Using DuckDB with WRDS: Python version

Ian D. Gow

2025-01-23

1 Introduction

In this note, I demonstrate how one can use the Ibis package in Python to get data from a PostgreSQL database and some advantages of using Ibis over a more traditional approach of using SQL to create Pandas data frames and then using Pandas for further data processing. As the working example, I focus on the creation of "daily CRSP data" as found in *Tidy Finance for Python*.

I show that using Ibis has a number of advantages over the approach used in Tidy Finance.

- Data retrieval is faster
- The code used is simpler and more "Pythonic"
- The code created with Ibis facilitates shifting from remote PostgreSQL data to a local data source (e.g., parquet files)
- Working with the final data is a lot easier and more performant

This note was written using Quarto and compiled with RStudio, an integrated development environment (IDE) for working with R and, to some extent, Python. The source code for this note is available here and the latest version of this PDF is here.

2 Getting data

Chapter 3 of *Tidy Finance for Python* describes the process for collecting daily stock data from CRSP and combining it with security-level data from CRSP and data on market performance from Fama-French.

The daily CRSP data set is one of two data sets used in *Tidy Finance for Python* that are large enough to create any concerns in terms of resources for working with those data (e.g., network bandwidth, storage, and RAM). This makes daily CRSP an excellent candidate for further analysis here.

2.1 General set-up

Because this note uses a number of different approaches to a set of problems, there are many packages I need to load:

```
import pandas as pd
import numpy as np
import duckdb
import pandas_datareader as pdr
import timeit
import os
import sqlite3
import ibis
from ibis import _
from os.path import join
from sqlalchemy import create_engine
from db2pq import wrds_update_pq
```

I follow Tidy Finance in focusing on data for the year 1960 to 2023.

```
start_date = "1960-01-01"
end_date = "2023-12-31"
```

2.2 Fama-French data

To keep this note self-contained, here I include code to get the Fama-French data used below. Note that this code is similar to code included in the Chapter 2 of *Tidy Finance with Python*.

If you are a reader of *Tidy Finance with Python* and already have the SQLite database created by running the code in Chapter 2 of that book, you can skip ahead to Section 2.3.

```
factors_ff3_daily_raw = pdr.DataReader(
  name="F-F_Research_Data_Factors_daily",
  data_source="famafrench",
  start=start_date,
  end=end_date)[0]

factors_ff3_daily = (factors_ff3_daily_raw
  .divide(100)
  .reset_index(names="date")
  .rename(str.lower, axis="columns")
  .rename(columns={"mkt-rf": "mkt_excess"})
)
```

```
if not os.path.exists("data"):
  os.makedirs("data")

tidy_finance = sqlite3.connect(database="data/tidy_finance_python.sqlite")
factors_ff3_daily.to_sql(name="factors_ff3_daily",
                         con=tidy_finance,
                         if_exists="replace",
                         index=False)
```

```
16108
```

2.3 Daily CRSP data – Tidy Finance version

Tidy Finance say "while [earlier data sets] can be downloaded in a meaningful amount of time, this is usually not true for daily return data. The daily CRSP data file is substantially larger than monthly data and can exceed 20 GB." "

Tidy Finance suggest that "this has two important implications: you cannot hold all the daily return data in your memory (hence it is not possible to copy the entire dataset to your local database), 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 Python session."

The solution proposed by Tidy Finance is to "split up the big task into several smaller tasks that are easier to handle." The following code is more or less copy-pasted from *Tidy Finance with Python*. The main change I made was to increase the batch size from 500 to 2,000, as doing so has little impact on performance (for me), but reduces the lines of output.

