# Using DuckDB with WRDS data

Ian D. Gow

7 October 2023

Demonstrate the power of DuckDB and dbplyr with WRDS data.

In this short note, I show how one can use DuckDB with WRDS data stored in the PostgreSQL database provided by WRDS. I then use some simple benchmarks to show how DuckDB offers a powerful, fast analytical engine for researchers in accounting and finance.

To make the analysis concrete, I focus on data used in the excellent recent book "Tidy Finance with R". Essentially, I combine data from CRSP's daily stock return file (`crsp.dsf`) with data on factor returns from Ken French's website.

## 1 Summary of findings

Using DuckDB speeds up the data collection from WRDS in this case from 15 minutes to about 1 minute. Using DuckDB to do the aggregate query reduces the time taken from over two minutes using `dbplyr` to well under one second. DuckDB is faster than `dplyr` even if the data are in RAM. Additionally DuckDB is faster than SQLite. In fact, DuckDB runs faster collecting data from WRDS than `dplyr` does with data on disk in an SQLite database. Performance isn't everything, but performance gains like these likely deliver real quality-of-life benefits to data analysts.

## 2 Databases and tidy data

A popular way to manage and store data is with SQL databases. Tidy Finance with R uses SQLite, which "implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine." In this note, I use DuckDB, which has been described as offering "SQLite for Analytics". DuckDB is like SQLite in not requiring a server process, but like server-based databases such as PostgreSQL in terms of support for advanced SQL features and data types.

While storing data in a DuckDB offers some benefits of SQLite (e.g., data compression), the real benefits of using DuckDB come from using the database engine for data analytics. For the most part, Tidy Finance with R uses SQLite for storage and uses dplyr and in-memory data frames for analysis. For example, in the chapter on beta estimation, the data are read into memory immediately using collect() before any analysis is conducted. However, the dbplyr package allows many analytical tasks to be performed in the database. In this note, I demonstrate how using DuckDB and dbplyr can lead to significant performance gains.

# 3 Getting data

There are two data sets that we need to collect. The first is the factor returns, which we collect from Ken French's website using the frenchdata package. The second is from CRSP's daily stock file, which we get from WRDS.

We start by loading three packages. Note that we load DBI rather than the underlying database driver package.[1] In addition to these three packages, you should have the duckdb and RSQLite packages installed. Use install.packages() in R to install any missing packages.

```
library(tidyverse)
library(DBI)
library(frenchdata)
library(arrow)
```

Next we set up a DuckDB database file in the data directory, creating this directory if need be. We set read_only = FALSE because we will want to write to this database connection.

```
if (!dir.exists("data")) dir.create("data")

tidy_finance <- dbConnect(
  duckdb::duckdb(),
  "data/tidy_finance.duckdb",
  read_only = FALSE)
```

## 3.1 Fama-French factor returns

We use the same start_date and end_date values used in "Tidy Finance with R" and the code below also is adapted from that book. However, we use the copy_to() function from dplyr to save the table to our database.

[1] This is how it's done in "R for Data Science". I have read comments by Hadley Wickham that this is the right way to do it, but I can't find those comments.

```
start_date <- ymd("1960-01-01")
end_date <- ymd("2021-12-31")

factors_ff_daily_raw <-
  download_french_data("Fama/French 3 Factors [Daily]")
```

```
New names:
* `` -> `...1`
```

```
factors_ff_daily <-
  factors_ff_daily_raw$subsets$data[[1]] |>
  transmute(
    date = ymd(date),
    rf = as.numeric(RF) / 100,
    mkt_excess = as.numeric(`Mkt-RF`) / 100,
    smb = as.numeric(SMB) / 100,
    hml = as.numeric(HML) / 100
  ) |>
  filter(date >= start_date & date <= end_date) |>
  copy_to(tidy_finance,
          df = _,
          name = "factors_ff_daily",
          temporary = FALSE,
          overwrite = TRUE)
```

