x 4
  name                                filingCount filingFrom filingTo
  <chr>                                     <int> <chr>      <chr>
1 CIK0000007694-submissions-001.json          159 1994-01-21 1997-08-07
```

Thus we see that the **base file** `CIK0000007694.json` contains filings data inside the `recent` element as well as a list of **supplemental files** in inside the `files` element.[8] We now examine the supplemental file identified above.

```
get_sec_json("CIK0000007694-submissions-001.json") |>
  unnest_longer(everything())
```

[8]There are not official terms, I made them up for the purpose of this note.

```
# A tibble: 159 x 14
  accessionNumber      filingDate reportDate acceptanceDateTime act   form
  <chr>                <chr>      <chr>      <chr>              <chr> <chr>
1 0000007694-97-000126 1997-07-10 "1997-06-~ 1997-07-10T00:00:~ ""    4
2 0000007694-97-000125 1997-07-10 "1997-06-~ 1997-07-10T00:00:~ ""    4
3 0000007694-97-000118 1997-07-09 "1997-07-~ 1997-07-09T00:00:~ ""    3
4 0000007694-97-000117 1997-07-09 ""         1997-07-09T00:00:~ ""    SC 1~
5 0000007694-97-000114 1997-07-02 "1997-07-~ 1997-07-02T00:00:~ ""    8-K
# i 154 more rows
# i 8 more variables: fileNumber <chr>, filmNumber <chr>, items <chr>,
#   size <int>, isXBRL <int>, isInlineXBRL <int>, primaryDocument <chr>,
#   primaryDocDescription <chr>
```

The output above suggests that the supplemental files (i.e., those identified in the `files` column) contain data in the same form as we get from applying `unnest_wider(recent)` to the base files. The base files contain the column `filings`, which the supplemental files do not. This suggests we could make one function that, depending on whether there is a `filing` column present, either extracts the contents of that field using (ultimately) `unnest_wider(recent)` or simply processes the data as is. The following `extract_filings()` function does just that.

```
extract_filings <- function(json_file, local_file = NULL) {

  raw_data <- get_sec_json(json_file, local_file)

  if ("filings" %in% colnames(raw_data)) {
    filings_raw <-
      raw_data |>
      select(filings) |>
      unnest_wider(filings) |>
      select(recent) |>
      unnest_wider(recent)
  } else {
    filings_raw <- raw_data
  }

  filings_raw |>
    unnest_longer(everything()) |>
    mutate(across(c(filingDate, reportDate), parse_date),
           acceptanceDateTime = parse_date_time(acceptanceDateTime,
                                                orders = "ymdHMS",
                                                tz = "America/New_York"),
           across(c(isXBRL, isInlineXBRL), as.logical))
}
```

In addition, `extract_filings()` converts `filingDate` and `reportDate` to date type, `isXBRL` and `isInlineXBRL` to logical type, `cik` to integer and `acceptanceDateTime` to date-time.

We can now see the rationale for using `json_file`, not `cik`, as the argument to `extract_filings()`: a single function can handle both base files (which can be identified using CIKs) and supplemental files, which require inspection of the base files to be identified.

Now we need to create a function that assembles the list of files (base files and supplemental files) for each CIK. The following `get_submission_files()` function does just that. It gets the JSON data for the base file and checks whether there are any non-null values in `files`. If there are no non-null values in `files`, then just the name of the base file is returned. Otherwise, the extracts the non-null values in `files` and adds them to the name of the base file.

```
get_submission_files <- function(cik, local_file = NULL) {

  json_file <- str_c("CIK", str_pad(cik, width = 10, pad = "0"), ".json")
  raw_data <- get_sec_json(json_file, local_file)

  files_df <-
    raw_data |>
    select(filings) |>
    unnest_wider(filings) |>
    select(files) |>
    unnest_longer(files) |>
    filter(!is.na(files))

  if (nrow(files_df) > 0) {
    files_df |>
      unnest_wider(files) |>
      select(name) |>
      pull() |>
      union_all(json_file)
  } else {
    json_file
  }
}
```

```
get_submission_files(2809)
```

```
[1] "CIK0000002809.json"
```

```
get_submission_files(7694)
```

```
[1] "CIK0000007694-submissions-001.json"
[2] "CIK0000007694.json"
```

The plan is for the output of `get_submission_files()` to be supplied to `extract_filings()`. The following `cik_to_filings()` function gives effect to this plan with the list of tibbles being combined into a single tibble using `list_rbind()`. Note that we use an anonymous function with `map()` so that we can pass along the value of the still mysterious `local_file` argument to `extract_filings()`.

