Responsive open-source software: Two examples from `dbplyr`

Ian D. Gow

2025-12-17

1 Introduction

In this note, I explore some recent changes in the open-source R package `dbplyr` to illustrate some of the beauty of how open-source software evolves in practice. In particular, I offer two case studies where features requested by users became reality in `dbplyr`, which may be my favourite R package.

> **Tip**
>
> In writing this note, I used the packages listed below.[1] At the time of writing, you also need to install the development version of `dbplyr`, which you can do using the `remotes::install_github()` command below. This note was written and compiled using Quarto with RStudio, an integrated development environment (IDE) for working with R. The source code for this note is available here and the latest version of this PDF is here.

```
library(farr)
library(dplyr, warn.conflicts = FALSE)
library(DBI)
```

```
remotes::install_github("tidyverse/dbplyr", ref = "main")
```

2 The `copy_inline()` function

In the early days of writing *Empirical Research in Accounting: Tools and Methods*, I encountered an issue. The vision from the outset was that all the analyses in the book could be executed by the reader

1. Run `install.packages(c("farr", "dplyr", "DBI", "duckdb"))` within R to install all the packages you need to run the code in this note.

(primarily students taking a course based on the book) with minimal overhead. One aspect of that was reliance on the WRDS PostgreSQL database as the primary source for data without the need to download large amounts of data.

However, if we wanted to run an event study for a set of events in a local data set using CRSP daily data, we'd either have to download the gigabytes of data in the CRSP table or somehow upload our events to the WRDS database. Unfortunately, WRDS does not allow users to upload data.

So on 31 March 2021, I filed an issue in the GitHub repository for `dbplyr`, the relevant R package, where I outlined my problem:

> I use a database server where normal users do not have `TEMPORARY` privileges. This means that connections to the database are read-only and functions such as `compute()` or `dbWriteTable()` are not available. I suspect that this is not uncommon in the real world.
>
> One downside of this is that if I have data locally that I want to join with data on the server, the only real option is to `collect()` or otherwise download data from the server. But in a typical use case, the data I have locally are small, while the data on the server are large. So this is very inefficient.

I even made a (fairly awful) function that provided a stop-gap fix for my problem and asked if it made sense to add a more robust function to `dbplyr`. The initial response (coming within one day) from Hadley Wickham, the creator of `dbplyr`, was that "this just feels too specialised for `dbplyr`, sorry."

But six weeks later, Kirill Müller, a leading `dbplyr` contributor, suggested a (much better) function for consideration. Still Hadley said he was "sceptical that there are many purely read-only databases in the wild; there's no security risk to enabling temporary tables." As such, there would be little demand for the proposed function. Nonetheless, after a couple of other users pushed back on this point and Kirill pointed out performance benefits of the proposed approach, the go-ahead was given for the proposal.[2]

The proposal evolved into the `copy_inline()` function, which was released as part of `dbplyr` 2.2.0 on 6 June 2022. For a function that seemed to get little love at first, `copy_inline()` seems to have proven quite useful, as there are eight references to it in the `NEWS.md` file for the `dbplyr` package.

A recent tweak makes this function even more useful, as one can simply supply `copy = "inline"` as an argument to a join function to implicitly invoke `copy_inline()`. The following example illustrates how I can merge the 423 events in the `michels_2017` data set from the `farr` package with returns over the period from three days before through to three days after the event.[3]

2. The "go-ahead" seems to be implied by Hadley's reopening of the issue on 24 June 2021.
3. See Chapter 17 of *Empirical Research in Accounting: Tools and Methods* for more on the `michels_2017` data. Of course, a more careful approach would probably use *trading days* before and after events; see Chapter 12 of *Empirical Research in Accounting: Tools and Methods* for discussion of how to do this.

```
Sys.setenv(PGHOST = "wrds-pgdata.wharton.upenn.edu",
           PGPORT = 9737L,
           # PGUSER = "your_WRDS_ID",
           PGDATABASE = "wrds")

db <- dbConnect(RPostgres::Postgres())

crsp <- tbl(db, I("crsp.dsf"))

event_rets <-
  crsp |>
  select(permno, date, ret) |>
  mutate(win_start = date - days(3L),
         win_end = date + days(3L)) |>
  inner_join(michels_2017 |> select(permno, eventdate),
             join_by(permno,
                     between(y$eventdate, x$win_start, x$win_end)),
             copy = "inline") |>
  select(-starts_with("win")) |>
  collect() |>
  system_time()
```

```
   user  system elapsed
  0.032   0.004   1.532
```