Note that I omit the code used to set the `WRDS_USER` environment variable, but you can set that using code provided by Tidy Finance on their website.

```
tic = timeit.default_timer()

connection_string = (
  "postgresql+psycopg2://"
 f"{os.getenv('WRDS_USER')}"
  "@wrds-pgdata.wharton.upenn.edu:9737/wrds"
)

wrds = create_engine(connection_string, pool_pre_ping=True)

factors_ff3_daily = pd.read_sql(
  sql="SELECT * FROM factors_ff3_daily",
```

```
  con=tidy_finance,
  parse_dates={"date"}
)

permnos = pd.read_sql(
  sql="SELECT DISTINCT permno FROM crsp.stksecurityinfohist",
  con=wrds,
  dtype={"permno": int}
)

permnos = list(permnos["permno"].astype(str))

batch_size = 2000
batches = np.ceil(len(permnos)/batch_size).astype(int)

for j in range(1, batches+1):

  permno_batch = permnos[
    ((j-1)*batch_size):(min(j*batch_size, len(permnos)))
  ]

  permno_batch_formatted = (
    ", ".join(f"'{permno}'" for permno in permno_batch)
  )
  permno_string = f"({permno_batch_formatted})"

  crsp_daily_sub_query = (
    "SELECT dsf.permno, dlycaldt AS date, dlyret AS ret "
      "FROM crsp.dsf_v2 AS dsf "
      "INNER JOIN crsp.stksecurityinfohist AS ssih "
      "ON dsf.permno = ssih.permno AND "
         "ssih.secinfostartdt <= dsf.dlycaldt AND "
         "dsf.dlycaldt <= ssih.secinfoenddt "
      f"WHERE dsf.permno IN {permno_string} "
           f"AND dlycaldt BETWEEN '{start_date}' AND '{end_date}' "
            "AND ssih.sharetype = 'NS' "
            "AND ssih.securitytype = 'EQTY' "
            "AND ssih.securitysubtype = 'COM' "
            "AND ssih.usincflg = 'Y' "
            "AND ssih.issuertype in ('ACOR', 'CORP') "
            "AND ssih.primaryexch in ('N', 'A', 'Q') "
            "AND ssih.conditionaltype in ('RW', 'NW') "
            "AND ssih.tradingstatusflg = 'A'"
```

```
  )

  crsp_daily_sub = (pd.read_sql_query(
      sql=crsp_daily_sub_query,
      con=wrds,
      dtype={"permno": int}
    )
    .dropna()
   )

  crsp_daily_sub['date'] = pd.to_datetime(crsp_daily_sub.date,
                                          format='%Y%m%d%H%M%S')

  if not crsp_daily_sub.empty:

      crsp_daily_sub = (crsp_daily_sub
        .merge(factors_ff3_daily[["date", "rf"]],
               on="date", how="left")
        .assign(
          ret_excess = lambda x:
            ((x["ret"] - x["rf"]).clip(lower=-1))
        )
        .get(["permno", "date", "ret_excess"])
      )

      if j == 1:
        if_exists_string = "replace"
      else:
        if_exists_string = "append"

      crsp_daily_sub.to_sql(
        name="crsp_daily",
        con=tidy_finance,
        if_exists=if_exists_string,
        index=False
      )

  print(f"Batch {j} out of {batches} done ({(j/batches)*100:.2f}%)")

toc = timeit.default_timer()
toc - tic
```

```
Batch 1 out of 19 done (5.26%)
```

```
Batch 2 out of 19 done (10.53%)
Batch 3 out of 19 done (15.79%)
Batch 4 out of 19 done (21.05%)
Batch 5 out of 19 done (26.32%)
Batch 6 out of 19 done (31.58%)
Batch 7 out of 19 done (36.84%)
Batch 8 out of 19 done (42.11%)
Batch 9 out of 19 done (47.37%)
Batch 10 out of 19 done (52.63%)
Batch 11 out of 19 done (57.89%)
Batch 12 out of 19 done (63.16%)
Batch 13 out of 19 done (68.42%)
Batch 14 out of 19 done (73.68%)
Batch 15 out of 19 done (78.95%)
Batch 16 out of 19 done (84.21%)
Batch 17 out of 19 done (89.47%)
Batch 18 out of 19 done (94.74%)
Batch 19 out of 19 done (100.00%)


331.9219591250876
```

So the code takes about 5 minutes and the result is a table `crsp_daily` inside an SQLite database.

2.4 Daily CRSP data – Ibis/DuckDB version

I now generate the same data table, but using Ibis to connect to PostgreSQL instead of using the Python package `sqlachemy`, which in turn uses the `psycopg2` package. Note I could use Ibis with a PostgreSQL backend directly, but I instead use a DuckDB backend and (implicitly) use the `postgres` extension for DuckDB to access PostgreSQL data within DuckDB. I say "implicitly" because the Ibis package is taking care of most of the details for me.