## 3.2 Getting daily returns from WRDS

Next, we will connect to the WRDS PostgreSQL database. It turns out to be most efficient for us to connect through our DuckDB connection, which we can do using `postgres_scanner`. This requires a few lines of code to set things up. First, we need to `LOAD postgres_scanner` (here I assume you have previously run `INSTALL postgres_scanner` to install the package … if not, please do that using `dbExecute()` first).[2] Second, because WRDS limits the number of connections we can make and DuckDB will happily try to make multiple connections to increase download speeds, we need to `SET threads TO 3`.

```
dbExecute(tidy_finance, "LOAD postgres_scanner")
dbExecute(tidy_finance, "SET threads TO 3")
```

[2] You can do this from R using `dbExecute(tidy_finance, "INSTALL postgres_scanner")`.

Third, I specify the connection details as follows. I recommend using environment variables (e.g., set using `Sys.setenv()`), as this facilitates sharing code with others. You should not include this chunk of code in your code, rather run it before executing your other code. In addition to setting these environment variables, you may want to set `PGPASSWORD` too. (Hopefully it is obvious that your should use *your* WRDS ID and password, not mine.)

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

Now we can connect to the CRSP daily stock file. While this is notionally `crsp.dsf`, it turns out that the "table" stored at `crsp.dsf` is actually a view, not a table and `postgres_scanner` requires a actual table. To access the table, we need to use the schema `crsp_a_stock`.[3]

```r
dsf_db <- tbl(tidy_finance,
              "postgres_scan_pushdown('', 'crsp_a_stock', 'dsf')")
```

As we can see, we have access to data in `crsp.dsf`.

```r
dsf_db
```

```
# Source:   table<postgres_scan_pushdown('', 'crsp', 'dsf')> [?? x 20]
# Database: DuckDB 0.9.0 [root@Darwin 23.1.0:R 4.3.1/data/tidy_finance.duckdb]
   cusip    permno permco issuno hexcd hsiccd date       bidlo askhi   prc   vol
   <chr>     <int>  <int>  <dbl> <dbl>  <dbl> <date>     <dbl> <dbl> <dbl> <dbl>
 1 68391610  10000   7952  10396     3   3990 1986-01-07  2.38  2.75 -2.56  1000
 2 68391610  10000   7952  10396     3   3990 1986-01-08  2.38  2.62 -2.5  12800
 3 68391610  10000   7952  10396     3   3990 1986-01-09  2.38  2.62 -2.5   1400
 4 68391610  10000   7952  10396     3   3990 1986-01-10  2.38  2.62 -2.5   8500
 5 68391610  10000   7952  10396     3   3990 1986-01-13  2.5   2.75 -2.62  5450
 6 68391610  10000   7952  10396     3   3990 1986-01-14  2.62  2.88 -2.75  2075
 7 68391610  10000   7952  10396     3   3990 1986-01-15  2.75  3    -2.88 22490
 8 68391610  10000   7952  10396     3   3990 1986-01-16  2.88  3.12 -3    10900
 9 68391610  10000   7952  10396     3   3990 1986-01-17  2.88  3.12 -3     8470
10 68391610  10000   7952  10396     3   3990 1986-01-20  2.88  3.12 -3     1000
# i more rows
# i 9 more variables: ret <dbl>, bid <dbl>, ask <dbl>, shrout <dbl>,
#   cfacpr <dbl>, cfacshr <dbl>, openprc <dbl>, numtrd <dbl>, retx <dbl>
```

[3] You may have access to `crsp_q_stock` or `crsp_m_stock` depending on your institution's subscription to CRSP.