This code runs in about one second and is quite easy to follow. The following is a sample of the resulting data frame.

```
event_rets
```

```
# A tibble: 2,089 x 4
   permno date            ret eventdate
    <int> <date>        <dbl> <date>
 1  50906 1994-01-14  0.0377  1994-01-17
 2  50906 1994-01-17  0.00990 1994-01-17
 3  50906 1994-01-18 -0.00654 1994-01-17
 4  50906 1994-01-19  0.00658 1994-01-17
 5  50906 1994-01-20 -0.0392  1994-01-17
 6  47730 2005-08-26 -0.0249  2005-08-29
 7  47730 2005-08-29  0.0115  2005-08-29
 8  47730 2005-08-30 -0.00155 2005-08-29
```

```
 9  47730 2005-08-31  0.0248  2005-08-29
10  47730 2005-09-01 -0.00857 2005-08-29
# i 2,079 more rows
```

3 Changes to `mutate()`

Another recent change to `dbplyr` relates to the `mutate()` function. Back in 2017, I made a request to add window-function capabilities to `dplyr`. This request morphed into an issue on `dbplyr` and resulted in the addition of `window_order()` and `window_frame()` functions in that package. These functions were one of two possible implementations considered at that time and, given the way that other functions worked, seemed to be the better of the two.

Subsequent developments in the `mutate()` function suggested the possibility that the other of the two implementations considered might now actually be preferable and I filed an issue on GitHub asking if it made sense to consider this alternative implementation.

Another recent change to `dbplyr` implements this alternative approach whereby `.order` and `.frame` arguments to `mutate()` allow users to access window functions.

To illustrate these changes, I will use data and queries from the book *SQL for Data Analysis* by Cathy Tanimura.[4]

To make it easy to get the data from the GitHub repository for *SQL for Data Analysis*, I made a couple of small functions:

```
load_csv <- function(conn, url, ...) {

  DBI::dbExecute(db, "INSTALL httpfs")

  df_sql <- paste0("SELECT * FROM read_csv('", url, "')")
  dplyr::tbl(conn, dplyr::sql(df_sql)) |>
    dplyr::compute(...)
}
```

```
get_data <- function(conn, dir, file, ...) {
  url <-  stringr::str_c("https://github.com/cathytanimura/",
                         "sql_book/raw/refs/heads/master/",
                         dir, "/", file)

  load_csv(conn, url, ...)
}
```

4. I have written about this book here.

3.1 The retail sales data set

The first query I will use to illustrate the use of window functions with `mutate()` comes from Chapter 3 of *SQL for Data Analysis*, which uses data on retail sales by industry in the United States to explore ideas on time-series analysis.

I will use DuckDB as my database engine. Creating a DuckDB database requires just one line of code:

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

I then call `get_data()` to load the data into the database. I name the table (`"retail_sales"`) so that I can refer to it when using SQL.[5]

```
retail_sales <- get_data(db,
                         dir = "Chapter 3: Time Series Analysis",
                         file = "us_retail_sales.csv",
                         name = "retail_sales")
```

The SQL version of the query provided in *SQL for Data Analysis* is as follows:

```
SELECT sales_month,
  avg(sales) OVER w AS moving_avg,
  count(sales) OVER w AS records_count
FROM retail_sales
WHERE kind_of_business = 'Women''s clothing stores'
WINDOW w AS (ORDER BY sales_month
             ROWS BETWEEN 11 PRECEDING AND CURRENT ROW)
ORDER BY sales_month DESC;
```