If you are working through this code interactively, you might find it helpful to set the following option:

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

```
tic = timeit.default_timer()
```

I begin by connecting to an in-memory DuckDB database, which is created with the following command:

```
con = ibis.duckdb.connect()
```

Rather than loading the `factors_ff3_daily` data from the SQLite database into a Pandas data frame, I load it into my DuckDB database. The following code is very similar to the above apart from the use of `con.read_sqlite()` in place of `pd.read_sql()`.

```
factors_ff3_daily = con.read_sqlite(
    path="data/tidy_finance_python.sqlite",
    table_name="factors_ff3_daily")
```

Again rather than connecting to the WRDS PostgreSQL database directly using `sqlalchemy`, I connect via my DuckDB database. Note that this requires a *slightly* different form for the URI (i.e., `postgres://` in place of `postgresql+psycopg2://`) and using `con.read_postgres()` in place of `pd.read_sql()`.

```
wrds_uri = (
  "postgres://"
 f"{os.getenv('WRDS_USER')}"
  "@wrds-pgdata.wharton.upenn.edu:9737/wrds"
)
```

```
dsf = con.read_postgres(wrds_uri,
                        table_name="dsf_v2", database="crsp")

ssih = con.read_postgres(wrds_uri,
                         table_name="stksecurityinfohist", database="crsp")
```

Now the *pièce de résistance*: I create `crsp_daily` using the following code. There are a few things to note about this code. First, creating it involves fairly trivial translation of the SQL above. For example, `ssih.issuertype IN ('ACOR', 'CORP')` becomes the more Pythonic `ssih.issuertype.isin(['ACOR', 'CORP'])` and `ssih.securitytype = 'EQTY'` sees the SQL equality operator replaced by the Pythonic `==`. But if you can read the SQL, you should be able to understand this code quite easily.

Note that the code above used Pandas to do some calculations that could have been performed using SQL. Specifically, the `.clip(lower=-1)` could be `CASE WHEN ret_excess < -1 THEN -1 ELSE ret_excess` and the `(_.ret_excess < -1).ifelse(-1, _.ret_excess))` is actually implemented by Ibis in this way.[1]

1. Note that "clipping" the excess returns in this way has no justification in financial economics. *Excess* returns are not bounded by -1 in the way that returns on unleveraged long positions in limited-liability securities are. I implement this here to be consistent with Tidy Finance and also to illustrate how this can be done using Ibis.

```
crsp_daily = (
  dsf
  .inner_join(ssih, "permno")
  .filter(ssih.secinfostartdt <= dsf.dlycaldt,
          dsf.dlycaldt >= start_date,
          dsf.dlycaldt <= end_date,
          ssih.sharetype == 'NS',
          ssih.securitytype == 'EQTY',
          ssih.securitysubtype == 'COM',
          ssih.usincflg == 'Y',
          ssih.issuertype.isin(['ACOR', 'CORP']),
          ssih.primaryexch.isin(['N', 'A', 'Q']),
          ssih.conditionaltype.isin(['RW', 'NW']),
          ssih.tradingstatusflg == 'A')
  .rename(date="dlycaldt", ret="dlyret")
  .left_join(factors_ff3_daily, "date")
  .mutate(ret_excess = _.ret - _.rf)
  .mutate(ret_excess = (_.ret_excess < -1).ifelse(-1, _.ret_excess))
  .select("permno", "date", "ret_excess"))
```

Rather that "downgrading" the data to an SQLite table, I will just export the data to a parquet file.[2]

```
crsp_daily.to_parquet(path = "data/crsp_daily.parquet")
```

```
toc = timeit.default_timer()
print(f"Elapsed time: {(toc - tic):.2f} seconds.")
```

```
Elapsed time: 127.19 seconds.
```

So one benefit is seen immediately: the code runs in about 2 minutes, or about 2.5 times faster than the original Tidy Finance code. A second benefit is that the code is much simpler (and shorter). There is no need to create `permnos`, run a `for`-loop, specify batch sizes, or blend SQL with Python variables using **f-strings**. A final benefit is that this code is more versatile, which we illustrate in a moment.

While we don't need `permnos` here, we could create it easily using the following code:

```
permnos = (ssih
  .select("permno")
  .distinct()
```

2. I have written on the topic of creating a repository of Parquet data files in Appendix E of *Empirical Research in Accounting: Tools and Methods*.

```
  .to_pandas())

permnos = list(permnos["permno"].astype(str))
```

2.5 Getting original data tables

Rather than creating a special local "daily CRSP" file, we could instead get and retain the original data files. An advantage of this approach is that it means we can write code that creates data tables directly off WRDS data files without requiring minutes to compute.