The following code is adapted from the Tidy Finance code here. But, whereas the original code is much more complicated and takes more than 15 minutes, this code takes just over a minute.[4]

```
rs <- dbExecute(tidy_finance, "DROP TABLE IF EXISTS crsp_daily")

system.time({
  crsp_daily <-
    dsf_db |>
    filter(between(date, start_date, end_date),
           !is.na(ret)) |>
    select(permno, date, ret) |>
    mutate(month = as.Date(floor_date(date, "month"))) |>
    left_join(factors_ff_daily |>
                select(date, rf), by = "date") |>
    mutate(
      ret_excess = ret - rf,
      ret_excess = pmax(ret_excess, -1, na.rm = TRUE)
    ) |>
    select(permno, date, month, ret_excess) |>
    compute(name = "crsp_daily", temporary = FALSE)
})
```

```
   user  system elapsed
 21.683   4.258  50.508
```

### 3.3 Saving data to SQLite

If you have been working through “Tidy Finance”, you may already have an SQLite database containing `crsp_daily`. If not, we can easily create one now and copy the table from our DuckDB database to SQLite.

```
tidy_finance_sqlite <- dbConnect(
  RSQLite::SQLite(),
  "data/tidy_finance.sqlite",
  extended_types = TRUE)
```

[4] Performance will vary according to the speed of your connection to WRDS. Note that this query does temporarily use a significant amount of RAM on my machine, it is not clear that DuckDB will use as much RAM if this is more constrained. If necessary, you can run (say) `dbExecute(tidy_finance, "SET memory_limit='1GB'")` to constrain DuckDB’s memory usage; doing so has little impact on performance for this query.

```
copy_to(tidy_finance_sqlite,
        crsp_daily,
        name = "crsp_daily",
        overwrite = TRUE,
        temporary = FALSE)

dbExecute(tidy_finance_sqlite, "VACUUM")
```

We can also save the data to a parquet file.

```
dbExecute(tidy_finance,
          "COPY crsp_daily TO 'data/crsp_daily.parquet'
          (FORMAT 'PARQUET')")
```

Having created our two databases, we disconnect from them. This mimics the most common "write-once, read-many" pattern for using databases.

```
dbDisconnect(tidy_finance_sqlite)
dbDisconnect(tidy_finance, shutdown = TRUE)
```

## 4 Benchmarking a simple aggregation query

The following is a simple comparison of three different ways of doing some basic data analysis with R. After running the code above, we have the table `crsp_daily` as described in Tidy Finance in two separate databases: a SQLite database and a DuckDB database.

The following examines the same query processed in three different ways.

1. Using `dplyr` on an in-memory dataframe
2. Using `dbplyr` with an SQLite database
3. Using `dbplyr` with a DuckDB database

### 4.1 dplyr

We first need to load the data into memory.

```
tidy_finance <- dbConnect(
  RSQLite::SQLite(),
  "data/tidy_finance.sqlite",
  extended_types = TRUE)
```

```
crsp_daily <- tbl(tidy_finance, "crsp_daily")
```

What takes most time is simply loading nearly 2GB of data into memory.

```
system.time(crsp_daily_local <- crsp_daily |> collect())
```

```
   user  system elapsed
117.925   2.329 121.749
```

Once the data are in memory, it is *relatively* quick to run a summary query.

```
system.time(crsp_daily_local |>
              group_by(month) |>
              summarize(ret = mean(ret_excess, na.rm = TRUE)) |>
              collect())
```

```
   user  system elapsed
  3.093   0.690   3.817
```

```
rm(crsp_daily_local)
```

### 4.2 dbplyr with SQLite

Things are faster with SQLite, though there's no obvious way to split the time between reading the data and performing the aggregation. Note that we have a `collect()` at the end. This will not take a noticeable amount of time, but seems to be a reasonable step if our plan is to analyse the aggregated data in R.

```
system.time(crsp_daily |>
              group_by(month) |>
              summarize(ret = mean(ret_excess, na.rm = TRUE)) |>
              collect())
```

```
   user  system elapsed
 20.535   3.360  24.119
```

```
dbDisconnect(tidy_finance)
```

## 4.3 dbplyr with DuckDB

Let's consider DuckDB. Note that we are only reading the data here, so we set `read_only = TRUE` in connecting to the database. Apart from the connection, there is no difference between the code here and the code above using SQLite.

```r
tidy_finance <- dbConnect(
  duckdb::duckdb(),
  "data/tidy_finance.duckdb",
  read_only = TRUE)
```

```r
crsp_daily <- tbl(tidy_finance, "crsp_daily")
```

```r
system.time(crsp_daily |>
              group_by(month) |>
              summarize(ret = mean(ret_excess, na.rm = TRUE)) |>
              collect())
```

```
   user  system elapsed 
  1.072   0.277   0.293 
```