Table 1: Displaying records 1 - 10

sales_month	moving_avg	records_count
2020-12-01	2210.500	12
2020-11-01	2301.917	12
2020-10-01	2395.583	12
2020-09-01	2458.583	12
2020-08-01	2507.417	12
2020-07-01	2585.667	12
2020-06-01	2659.667	12
2020-05-01	2763.417	12

5. It is not necessary to specify a table name if we are just using `dbplyr` to analyse the data.

sales_month	moving_avg	records_count
2020-04-01	2989.083	12
2020-03-01	3248.167	12

To do the same using `dbplyr`, I can just specify `.order` and `.frame` in the call to `mutate()`.

```
mvg_avg <-
  retail_sales |>
  filter(kind_of_business == "Women's clothing stores") |>
  mutate(moving_avg = mean(sales, na.rm = TRUE),
         records_count = n(),
         .order = sales_month,
         .frame = c(-11, 0)) |>
  select(sales_month, moving_avg, records_count) |>
  arrange(desc(sales_month))
```

As can be seen in Table 2, the resulting data set is the same.[6]

```
mvg_avg |>
  collect(n = 10)
```

Table 2: Moving average sales for women's clothing store (first 10 records)

sales_month	moving_avg	records_count
2020-12-01	2210.500	12
2020-11-01	2301.917	12
2020-10-01	2395.583	12
2020-09-01	2458.583	12
2020-08-01	2507.417	12
2020-07-01	2585.667	12
2020-06-01	2659.667	12
2020-05-01	2763.417	12
2020-04-01	2989.083	12
2020-03-01	3248.167	12

6. This makes sense as the `dplyr`/`dbplyr` code is translated into SQL behind the scenes.

3.2 The legislators data

Another data set I will use to illustrate the use of `mutate()` is the legislators data set used in Chapter 4 of *SQL for Data Analysis* to explore cohort analysis. The legislators data set comprises two tables, which I read into my DuckDB database using the following code.

```
legislators_terms <- get_data(db,
                          dir = "Chapter 4: Cohorts",
                          file = "legislators_terms.csv",
                          name = "legislators_terms")

legislators <- get_data(db,
                          dir = "Chapter 4: Cohorts",
                          file = "legislators.csv",
                          name = "legislators")
```

A third data set used in Chapter 4 of *SQL for Data Analysis* is the `year_ends` table, which I construct in R and copy to my DuckDB database using the following code.[7]

```
year_ends <-
  tibble(date = seq(as.Date('1770-12-31'),
                    as.Date('2030-12-31'),
                    by = "1 year")) |>
  copy_to(db, df = _, overwrite = TRUE, name = "year_ends")
```

Finally, I add a minor tweak to original query by adding an enumerated data type that ensures the tables are ordered meaningfully.[8]

```
CREATE TYPE band AS ENUM ('1 to 4', '5 to 10', '11 to 20', '21+')
```

The following is a modified version of the SQL query found on page 173 of Chapter 4 of *SQL for Data Analysis*.[9] As can be seen, because of how we defined the `band` data type, it is meaningful to sort by `tenure` (check what happens if you omit the casting of `tenure` to type `band` using `::band`).

```
WITH term_dates AS (
  SELECT DISTINCT a.id_bioguide, b.date
  FROM legislators_terms a
```

7. In Chapter 4 of *SQL for Data Analysis*, `year_ends` is created using SQL in the relevant dialect, which is PostgreSQL in the book.
8. This data type is similar to **factors** in R, a topic covered in Chapter 2 of *Empirical Research in Accounting: Tools and Methods*.
9. Apart from formatting changes, the main modification I made to the query was the use of **common-table expressions** (**CTEs**) in place of subqueries. I discuss the merits of CTEs (and of using `dbplyr` to write SQL) here.

```
  JOIN year_ends b
  ON b.date BETWEEN a.term_start AND a.term_end
    AND b.date <= '2020-01-01'),

cum_term_dates AS (
  SELECT id_bioguide, date,
    count(date) OVER w AS cume_years
  FROM term_dates
  WINDOW w AS (PARTITION BY id_bioguide
               ORDER BY date
               ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)),

cum_term_bands AS (
  SELECT date,
    CASE WHEN cume_years <= 4 THEN '1 to 4'
         WHEN cume_years <= 10 THEN '5 to 10'
         WHEN cume_years <= 20 THEN '11 to 20'
         ELSE '21+' END AS tenure,
    COUNT(DISTINCT id_bioguide) AS legislators
  FROM cum_term_dates
  GROUP BY 1,2)

SELECT date, tenure::band AS tenure,
  legislators * 100.0 / sum(legislators) OVER w AS pct_legislators
FROM cum_term_bands
WINDOW w AS (partition by date)
ORDER BY date DESC, tenure;
```

