Converting lazy data frames into Parquet files (Python version)

Ian D. Gow

23 March 2026

Abstract

In a note I published yesterday (*Converting lazy data frames into Parquet files*), I showed how my `db2pq` R package can be used to turn "lazy data frames" produced by the R package `dbplyr` into Parquet files on disk without needing to load the data into memory. Today, I wondered if I could do the same with my `db2pq` *Python* package. In this note, I turn "lazy data frames" produced by Ibis into Parquet files on disk.

Note

The source for this note is available in a folder on GitHub that contains two files:

- `ibis_to_pq.qmd`
- `pyproject.toml`

To run this note locally with `uv`, put the two files above in a directory on your computer, I will refer to this directory as the **project directory** in this now.
For the convenience of *Tidy Finance* readers, I follow the approach laid out here closely. The *Tidy Finance* instructions start with installation of "`uv`, a modern Python package and project manager." You should install `uv` using the instructions provided by *Tidy Finance*; these are taken from the homepage for `uv`.
From the project directory, run the following command:

```
uv sync
```

Next, you should follow the instructions provided by *Tidy Finance* for setting things up to connect to WRDS:

> Create a `.env` file in your project directory. For the purpose of this book, we create and save the following variables (where user and password are our private login credentials):

```
WRDS_USER=user
WRDS_PASSWORD=password
```

Once you have done the above, you should be able to run the code in this note. For example, you could `uv run jupyter lab` and copy-paste the commands show here (in order). Alternatively, if you are comfortable with Quarto, you could open `ibis_to_pq.qmd` and treat it like a notebook.

1 Motivating example

According to Tidy Finance with Python:

> The daily CRSP data file is substantially larger than monthly data and can exceed 20 GB. This has two important implications: you cannot hold all the daily return data in your memory (hence it is not possible to save the entire dataset to your local folder), and in our experience, the download usually crashes (or never stops) because it is too much data for the WRDS cloud to prepare and send to your R session.
>
> There is a solution to this challenge. As with many big data problems, you can split up the big task into several smaller tasks that are easier to handle. That is, instead of downloading data about all stocks at once, download the data in small batches of stocks consecutively.

Below I show how one can simplify the code dramatically (no batches!) and download the data faster (for me, it takes *less than a minute*) and with no significant burden on either the CPU or RAM.

An issue with translating the `lazy_tbl_to_pq()` function from the R version of my `db2pq` package is that there is no equivalent of `dbplyr` for Python. While Python Polars has its `LazyFrame`, the Tidy Finance task involves extracting data from a PostgreSQL database and Polars has no way of creating lazy data frames from database connections. The closest analogue is surely Ibis. So I added a function `ibis_to_pq()` to my Python package `db2pq` and in this note I take it for a spin.

I start by creating a connection to the WRDS PostgreSQL database. Here I follow the *Tidy Finance* approach closely. But so long as `wrds` is a `psycopg`-backed SQLAlchemy engine for the WRDS PostgreSQL database, you should be able to use whatever approach you are used to.

```
from sqlalchemy import create_engine
import os
from dotenv import load_dotenv

load_dotenv()

connection_string = (
    "postgresql+psycopg://"
    f"{os.getenv('WRDS_USER')}:{os.getenv('WRDS_PASSWORD')}"
```

```
    "@wrds-pgdata.wharton.upenn.edu:9737/wrds"
)

wrds = create_engine(connection_string, pool_pre_ping=True)
```

Having established `wrds`, I load in the packages I will be using.

```
import polars as pl
import ibis
from ibis import _
from db2pq import ibis_to_pq
```

Then I turn `wrds` into an Ibis instance with WRDS PostgreSQL as its **backend**. You can learn more about Ibis here.

```
db = ibis.postgres.from_connection(
    wrds.connect().connection.driver_connection
)
```

I then set up Ibis **lazy tables** for the three tables I will use here.[1]

```
dsf = db.table("dsf_v2", database="crsp")
stksecurityinfohist = db.table("stksecurityinfohist", database="crsp")
factors_daily = db.table("factors_daily", database="ff")
```

Ibis offers an "interactive" option that makes it easier to work with lazy tables.

```
ibis.options.interactive = True
```