I have used two main approaches to maintaining a local data repository:

1. PostgreSQL database. In 2011, long before WRDS had its own PostgreSQL database, I wrote Perl scripts to export data from SAS files on the WRDS server to tables in a local PostgreSQL database. These Perl scripts eventually became the `wrds2pg` Python package.[3] I wrote more about creating such a database in Appendix D of *Empirical Research in Accounting: Tools and Methods*.]
2. Folders of parquet files. More recently I have used a local repository of parquet files as an alternative to the WRDS PostgreSQL database. While I have more about creating such a repository in Appendix E of *Empirical Research in Accounting: Tools and Methods*, here I discuss how we can get just the two files we need from WRDS for our current exercise. This requires just the function `wrds_update_pq()` from my Python package `db2pq`.[4]

```
from db2pq import wrds_update_pq
```

The `wrds_update_pq()` function keys of the environment variable `DATA_DIR` and stores WRDS tables as files organized into directories that match the schemas in the WRDS PostgreSQL database. I start by getting `crsp.dsf_v2`, which is about 29 GB in SAS form and 24 GB in PostgreSQL form on the WRDS servers. As can be seen, getting `crsp.dsf_v2` requires a single line of code.[5]

```
tic = timeit.default_timer()
wrds_update_pq("dsf_v2", schema='crsp')
toc = timeit.default_timer()
print(f"Elapsed time: {(toc - tic):.2f} seconds.")
```

```
Updated crsp.dsf_v2 is available.
Getting from WRDS.
Beginning file download at 2025-01-23 19:03:13 UTC.
```

3. Run `pip install wrds2pg` to install this package.
4. Run `pip install db2pq` to install this package.
5. The lines around this line are there just to record the time needed to do this.

```
FloatProgress(value=0.0, layout=Layout(width='auto'), style=ProgressStyle(bar_color='black'))


Completed file download at 2025-01-23 19:24:41 UTC.

Elapsed time: 1293.21 seconds.
```

It takes me about 22 minutes to get the data.[6] The resulting parquet file is 4.35 GB due to the compression and efficient storage offered by the parquet format. This compares not too favourably with the 2.7 GB for an SQLite database containing just these two tables, which is a cut-down version of the data created by Tidy Finance to save on space.

Note that the `wrds_update_pq()` checks the "last updated" value for the SAS data file on WRDS and compares that with the corresponding metadata in any parquet file in the existing location and only gets data from the PostgreSQL server if the last updated value has changed. The WRDS subscription at my institution only offers annual updates, to the 20-minute download of `crsp.dsf_v2` only needs to happen once a year. Another feature of `wrds_update_pq()` from the `db2pq` package is that I wrote it to be very sparing in its use of RAM in creating the parquet file.

Getting `crsp.stksecurityinfohist` takes only a few seconds because the underlying data table is much smaller.

```
tic = timeit.default_timer()
wrds_update_pq("stksecurityinfohist", schema="crsp")
toc = timeit.default_timer()
print(f"Elapsed time: {(toc - tic):.2f} seconds.")
```

```
Updated crsp.stksecurityinfohist is available.
Getting from WRDS.
Beginning file download at 2025-01-23 19:24:46 UTC.

FloatProgress(value=0.0, layout=Layout(width='auto'), style=ProgressStyle(bar_color='black'))


Completed file download at 2025-01-23 19:25:26 UTC.

Elapsed time: 45.37 seconds.
```

6. This is on a MacBook Pro with M1 Pro chip and a 1 Gbps internet connection.

2.6 Using local data files

So now that I have local copies of `crsp.dsf_v2` and `crsp.stksecurityinfohist`, I can use those rather than going back to the WRDS PostgreSQL server to get data every time. Much of the code is unchanged from that in Section 2.4.

```
tic = timeit.default_timer()
con = ibis.duckdb.connect()
```

```
factors_ff3_daily = con.read_sqlite(
    path="data/tidy_finance_python.sqlite",
    table_name="factors_ff3_daily")
```

The only real difference in the code is in the lines creating `dsf` and `ssih`. Before I used `con.read_postgres()` to connect to the WRDS PostgreSQL server. Now I use `con.read_parquet()` to connect to local copies in parquet form.

```
data_dir = os.environ["DATA_DIR"]

dsf = con.read_parquet(join(data_dir, "crsp", "dsf_v2.parquet"))
ssih = con.read_parquet(join(data_dir, "crsp", "stksecurityinfohist.parquet"))
```