```
cik_to_filings <- function(cik, local_file = NULL) {
  get_submission_files(cik, local_file = local_file) |>
    map(\(x) extract_filings(x, local_file = local_file)) |>
    list_rbind() |>
    mutate(cik = as.integer(cik)) |>
    select(cik, everything())
}
```

We can check that the `cik_to_filings()` function behaves as we expect it to do for the two companies we have focused on so far.

```
cik_to_filings(2809)
```

```
# A tibble: 716 x 15
    cik accessionNumber     filingDate reportDate acceptanceDateTime  act
  <int> <chr>               <date>     <date>     <dttm>              <chr>
1  2809 0001104659-24-0755~ 2024-06-27 2024-06-27 2024-06-27 14:16:35 34
2  2809 0001104659-24-0738~ 2024-06-21 2024-06-19 2024-06-21 15:13:26 34
3  2809 0001104659-24-0712~ 2024-06-13 NA         2024-06-13 16:27:55 33
4  2809 0001104659-24-0703~ 2024-06-11 NA         2024-06-11 14:38:35 33
5  2809 0001104659-24-0703~ 2024-06-11 NA         2024-06-11 14:22:39 33
# i 711 more rows
# i 9 more variables: form <chr>, fileNumber <chr>, filmNumber <chr>,
#   items <chr>, size <int>, isXBRL <lgl>, isInlineXBRL <lgl>,
#   primaryDocument <chr>, primaryDocDescription <chr>
```

```
cik_to_filings(7694)
```

```
# A tibble: 1,160 x 15
    cik accessionNumber     filingDate reportDate acceptanceDateTime  act
  <int> <chr>               <date>     <date>     <dttm>              <chr>
1  7694 0000007694-97-0001~ 1997-07-10 1997-06-30 1997-07-10 00:00:00 ""
```

```
2  7694 0000007694-97-0001~ 1997-07-10 1997-06-30 1997-07-10 00:00:00 ""
3  7694 0000007694-97-0001~ 1997-07-09 1997-07-01 1997-07-09 00:00:00 ""
4  7694 0000007694-97-0001~ 1997-07-09 NA         1997-07-09 00:00:00 ""
5  7694 0000007694-97-0001~ 1997-07-02 1997-07-01 1997-07-02 00:00:00 ""
# i 1,155 more rows
# i 9 more variables: form <chr>, fileNumber <chr>, filmNumber <chr>,
#   items <chr>, size <int>, isXBRL <lgl>, isInlineXBRL <lgl>,
#   primaryDocument <chr>, primaryDocDescription <chr>
```

Thus we have two core functions: `get_company_data()` to get the company-level data and `cik_to_filings()` to get the filing-level data. In principle, if we had a universe of CIKs that we wanted submission data for, we could put these in a vector and `map()` each of these two functions to that vector to get data.

```
ciks <- c(2809, 7694)
```

```
ciks |>
  map(cik_to_filings) |>
  list_rbind()
```

```
# A tibble: 1,876 x 15
    cik accessionNumber     filingDate reportDate acceptanceDateTime  act
  <int> <chr>               <date>     <date>     <dttm>              <chr>
1  2809 0001104659-24-0755~ 2024-06-27 2024-06-27 2024-06-27 14:16:35 34
2  2809 0001104659-24-0738~ 2024-06-21 2024-06-19 2024-06-21 15:13:26 34
3  2809 0001104659-24-0712~ 2024-06-13 NA         2024-06-13 16:27:55 33
4  2809 0001104659-24-0703~ 2024-06-11 NA         2024-06-11 14:38:35 33
5  2809 0001104659-24-0703~ 2024-06-11 NA         2024-06-11 14:22:39 33
# i 1,871 more rows
# i 9 more variables: form <chr>, fileNumber <chr>, filmNumber <chr>,
#   items <chr>, size <int>, isXBRL <lgl>, isInlineXBRL <lgl>,
#   primaryDocument <chr>, primaryDocDescription <chr>
```

However, I do not advise this approach because it involves a lot of internet traffic and is likely to take several hours. If there's a hiccup in the middle, we would finish up empty-handed and need to try all over again.

Regarding internet traffic, note that the code is fairly inefficient. The base file URL for Ashland Inc is accessed three times: once for `get_company_data()` and twice for `cik_to_filings()`. If internet traffic ends up being a bottleneck, then it might make sense to modify the code above to download data just once for each CIK. But I shelve this issue for now.[9]

[9] In fact, it turns out that this inefficiency has no significant impact on the running time for the code and, given that we use a bulk download file, I actually ignore this issue in this note.

Bulk data

According to the SEC website, "the most efficient means to fetch large amounts of API data is the bulk archive ZIP files, which are recompiled nightly. The `submission.zip` file contains the public EDGAR filing history for all filers from the Submissions API `https://www.sec.gov/Archives/edgar/daily-index/bulkdata/submissions.zip`."

Let's download this buk file:

```
bulk_url <- str_c("https://www.sec.gov/Archives/edgar/",
                  "daily-index/bulkdata/submissions.zip")
data_dir <- "data"
bulk_path <- file.path(data_dir, "submissions.zip")

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

if (!file.exists(bulk_path)) {
  result <-
    download.file(bulk_url, bulk_path) |>
    system_time()
}
```

```
   user  system elapsed
  4.606   1.194  17.643
```

From the following output, we can see that `submissions.zip` contains nearly 900,000 files. While we could simply "unzip" this file, this would take some time and we would have nearly 900,000 files in a single directory and working with files in that directory can become quite painful as a result.

```
bulk_files <-
  unzip(bulk_path, list = TRUE) |>
  as_tibble()

bulk_files
```

```
# A tibble: 888,257 x 3
  Name                Length Date
  <chr>                <dbl> <dttm>
1 CIK0000002809.json 108540 2024-06-28 05:03:00
2 CIK0000000013.json   1027 2021-04-30 04:21:00
3 CIK0000002969.json 137519 2024-06-21 05:06:00
```

```
4 CIK0000000003.json   1933 2021-04-30 04:21:00
5 CIK0000003499.json  79548 2024-06-13 05:02:00
# i 888,252 more rows
```

It turns out that we can use the `unz()` function to unzip a single file and, as we can identify the files we need from the CIKs themselves and inspection of the base files for those CIKs, we can elect to extract just the files we need as and when we need them. It is at this point that we can reveal the purpose of the `local_file` argument that we included above, but neither explained nor used. In short, we want to create a new version of `get_sec_json()` that gets the relevant data from a local file if `local_file` is supplied, but goes to the relevant URL if not. Note that the version of `get_sec_json()` below uses `parse_json()` instead of `read_json()` when working with a local file because the argument supplied represents a "connection" rather than an actual file.

```
get_sec_json <- function(json_file, local_file = NULL) {

  local <- !is.null(local_file)

  if (local) {
    t <- unz(local_file, json_file, open = "rb")
    raw_data <-
      tibble(json = list(parse_json(t, null = 'list'))) |>
      unnest_wider(json)
    close(t)
    unlink(t)
  } else {
    url <- paste0("https://data.sec.gov/submissions/", json_file)
    raw_data <-
      tibble(json = list(read_json(url))) |>
      unnest_wider(json)
  }
  raw_data
}
```

As we can see from the output below, our functions work exactly as they did before, but using the already-downloaded file.

```
cik_to_filings(7694, local_file = bulk_path)
```

```
# A tibble: 1,160 x 15
    cik accessionNumber     filingDate reportDate acceptanceDateTime  act
  <int> <chr>               <date>     <date>     <dttm>               <chr>
```

```
1  7694 0000007694-97-0001~ 1997-07-10 1997-06-30 1997-07-10 00:00:00 ""
2  7694 0000007694-97-0001~ 1997-07-10 1997-06-30 1997-07-10 00:00:00 ""
3  7694 0000007694-97-0001~ 1997-07-09 1997-07-01 1997-07-09 00:00:00 ""
4  7694 0000007694-97-0001~ 1997-07-09 NA         1997-07-09 00:00:00 ""
5  7694 0000007694-97-0001~ 1997-07-02 1997-07-01 1997-07-02 00:00:00 ""
# i 1,155 more rows
# i 9 more variables: form <chr>, fileNumber <chr>, filmNumber <chr>,
#   items <chr>, size <int>, isXBRL <lgl>, isInlineXBRL <lgl>,
#   primaryDocument <chr>, primaryDocDescription <chr>
```

Getting a list of CIKs

To put our code to the test, we will look at collecting submissions data for a larger set of CIKs. For most research applications, we would not be interested in all files in `submissions.zip`. Instead, we will use data from SEC index files to get the population of CIKs for filers that ever filed on a Form 8-K. Because we will use a persistent database to store the processed data, I store the table I create here in that database under the name `ciks`.

