Improving performance of SQLite data

Ian D. Gow

29 December 2023

The *Tidy Finance* books use SQLite for data storage. However, SQLite appears to have appalling performance for some common tasks. In this note, I discuss some options for dramatically improving on this performance (e.g., reducing time to read data off disk from over two minutes to under one second).

1 Introduction

Tidy Finance with R and *Tidy Finance with Python* provide excellent introductions to doing data analysis for academic finance. Chapters 2–4 of either book provide code to establish an SQLite database that is used a the data source for the rest of the book. Recently I have been dabbling with the Python version of the book and have found the analyses to be surprisingly sluggish on my computer. This note explores some options for improving this performance while still getting the results found in the book.[1]

Here I show how one can easily convert an SQLite database to a set of parquet files. A similar approach could be used to convert, say, a schema of a PostgreSQL database to parquet files.[2] I may explore such an approach as an alternative to the `wrds2pg` package, which runs SAS code on the WRDS server to generate data used to construct parquet files.[3]

A virtue of the approach used in *Tidy Finance* is that it is flexible. Readers of *Tidy Finance* can easily incorporate the approach used here as they work though the core parts of that book. The source code for this note can be found here.

In this note, I will load the following R packages. I will also use `duckdb` and `RSQLite` packages. use `install.packages()` to get any package that you don't already have.

[1] I ran the code in this note on an M1 Pro MacBook Pro. Running the same code on an i7-3770K running Linux Mint gave similar relative results. Fast code on an old computer handily beats slow code on a new computer.

[2] This would involve using the `postgres` extension to DuckDB.

[3] Issue to be addressed would be supporting the various arguments to the `wrds_update()` function in `wrds2pg`, such as `col_types` and `keep`, and addressing memory limitations with large tables.

```
library(DBI)
library(dplyr)
library(dbplyr)
library(farr)
```

We will use the following small function to calculate the time taken for steps below.[4]

```
system_time <- function(x) {
  print(system.time(x))
  x
}
```

I have created an SQLite database comprising data generated from running code in chapter 2 and chapter 3.[5]

```
db_path <- "data/tidy_finance_python.sqlite"
```

2 Using RSQLite

To start with, I follow the approach used in *Tidy Finance*, which generally involves connecting to the SQLite database before using `collect()` to read data into memory in R. The following code is pretty much copy-pasted from Chapter 5.

```
tidy_finance <- dbConnect(
  RSQLite::SQLite(),
  db_path,
  extended_types = TRUE
)
```

```
crsp_daily <- tbl(tidy_finance, "crsp_daily") |>
  select(permno, month, date, ret_excess) |>
  collect() |>
  system_time()
```

[4] Unlike the base R `system.time()`, this function works with assignment. If we put `system.time()` at the end of a pipe, then the value returned by `system.time()` would be stored rather than the result of the pipeline preceding it. Hadley Wickham explained to me that this function works because of **lazy evaluation**, which is discussed in "Advanced R" here. Essentially, `x` is evaluated just once—inside `system.time()`—and its value is returned in the next line.

[5] Chapter 4 covers data that are not needed in the later chapters I am currently looking at.

```
   user  system elapsed
146.369   2.471 151.061
```

That's more than two minutes just to read one table. And things are no better using Python.

```
import sqlite3
import time
import pandas as pd

tidy_finance = sqlite3.connect(
  database="data/tidy_finance_python.sqlite"
)

start = time.time()
crsp_daily = (pd.read_sql_query(
    sql=("SELECT permno, month, date, ret_excess "
         "FROM crsp_daily"),
    con=tidy_finance,
    parse_dates={"month", "date"})
  .dropna()
)
end = time.time()

print(f"Loading crsp_daily: {end - start:.3f} seconds")
```

```
Loading crsp_daily: 178.413 seconds
```

```
dbDisconnect(tidy_finance)
```

These numbers indicate mind-bogglingly bad performance. Mind-boggling because it is not clear how it can take so long to read 4GB of structured data into memory.