Table 3: Displaying records 1 - 10

date	tenure	pct_legislators
2019-12-31	1 to 4	29.98138
2019-12-31	5 to 10	32.02980
2019-12-31	11 to 20	20.11173
2019-12-31	21+	17.87710
2018-12-31	1 to 4	25.60297
2018-12-31	5 to 10	33.76623
2018-12-31	11 to 20	21.33581
2018-12-31	21+	19.29499
2017-12-31	1 to 4	24.53532
2017-12-31	5 to 10	34.75836

Translating the query to `dbplyr` is greatly facilitated by the use of CTEs, as each CTE can be constructed as a separate **remote** or **lazy data frame**. Here *remote* means that the data are in a database. In this case, "remote" does not mean "far away", but the data could be physically distant as in the case of the `dsf` data frame examined above. The term "lazy" refers to the fact that the underlying SQL query for the data frame is not executed until we ask it to be evaluated using functions like `collect()` (to bring the data into R) or `compute()` (to create a temporary table in the database).

```
term_dates <-
  legislators_terms |>
  inner_join(year_ends |> filter(date <= '2020-01-01'),
             join_by(between(y$date, x$term_start, x$term_end))) |>
  distinct(id_bioguide, date)
```

Here I use `.order`, `.frame`, and `.by` to get the same window used in the SQL above.

```
cum_term_dates <-
  term_dates |>
  mutate(cume_years = n(),
         .by = id_bioguide,
         .order = date,
         .frame = c(-Inf, 0)) |>
  select(id_bioguide, date, cume_years)
```

The following query aggregates the data into different ranges of tenure. Note that a glitch in `n_distinct()` in the `duckdb` package (presumably something that will be fixed soon enough) means that I need to directly call SQL `COUNT(DISTINCT id_bioguide)` as can be seen in the code below.

```
cum_term_bands <-
  cum_term_dates |>
  mutate(tenure = case_when(cume_years <= 4 ~ '1 to 4',
                            cume_years <= 10 ~ '5 to 10',
                            cume_years <= 20 ~ '11 to 20',
                            TRUE ~ '21+')) |>
  mutate(tenure = sql("tenure::band")) |>
  summarize(legislators = sql("COUNT(DISTINCT id_bioguide)"),
            .by = c(date, tenure))
```

The final query pulls everything together and uses a window function to count the number of legislators in the denominator of `pct_legislators`. As can be seen in Table 4, the resulting data set is the same as that produced by the SQL above.

```
cum_term_bands |>
  mutate(sum_legislators = sum(legislators),
         pct_legislators = legislators * 100.0 / sum_legislators,
         .by = date) |>
  select(date, tenure, pct_legislators) |>
  arrange(desc(date), tenure) |>
  collect(n = 10)
```

Table 4: Percentage of legislators by tenure (first 10 records)

date	tenure	pct_legislators
2019-12-31	1 to 4	29.981
2019-12-31	5 to 10	32.030
2019-12-31	11 to 20	20.112
2019-12-31	21+	17.877
2018-12-31	1 to 4	25.603
2018-12-31	5 to 10	33.766
2018-12-31	11 to 20	21.336
2018-12-31	21+	19.295
2017-12-31	1 to 4	24.535
2017-12-31	5 to 10	34.758

Again, the new functionality supports powerful analyses with clean, easy-to-read code.