Having done our benchmarks, we can take a quick peek at the data.

```r
crsp_daily |>
  group_by(month) |>
  summarize(ret = mean(ret_excess, na.rm = TRUE)) |>
  arrange(month) |>
  collect()
```

```
# A tibble: 744 x 2
   month            ret
   <date>         <dbl>
 1 1960-01-01 -0.00213 
 2 1960-02-01  0.000325
 3 1960-03-01 -0.00115 
 4 1960-04-01 -0.00106 
 5 1960-05-01  0.00114 
 6 1960-06-01  0.000935
 7 1960-07-01 -0.000955
 8 1960-08-01  0.00159 
 9 1960-09-01 -0.00289
10 1960-10-01 -0.00116
# i 734 more rows
```

Finally, we disconnect from the database. This will happen automatically if we close R, etc., and is less important if we have `read_only = TRUE` (so there is no lock on the file), but we keep things tidy here.

```
dbDisconnect(tidy_finance, shutdown = TRUE)
```

### 4.3.1 dbplyr with DuckDB and parquet files

Let's do one more benchmark using the parquet data.

```
db <- dbConnect(duckdb::duckdb())
crsp_daily <- tbl(db, "read_parquet('data/crsp_daily.parquet')")
```

```
system.time(crsp_daily |>
              group_by(month) |>
              summarize(ret = mean(ret_excess, na.rm = TRUE)) |>
              collect())
```

```
   user  system elapsed
  1.739   0.416   0.451
```

```
dbDisconnect(db, shutdown = TRUE)
```

### 4.3.2 dbplyr with dplyr and parquet files

Let's do one more benchmark using the parquet data with the `arrow` library.

```
crsp_daily <- open_dataset('data/crsp_daily.parquet')
```

```
system.time(crsp_daily |>
              group_by(month) |>
              summarize(ret = mean(ret_excess, na.rm = TRUE)) |>
              collect())
```

```
   user  system elapsed 
  1.616   0.217   0.322 
```

### 4.3.3 Python and parquet files

It seems that Python (with pandas) is constrained by the need to load the data into memory.

```
import pandas as pd
import time
start_time = time.monotonic()
df = pd.read_parquet('data/crsp_daily.parquet')
avg = df.groupby('month').agg(ret = ('ret_excess', 'mean'))
seconds = time.monotonic() - start_time
print("Time Taken: {:0.2f} seconds.".format(seconds))
print(avg)
```

```
Time Taken: 19.07 seconds.
                 ret
month               
1960-01-01 -0.002130
1960-02-01  0.000325
1960-03-01 -0.001154
1960-04-01 -0.001062
1960-05-01  0.001137
...              ...
2021-08-01  0.000985
2021-09-01 -0.001301
2021-10-01  0.001344
2021-11-01 -0.002161
2021-12-01  0.000075

[744 rows x 1 columns]
```

```
import pyarrow as pa
import pyarrow.parquet as pq
import time
start_time = time.monotonic()
df = pq.read_table('data/crsp_daily.parquet')
avg = df.group_by('month').aggregate([('ret_excess', 'mean')])
seconds = time.monotonic() - start_time
print("Time Taken: {:0.2f} seconds.".format(seconds))
```

```python
print(avg.to_pandas().sort_values(by=['month']))
```

```
Time Taken: 0.36 seconds.
          month  ret_excess_mean
0    1960-01-01        -0.002130
1    1960-02-01         0.000325
2    1960-03-01        -0.001154
3    1960-04-01        -0.001062
4    1960-05-01         0.001137
..          ...              ...
720  2021-08-01         0.000985
721  2021-09-01        -0.001301
722  2021-10-01         0.001344
723  2021-11-01        -0.002161
724  2021-12-01         0.000075

[744 rows x 2 columns]
```