Let's look at `ff3`:

```
factors_daily
```

```
  date         mktrf            smb              hml              rf               umd
  date         decimal(8, 6)    decimal(8, 6)    decimal(8, 6)    decimal(7, 5)    decimal(8, 6)
```

1. The R version of the note did not use `ff.factors_daily` from WRDS, but I do so here because I don't have the same `copy_inline()` functionality I had when using R in yesterday's note.

```
  1926-07-01        0.000900       -0.002500       -0.002700        0.00010           NULL
  1926-07-02        0.004500       -0.003300       -0.000600        0.00010           NULL
  1926-07-06        0.001700        0.003000       -0.003900        0.00010           NULL
  1926-07-07        0.000900       -0.005800        0.000200        0.00010           NULL
  1926-07-08        0.002200       -0.003800        0.001900        0.00010           NULL
  1926-07-09       -0.007100        0.004300        0.005700        0.00010           NULL
  1926-07-10        0.006100       -0.005300       -0.001000        0.00010           NULL
  1926-07-12        0.000400       -0.000300        0.006400        0.00010           NULL
  1926-07-13        0.004800       -0.002800       -0.002000        0.00010           NULL
  1926-07-14        0.000400        0.000700       -0.004300        0.00010           NULL
  …                        …               …               …              …              …
```

And then `dsf`:

```
dsf
```

```
  permno   hdrcusip    permco   siccd   nasdissuno   yyyymmdd   sharetype   securitytype   securit

  int32    string(8)   int32    int32   int32        int32      string(3)   string(4)      string(3

   10000   68391610      7952    3990        10396   19860107   NS          EQTY           COM
01-07   N                2.562500   BA                9430.00   BP                    NULL   NS
   10000   68391610      7952    3990        10396   19860108   NS          EQTY           COM
01-08   N                2.500000   BA                9200.00   BP                2.562500   BA
01-07          9430.00   PB                  -0.024390        -0.024390        0.000000   NA
   10000   68391610      7952    3990        10396   19860109   NS          EQTY           COM
01-09   N                2.500000   BA                9200.00   BP                2.500000   BA
01-08          9200.00   PB                   0.000000         0.000000        0.000000   NA
   10000   68391610      7952    3990        10396   19860110   NS          EQTY           COM
01-10   N                2.500000   BA                9200.00   BP                2.500000   BA
01-09          9200.00   PB                   0.000000         0.000000        0.000000   NA
   10000   68391610      7952    3990        10396   19860113   NS          EQTY           COM
01-13   N                2.625000   BA                9660.00   BP                2.500000   BA
01-10          9200.00   PB                   0.050000         0.050000        0.000000   NA
   10000   68391610      7952    3990        10396   19860114   NS          EQTY           COM
01-14   N                2.750000   BA               10120.00   BP                2.625000   BA
01-13          9660.00   PB                   0.047619         0.047619        0.000000   NA
   10000   68391610      7952    3990        10396   19860115   NS          EQTY           COM
01-15   N                2.875000   BA               10580.00   BP                2.750000   BA
01-14         10120.00   PB                   0.045455         0.045455        0.000000   NA
```

```
   10000   68391610      7952    3990        10396   19860116   NS          EQTY           COM
01-16   N                3.000000   BA                 11040.00   BP                2.875000   BA
01-15         10580.00   PB                    0.043478         0.043478         0.000000   NA
   10000   68391610      7952    3990        10396   19860117   NS          EQTY           COM
01-17   N                3.000000   BA                 11040.00   BP                3.000000   BA
01-16         11040.00   PB                    0.000000         0.000000         0.000000   NA
   10000   68391610      7952    3990        10396   19860120   NS          EQTY           COM
01-20   N                3.000000   BA                 11040.00   BP                3.000000   BA
01-17         11040.00   PB                    0.000000         0.000000         0.000000   NA
       …   …                 …        …           …          …   …           …              …
```

```
start_date = "1960-01-01"
end_date = "2024-12-31"
```

For clarity, in translating the original *Tidy Finance* query, I break out the `security` sub-query from the main query.

```
security = (
    stksecurityinfohist
    .filter(
        _.sharetype == "NS",
        _.securitytype == "EQTY",
        _.securitysubtype == "COM",
        _.usincflg == "Y",
        _.issuertype.isin(["ACOR", "CORP"]),
        _.primaryexch.isin(["N", "A", "Q"]),
        _.conditionaltype.isin(["RW", "NW"]),
        _.tradingstatusflg == "A",
    ).select(
        "permno",
        "secinfostartdt",
        "secinfoenddt",
    )
)
```