```
db_path <- file.path(data_dir, "submissions.duckdb")
db <- dbConnect(duckdb::duckdb(), dbdir = db_path, read_only = FALSE)

zip_ciks <-
  as_tibble(unzip(bulk_path, list = TRUE)) |>
  mutate(cik = as.integer(str_extract(Name, "[0-9]{10}"))) |>
  select(cik) |>
  distinct() |>
  copy_to(db, df = _, name = "zip_ciks", overwrite = TRUE,
          temporary = FALSE)

dbDisconnect(db)
```

```
db <- dbConnect(duckdb::duckdb(), dbdir = db_path, read_only = FALSE)

zip_ciks <- tbl(db, "zip_ciks")
sec_index <- load_parquet(db, "sec_index*", "edgar")
ciks <-
  sec_index |>
  filter(str_detect(form_type, "^8-K")) |>
  distinct(cik) |>
  semi_join(zip_ciks, by = "cik") |>
  compute(name = "ciks", overwrite = TRUE, temporary = FALSE)
```

This yields a total of 43,664 CIKs. To evaluate how long it will take to process submissions data for all of the CIKs, we can take a sample of 100 CIKs and time the processing for these and then extrapolate.

```
cik_sample <-
  ciks |>
  collect(n = 100) |>
  pull(cik)

dbDisconnect(db)
```

First, the company-level data. The first part of `process_companies()` runs `get_company_data()` for the CIKs in `ciks` and combines the result into a single data frame. Then `company_data` is split into three tables exactly as we did earlier. Finally, if `write_to_db` is `TRUE` the data are written to the database at the location specified by the argument `dbdir`.[10]

```
process_companies <- function(ciks, dest_table = "companies",
                              write_to_db = TRUE,
                              dbdir = db_path) {
  company_data <-
    ciks |>
    map(\(x) get_company_data(x, local_file = bulk_path)) |>
    list_rbind()

  companies <-
    company_data |>
    select(-any_of(c("tickers", "exchanges",
                     "addresses", "ownerOrg")))

  addresses <-
    company_data |>
    select(cik, addresses) |>
    unnest_longer(addresses, indices_to = "address_type") |>
    unnest_wider(addresses) |>
    select(cik, address_type, everything())

  tickers <-
    company_data |>
    select(cik, tickers, exchanges) |>
    unnest_longer(c(tickers, exchanges)) |>
    rename(ticker = tickers,
```

[10] The default value for `dbdir` is the value `db_path` set above.

```
            exchange = exchanges)

  if (write_to_db) {
    db <- dbConnect(duckdb::duckdb(), dbdir = dbdir, read_only = FALSE)
    dbWriteTable(db, dest_table, companies, append = TRUE)
    dbWriteTable(db, "addresses", addresses, append = TRUE)
    dbWriteTable(db, "tickers", tickers, append = TRUE)
    dbDisconnect(db)
  } else {
    list(companies = companies,
         addresses = addresses,
         tickers = tickers)
  }
}
```

We can test this function on the sample of CIKs in `cik_sample`:

```
cik_sample |>
  process_companies(write_to_db = FALSE) |>
  system.time()
```

```
   user  system elapsed
  9.566  10.452  20.267
```

Second, the filing-level data. The `process_filings()` function is relatively straightforward because just one table is produced by this code.

```
process_filings <- function(ciks, dest_table = "filings",
                            write_to_db = TRUE,
                            dbdir = db_path) {
  filings <-
    ciks |>
    map(\(x) cik_to_filings(x, local_file = bulk_path)) |>
    list_rbind()

  if (write_to_db) {
    db <- dbConnect(duckdb::duckdb(), dbdir = dbdir, read_only = FALSE)
    dbWriteTable(db, dest_table, filings,
                 append = TRUE)
    dbDisconnect(db)
  }
}
```

We can test `process_filings()` on the sample of CIKs in `cik_sample`:

```
cik_sample |>
  process_filings(write_to_db = FALSE) |>
  system.time()
```

```
   user  system elapsed
 32.621  33.654  70.274
```

So we will need around 50 seconds for 100 filings and thus roughly 6 hours for all filings. My experience is that estimates generated from small samples like this are likely to be underestimates because outliers will exist that require more time and such outliers may not be found in small samples.

The batch processing paradigm

In principle, we could just `collect()` all the CIKs in `ciks` and run the code above. However, this would be problematic for at least two reasons.

First, issues due to anomalies in the data are likely to arise that cause a breakdown midway through data collection. In practice it will be easier to address this issues if the process is broken into chunks (e.g., batches of 100 CIKs) so that problematic cases can be identified and remedied.

Second, even if the code has been adapted to address all data issues, a long-running process can still go awry for various random reasons (e.g., internet hiccups). A batched approach that allows the data collection effort to continue after interruption with minimal loss of data will minimize the costs of any interruptions.

Here I use the following **batch-processing paradigm** to process data in batches of CIKs.[11]

1. Create a population of units to the processed. Depending on the task a "unit" might be a conference call, or a 10-K filing, or (in this case) a CIK.
2. Get a sample of units that have not yet been processed.
3. Collect data for this sample.
4. Store the collected data.
5. Return to step (2) above until all units have been processed.

One key requirement to implement this paradigm is that we need to keep track of CIKs that have been processed already so that we only work on ones for which we need to collect data. For this purpose a database provides a natural structure. If the population identified in (1) is stored in a table `units_to_process` and the stored results in (4) go into `processed_data`, then

[11] I call this a *paradigm* because this is a common pattern or model that applies in many situations.

`units_to_process |> anti_join(processed_data)` provides the unprocessed units that can be sampled in step (2). Here I use the DuckDB database I created above to implement this paradigm. The table `ciks` we stored in that database represents the population.

The `get_ciks()` function takes two arguments. The first argument (`n_ciks`) specifies the size of the batch (i.e., number of CIKs) and here I set the default to 100 CIKs. The second argument (`dest_table`) identifies the table that is the primary destination for the data we are generating.

The basic idea of `get_ciks()` is quite simple: The `ciks` table (the universe of CIKs we are interested in) is compared with `dest_table` using an `anti_join()` so that only CIKs not found on the latter table are returned.[12] Finally, `get_ciks()` returns a sample of `n_ciks` CIKs.

```
get_ciks <- function(n_ciks = 100, dest_table = "companies",
                             dbdir = db_path) {
  db <- dbConnect(duckdb::duckdb(), dbdir = dbdir, read_only = TRUE)

  ciks <- tbl(db, "ciks")
  zip_ciks <- tbl(db, "zip_ciks")

  if (dest_table %in% dbListTables(db)) {
    to_update <- tbl(db, dest_table)
    unprocessed <-
      ciks |>
      semi_join(zip_ciks, by = "cik") |>
      anti_join(to_update, by = "cik")
  } else {
    unprocessed <- ciks
  }

  ciks <-
    unprocessed |>
    collect(n = n_ciks) |>
    select(cik) |>
    pull()

  dbDisconnect(db)

  ciks
}
```

We break the main data collection efforts into two steps: one for the company-level data and

[12] Initially there will be no `dest_table`, in which case the code just returns a sample of CIKs from `ciks`.

one for the filing-level data. Each of these steps is place in a `while()` loop that will run until no more data needs to be collected.

Company-level data

The code below starts by running `get_ciks(dest_table = "companies")` and stores the result in `ciks`. If there are CIKs in `ciks` (i.e., `length(ciks) > 0`), then the loop runs.

One difference between the issues for the two list-columns in the company-level data is that the `addresses` issue created problems immediately while `tickers-exchanges` issue did not cause problems until several thousand CIKs had been processed. An advantage of the while-loop structure is that when an error causes the code to stop, it is relatively easy to step through the code to identify the fields and observations that are causing problems.

Another issue that only appeared after several thousand CIKs had been processed related to the field `ownerOrg`, which isn't even meant to be part of the data. But, when dealing with data related to more than 40,000 entities, strange issues can crop up and the mistaken inclusion of data for `ownerOrg` is one of these. Note that we write `select(-any_of("ownerOrg"))` because `select(-ownerOrg)` will result in an error if `ownerOrg` is not on the `company_data` table, as it generally isn't.

Once we have created the three tables, we write them to the persistent DuckDB database we created above. We use `append = TRUE` to indicate that the data are to be added to whatever data are already there.

The code then returns to the beginning of the loop, where `get_ciks()` will examine the newly updated contents of `companies` so that only unprocessed CIKs are returned.

```
while (length(ciks <- get_ciks(dest_table = "companies",
                                           n_ciks = 100)) > 0) {
  process_companies(ciks, dest_table = "companies")
}
```

Filing-level data

The while-loop for `filings` is similar. The `cik_to_filings()` function takes the place of `get_company_data()` and there is no need to split the data into separate tables.