3 Using DuckDB to interact with SQLite

Rather than using `RSQLite` to interact with the *Tidy Finance* SQLite database, we could use DuckDB. DuckDB is touted as "SQLite for analytics", as it does not require a separate server process and is easy to install, but has a much more feature-rich SQL dialect and a larger set of data types and offers superior performance. To use DuckDB, we first instantiate an in-memory database.

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

Next, we load the `sqlite` extension and attach the *Tidy Finance* SQLite database. Note that the `INSTALL` takes less than a second and happens only once per installation of DuckDB, but the `LOAD` is needed for each new connection.[6] Windows users may need to o more work to install the `sqlite` extension (see discussion here).[7]

```
dbExecute(db, "INSTALL sqlite")
dbExecute(db, "LOAD sqlite")
dbExecute(db, paste0("ATTACH '", db_path, "' AS tf"))
dbExecute(db, "USE tf")
```

Now, we could `collect()` the data from `crsp_daily` above using DuckDB. As can be seen, it is much faster than using SQLite to do this.

```
crsp_daily <- tbl(db, "crsp_daily") |>
  select(permno, month, date, ret_excess) |>
  collect() |>
  system_time()
```

```
   user  system elapsed
 11.903   1.172  13.135
```

While much faster, more than ten seconds is not great. In fact, we may find it better to leave the data in the database and do computations there. For example, we can do a version of the monthly beta calculations done in *Tidy Finance* inside the DuckDB database. This calculation requires two tables: `crsp_monthly` and `factors_ff3_monthly`.

```
crsp_monthly <- tbl(db, "crsp_monthly")
factors_ff3_monthly <- tbl(db, "factors_ff3_monthly")
```

We will also use **window functions**

```
w <- paste("OVER (PARTITION BY permno",
           "ORDER BY date",
           "RANGE BETWEEN INTERVAL '60 MONTHS' PRECEDING AND",
           "CURRENT ROW)")
```

[6] Subsequent calls to `INSTALL` appear to have no effect.
[7] I don't have ready access to a Windows computer to check.

All the calculations below occur in DuckDB with the data not reaching R until the `collect()` at the end.[8]

```
beta_monthly <-
  crsp_monthly |>
  inner_join(factors_ff3_monthly, by = "month") |>
  mutate(month = as.Date(month)) |>
  mutate(beta = sql(paste("regr_slope(ret_excess, mkt_excess)", w)),
         n_rets = sql(paste("regr_count(ret_excess, mkt_excess)", w))) |>
  filter(n_rets >= 48) |>
  select(permno, month, beta) |>
  ungroup() |>
  collect() |>
  system_time()
```

```
   user  system elapsed
  2.824   0.508   0.447
```

We have loaded data, calculated 2,262,482 betas (each involving a regression) and brought the data into memory in R in about half a second!

```
beta_monthly
```

```
# A tibble: 2,262,482 x 3
   permno month       beta
    <dbl> <date>     <dbl>
 1  10122 1990-03-01 0.944
 2  10122 1990-04-01 0.931
 3  10122 1990-05-01 0.828
 4  10122 1990-06-01 0.846
 5  10122 1990-07-01 0.875
 6  10122 1990-08-01 0.809
 7  10122 1990-09-01 0.878
 8  10122 1990-10-01 0.870
 9  10122 1990-11-01 0.922
10  10122 1990-12-01 0.927
# i 2,262,472 more rows
```

[8] Usually one would use `group_by()`, `window_order()`, and `window_range()` to generate the requisite `PARTITION BY`, `ORDER BY`, and `ROWS BETWEEN` SQL clauses. However, a gap in the `dbplyr` package means that it doesn't recognize `regr_slope()` and `regr_count()` functions as aggregates that require them to be put in a "window context". I will likely file an issue related to this on the `dbplyr` GitHub page soon.