I next construct the main `crsp_daily` query.

```
crsp_daily = (
    dsf
    .filter(_.dlycaldt.between(start_date, end_date))
    .inner_join(security,
```

```
                dsf.permno == security.permno)
    .filter(_.dlycaldt.between(_.secinfostartdt, _.secinfoenddt))
    .select(
        permno=_.permno,
        date=_.dlycaldt,
        ret=_.dlyret,
    )
    .drop_null(["permno", "date", "ret"])
    .left_join(
        factors_daily
        .select("date", "rf")
        .rename(risk_free="rf"),
        _.date == factors_daily.date,
    )
    .mutate(
        ret_excess=ibis.greatest(_.ret - _.risk_free, -1)
    )
    .select(
        "permno", "date",
        ret=_.ret.cast("float64"),
        ret_excess=_.ret_excess.cast("float64"),
    )
)
```

Let's take a peek at `crsp_daily`, which is still a lazy table:

```
crsp_daily
```

```
┏━━━━━━━━┳━━━━━━━━━━━━┳━━━━━━━━━━━┳━━━━━━━━━━━━┓
┃ permno ┃ date       ┃ ret       ┃ ret_excess ┃
┡━━━━━━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━╇━━━━━━━━━━━━┩
│ int32  │ date       │ float64   │ float64    │
├────────┼────────────┼───────────┼────────────┤
│  10000 │ 1986-01-08 │ -0.024390 │  -0.024690 │
│  10000 │ 1986-01-09 │  0.000000 │  -0.000300 │
│  10000 │ 1986-01-10 │  0.000000 │  -0.000300 │
│  10000 │ 1986-01-13 │  0.050000 │   0.049700 │
│  10000 │ 1986-01-14 │  0.047619 │   0.047319 │
│  10000 │ 1986-01-15 │  0.045455 │   0.045155 │
│  10000 │ 1986-01-16 │  0.043478 │   0.043178 │
│  10000 │ 1986-01-17 │  0.000000 │  -0.000300 │
│  10000 │ 1986-01-20 │  0.000000 │  -0.000300 │
```

```
   10000   1986-01-21    0.000000    -0.000300
       …   …                     …            …
```

The next step is to create a Parquet file from `crsp_daily` using `ibis_to_pq()`.

```
%%time
ibis_to_pq(crsp_daily, "crsp_daily.parquet")
```

```
CPU times: user 50.4 s, sys: 7.05 s, total: 57.5 s
Wall time: 5min 57s

'crsp_daily.parquet'
```

For some reason, `ibis_to_pq()` does not perform as well as its sibling `lazy_tbl_to_pq()` in the R version of `db2pq`. This performance is a bit disappointing given that I had `ibis_to_pq()` use the ADBC driver rather than the default `psycopg` driver used by Ibis. I was surprised that Ibis did not support the Arrow driver.

```
%%time
ibis_to_pq(factors_daily, "factors_ff3_daily.parquet")
```

```
CPU times: user 10.3 s, sys: 529 ms, total: 10.8 s
Wall time: 20.6 s

'factors_ff3_daily.parquet'
```

1.1 Using the data

I figured that it would be a little dull to do no more than simply create the Parquet files. But looking through *Tidy Finance with Python*, the only use of `crsp_daily` seemed to be in the section *Estimating Beta Using Daily Returns*.

There *Tidy Finance* say:

> We then create a connection to the daily CRSP data, but we don't load the whole table into our memory. We only extract all distinct `permno` because we loop the beta estimation over batches of stocks with size 500. To estimate the CAPM over a consistent lookback window while accommodating different return frequencies, we adjust the minimum required number of observations accordingly. Specifically, we require at least 1,000 daily returns over a five-year period for a valid estimation. This threshold is consistent with the monthly requirement of 48 observations out of 60 months, given that there are roughly 252 trading days in a year.