```
while (length(ciks <- get_ciks(dest_table = "filings")) > 0) {
  process_filings(ciks, "filings")
}
```

Extending the batch-processing paradigm

Whether we use a local copy of `submissions.zip` or access the JSON data directly on SEC EDGAR, the process here is inherently single-threaded. With `submissions.zip`, only one process can access the file at any time and the SEC imposes limits on accessing multile JSON files at once. As such the limitation that only a single process can access a DuckDB database with read-write access does not bind.

However, other tasks will not be subject to this limitation and parallel processing can provide dramatic performance improvements. For such parallel processing a server-based database system, such as PostgreSQL, is ideal. However, some modification is needed because we don't want one process to be processing data that another process is already working on.

1. Create a population of units to the processed. Assume this is in table `population`.
2. Get a sample of units that have not yet been processed
3. Store this sample table that identifies units selected for processing (e.g., `processed`)
4. Collect data for this sample.
5. Store the collected data in a table (e.g., `results`)
6. Return to step (2) above until all units have been processed.

Here step (2) will involve `population |> anti_join(processed)` and if steps (2) and (3) are put in a single database transaction, then it is possible to ensure that each unit is processed only once even if many processes are working simultaneously.

Exporting the data to parquet format

Finally, we export the four tables we have created as 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 *R for Data Science* and a guide to creating a parquet data repository are provided here.

For reasons discussed here, I use the environment variable `DATA_DIR` to identify the location in which I will store the data. If you do not have `DATA_DIR` set, you can easily set it using a command like this:[13]

```
Sys.setenv(DATA_DIR = "data")
```

I will store the data in a `submissions` directory under the `DATA_DIR` location.

[13] Change the directory to an appropriate location on your own computer.

```
pq_path = file.path(Sys.getenv("DATA_DIR"), "submissions")
```

Here I create a small function that uses the built-in functionality of DuckDB to save tables as parquet files.

```
db_to_parquet <- function(db, table, data_path) {
  if (!dir.exists(data_path)) dir.create(data_path)
  file_path <- file.path(data_path, str_c(table, ".parquet"))

  dbExecute(db, str_c("COPY ", table, " TO '", file_path,
                      "' (FORMAT PARQUET)"))
}
```

With `db_to_parquet()` in hand, it is easy to connect to the database and export the four tables.

```
db <- dbConnect(duckdb::duckdb(), dbdir = db_path, read_only = TRUE)

db_to_parquet(db, table = "companies", data_path = pq_path)
```

```
[1] 43664
```

```
db_to_parquet(db, table = "tickers", data_path = pq_path)
```

```
[1] 8610
```

```
db_to_parquet(db, table = "addresses", data_path = pq_path)
```

```
[1] 87728
```

```
db_to_parquet(db, table = "filings", data_path = pq_path)
```

```
[1] 12120262
```

```
dbDisconnect(db)
```

These four parquet files can be downloaded from here. We provide a guide to working with parquet files here. If you have set `DATA_DIR` correctly, then the `load_parquet()` function from the `farr` package makes it easy to access the data, as illustrated below.

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

filings <- load_parquet(db, table = "filings", schema = "submissions")
filings
```

```
# Source:   SQL [?? x 15]
# Database: DuckDB v1.0.0 [root@Darwin 23.6.0:R 4.4.0/:memory:]
    cik accessionNumber     filingDate reportDate acceptanceDateTime  act
  <int> <chr>               <date>     <date>     <dttm>              <chr>
1 93706 0000897101-99-0010~ 1999-11-12 NA         1999-11-12 05:00:00 ""
2 93706 0000897101-99-0008~ 1999-08-13 1999-06-30 1999-08-13 04:00:00 ""
3 93706 0000897101-99-0005~ 1999-05-14 1999-03-31 1999-05-14 04:00:00 ""
4 93706 0000897101-99-0003~ 1999-03-31 1998-12-31 1999-03-31 05:00:00 ""
5 93706 0000897101-98-0011~ 1998-11-16 1998-09-30 1998-11-16 05:00:00 ""
# i more rows
# i 9 more variables: form <chr>, fileNumber <chr>, filmNumber <chr>,
#   items <chr>, size <int>, isXBRL <lgl>, isInlineXBRL <lgl>,
#   primaryDocument <chr>, primaryDocDescription <chr>
```

```r
dbDisconnect(db)
```

References

Broman, Karl W., and Kara H. Woo. 2018. "Data Organization in Spreadsheets." *The American Statistician* 72 (1): 2–10. https://doi.org/10.1080/00031305.2017.1375989.

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.