4 Converting the *Tidy Finance* database to parquet files

Rather than keeping the data in an SQLite database, we could actually convert the entire *Tidy Finance* database to parquet files using the following function, which leverages DuckDB's ability to create parquet files.[9]

```
to_parquet <- function(con, table, schema = "",
                       data_dir = "data") {
  df <- tbl(con, table) |>
    mutate(across(any_of(c("month", "date")), as.Date)) |>
    compute(name = "df", overwrite = TRUE)
  pq_dir <- file.path(data_dir, schema = schema)
  if (!dir.exists(pq_dir)) dir.create(pq_dir)
  pq_file_name <- paste0(table, ".parquet")
  pq_path <- file.path(pq_dir, pq_file_name)
  res <- dbExecute(con, paste0("COPY (SELECT * FROM df) TO '",
                               pq_path, "' (FORMAT PARQUET)"))
  dbExecute(con, "DROP TABLE df")
  return(tibble(table = table, rows = res))
}
```

Converting all the tables takes around 20 seconds. While there are now 11 files rather than one, these files use less than a quarter of the disk space of the original database file.

```
db |>
  dbListTables() |>
  lapply(X = _, \(x) to_parquet(db, x)) |>
  bind_rows() |>
  arrange(desc(rows)) |>
  system_time()
```

```
   user  system elapsed
 26.475   3.167  19.965
```

```
# A tibble: 11 x 2
   table                     rows
   <chr>                    <dbl>
 1 crsp_daily            71111235
 2 beta                   3338738
```

[9] Make sure you don't use this function with a database that already has a table named `df` in it, especially if you care about that table.

```
 3 crsp_monthly           3326353
 4 compustat               549537
 5 factors_ff3_daily        15858
 6 cpi_monthly                756
 7 factors_ff3_monthly        756
 8 industries_ff_monthly      756
 9 macro_predictors           755
10 factors_ff5_monthly        714
11 factors_q_monthly          672
```

Now that we have the everything in parquet format, we can read the data even faster than using DuckDB to read SQLite tables. These files can be used equally easily with Python. And R and Python have libraries (e.g., `arrow` in R) for working with parquet files directly.

Returning to our original benchmark, how long does it take to load the `crsp_daily` data into memory in R?

```
crsp_daily <-
  load_parquet(db, "crsp_daily", data_dir ="data") |>
  select(permno, month, date, ret_excess) |>
  collect() |>
  system_time()
```

```
   user  system elapsed
  1.985   0.547   0.726
```

Less than a second? Here's a peek at the data.

```
crsp_daily
```

```
# A tibble: 71,111,235 x 4
   permno month      date       ret_excess
    <dbl> <date>     <date>          <dbl>
 1  10000 1986-01-01 1986-01-08   -0.0246
 2  10000 1986-01-01 1986-01-09   -0.00025
 3  10000 1986-01-01 1986-01-10   -0.00025
 4  10000 1986-01-01 1986-01-13    0.0498
 5  10000 1986-01-01 1986-01-14    0.0474
 6  10000 1986-01-01 1986-01-15    0.0452
 7  10000 1986-01-01 1986-01-16    0.0432
 8  10000 1986-01-01 1986-01-17   -0.00025
```

```
 9  10000 1986-01-01 1986-01-20   -0.00025
10  10000 1986-01-01 1986-01-21   -0.00025
# i 71,111,225 more rows
```

4.1 Calculating monthly betas—again

Let's do the beta calculation again. Everything is as it was above except the first two lines, which point `crsp_monthly` and `factors_ff3_monthly` to parquet files in place of SQLite tables. Perhaps unsurprisingly, performance is similar—it would be difficult to beat what we saw earlier by much.

```
crsp_monthly <- load_parquet(db, "crsp_monthly", data_dir = "data")
factors_ff3_monthly <- load_parquet(db, "factors_ff3_monthly",
                                    data_dir = "data")
```

```
beta_monthly <-
  crsp_monthly |>
  inner_join(factors_ff3_monthly, by = "month") |>
  mutate(beta = sql(paste("regr_slope(ret_excess, mkt_excess)", w)),
         n_rets = sql(paste("regr_count(ret_excess, mkt_excess)", w))) |>
  filter(n_rets >= 48) |>
  select(permno, month, beta) |>
  collect() |>
  system_time()
```

```
   user  system elapsed
  2.059   0.131   0.295
```

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