The *Tidy Finance* code uses pandas, but I will use Python Polars, which does much better with larger data sets. I start by creating `LazyFrame` instances of the two tables that are used in calculating betas.

```
crsp_daily = pl.scan_parquet("crsp_daily.parquet")
factors_ff3_daily = pl.scan_parquet("factors_ff3_daily.parquet")
```

The next step is to implement the approach described above. Python Polars can handle the details of creating the windows and so on, but we don't really want to have to hand over all the data to Statsmodels to do the regressions.[2] But if you're reading *Tidy Finance with Python*, it seems reasonable to assume that you know how to calculate a univariate regression coefficient, so let's just use Polars expressions and do it by hand.

Unfortunately, it turns out that we will be calculating about 2.5 million betas over windows of up to about 1,250 days each and that will be a lot of data even for Polars![3]

So I ended up using batches of 500, just as *Tidy Finance* do. I make a function that calculates β for 500 `permno` values at a time. The key element here is `group_by_dynamic()`: For each `permno`, this creates a sequence of monthly time windows over date, then aggregates within each window. More specifically:

- `group_by="permno"` means each stock is handled separately.
- `every="1mo"` means start a new window every 1 month.
- `period="60mo"` means each window spans 60 months.
- `offset="-60mo"` shifts each window backward by 60 months relative to its label/start schedule, so each monthly output row looks back over the previous 60 months rather than forward.
- `closed="left"` means the left endpoint is included and the right endpoint is excluded.
- `label="right"` means the output date attached to each aggregated window is the right edge of the window.

So in plain English: For each stock, every month, compute summary statistics using up to 60 months of prior data, and label the result with the date at the right edge of that lookback window.

Tidy Finance do not provide any indication of how long their code takes to run, but let's see how long this takes to run:

```
%%time
min_obs = 1000
batch_size = 500

def get_betas_batch(permno_batch: pl.DataFrame) -> pl.LazyFrame:
    return (
```

2. The original pandas code uses `smf.ols()` from Statsmodels.
3. There will be a lot of overlap in these windows and I had thought that Polars would just take the data and just process the windows at the start of each month as it passed through for each `permno`, but it seems it literally stacks copies of data for each window and I quickly ran out of RAM when I tried this.

```
        crsp_daily
        .join(permno_batch.lazy(), on="permno", how="semi")
        .select("permno", "date", "ret_excess")
        .join(
            factors_ff3_daily.select(
                "date",
                pl.col("mktrf").cast(pl.Float64),
                pl.col("rf").cast(pl.Float64),
            ),
            on="date",
            how="inner",
        )
        .select(
            "permno",
            "date",
            "ret_excess",
            mkt_excess=pl.col("mktrf") - pl.col("rf"),
        )
        .sort("permno", "date")
        .group_by_dynamic(
            "date",
            every="1mo",
            period="60mo",
            offset="-60mo",
            group_by="permno",
            closed="left",
            label="right",
        )
        .agg(
            beta=pl.cov("mkt_excess", "ret_excess") / pl.col("mkt_excess").var(),
            mean_ret=pl.col("ret_excess").mean(),
            mean_mkt=pl.col("mkt_excess").mean(),
            n=pl.len(),
        )
        .filter(pl.col("n") >= min_obs)
        .with_columns(
            alpha=pl.col("mean_ret") - pl.col("beta") * pl.col("mean_mkt")
        )
        .drop("mean_ret", "mean_mkt")
    )

permnos = (
    crsp_daily
```

```
    .select("permno")
    .unique()
    .collect()
)

betas = pl.concat(
    (
        get_betas_batch(permnos.slice(i, batch_size)).collect()
        for i in range(0, permnos.height, batch_size)
    ),
    rechunk=True,
)
```

```
CPU times: user 28.8 s, sys: 12.5 s, total: 41.3 s
Wall time: 18 s
```

So about 15 seconds for the whole lot.

And let's take a peek at the data:

```
betas
```

permno i32	date date	beta f64	n u32	alpha f64
10230	1990-05-01	0.467184	1005	-0.00063
10230	1990-06-01	0.459434	1027	-0.000721
10230	1990-07-01	0.462415	1048	-0.000712
10230	1990-08-01	0.469646	1069	-0.00084
10230	1990-09-01	0.432908	1092	-0.000953
…	…	…	…	…
93436	2025-09-01	1.971018	1090	0.000748
93436	2025-10-01	1.924259	1069	0.000801
93436	2025-11-01	1.937688	1047	0.000863
93436	2025-12-01	1.953512	1027	0.00071
93436	2026-01-01	1.949638	1005	0.000583