The following code is identical to the code in Section 2.4. But now it takes 3 or 4 seconds to run rather than the 120 or so seconds needed earlier.

```
crsp_daily = (
  dsf
  .inner_join(ssih, "permno")
  .filter(ssih.secinfostartdt <= dsf.dlycaldt,
          dsf.dlycaldt >= start_date,
          dsf.dlycaldt <= end_date,
          ssih.sharetype == 'NS',
          ssih.securitytype == 'EQTY',
          ssih.securitysubtype == 'COM',
          ssih.usincflg == 'Y',
          ssih.issuertype.isin(['ACOR', 'CORP']),
          ssih.primaryexch.isin(['N', 'A', 'Q']),
          ssih.conditionaltype.isin(['RW', 'NW']),
          ssih.tradingstatusflg == 'A')
  .rename(date="dlycaldt", ret="dlyret")
  .left_join(factors_ff3_daily, "date")
```

```
  .mutate(ret_excess = _.ret - _.rf)
  .mutate(ret_excess = (_.ret_excess < -1).ifelse(-1, _.ret_excess))
  .select("permno", "date", "ret_excess"))
```

```
crsp_daily.to_parquet(path = "data/crsp_daily_alt.parquet")

toc = timeit.default_timer()
print(f"Elapsed time: {(toc - tic):.2f} seconds.")
```

```
Elapsed time: 4.04 seconds.
```

While the field I know best—empirical accounting research—seems to be somewhere between uninterested and hostile to ideas of reproducibility (perhaps because of the implications for the p-hacking that dominates the field), readers of Tidy Finance are more likely to be in finance, which seems to be embracing reproducibility more enthusiastically. Being able to run code quite quickly (e.g., against local parquet files) that can be then passed along to a data editor who can run the same code against a standard data set (e.g., against the WRDS PostgreSQL server) has a lot to recommend it.

2.7 Some observations on using Python versus R

In some ways this note covers ground explored in an earlier note that used R. My impression is that some data analysts perceive Python to be somehow cooler and its users to be smarter than is the case for R, and *a fortiori* for the Tidyverse. But I believe this thinking is misguided.

In a lot of ways doing data analysis with the Tidyverse is more facile than doing the same with Python.

- The Tidyverse feels snappier than Ibis. In situations where there is very little latency with remote data frames using `dplyr` (actually `dbplyr` behind the scenes), the Ibis equivalents would take a few seconds to generate. While not a huge deal, I feel that building queries up using `dplyr` is probably a better way of writing SQL for many users and this snappiness adds to the quality of life for a data analyst.
- At times the Tidyverse code is a little less clunky to write and look at. **Non-standard evaluation** means I can write `select(permno, date, ret_excess)` instead of `select("permno", "date", "ret_excess")` and `mutate(ret_excess = ret - rf)` instead of `from ibis import _` followed by `mutate(ret_excess = _.ret- _.rf)`.
- Better workflow for creating documents. Python seems to work best with notebooks. Others have spoken about problems with notebooks (e.g., a YouTube presentation by Joel Grus). I think my issues are related, but are due more to my coming from a different workflow based on

Quarto. In the "Quarto workflow" (as I use it), there is a clear distinction between source code and output. This is particularly clear when committing edits using Git.

I don't like how output and source code get mixed up in Jupyter notebooks. While one can write scripts to scrub output from notebooks before committing them, at some point it can be useful to have output committed to Git as well. Spliting the source code (`.qmd`) and output (say, `.pdf`) into two documents solves this problem.

Some aspects of my Quarto-oriented workflow did not carry over from R to Python. For example, `knitr` (the package used in compiling R-based Quarto documents) offers caching at the level of a code chunk, while Jupyter appears to only offer document-level caching. This means I can tweak small portions of the R code without triggering costly recalculations. One solution to this problem in Python would be to switch to more of an interactive notebook-based approach, but this introduces all the issues raised by Joel Grus (e.g., I need to be thinking about what code I've run previously in a sesssion).

- Better learning curve. A data analyst learning Python probaly starts out with Pandas. Moving up to larger data sets probably means learning a new language (e.g., SQL) or at least a new package (e.g., Ibis). In contrast, I think that a data analyst starting with R would learn Tidyverse approaches using local data frames and then "move up" to using `dbplyr` using remote data frames. That's exactly the progression I use in *Empirical Research in Accounting: Tools and Methods*: from the introduction in Chapter 2 through to the introduction to remote data frames in Chapter 6, there is barely a shift in gears.[7]

  In terms of languages, learning Pandas might be like learning Italian and then moving onto French when learning Ibis (SQL is German?). In contrast, knowing `dplyr` is like being a native speaker of American English when it comes to moving to `dbplyr` (standard British English). A few words are different, but you can understand pretty much everything and quickly start calling the local currency "pounds" not "dollars".

- R is more purpose-built for data analysis. In both R and Python, there are the age-old issues of dependencies and code that works with one version of a package, but not another. But these issues seem heightened with Python. In writing *Empirical Research in Accounting: Tools and Methods* over several years, I would occasionally come across code that needed updating to work with the latest version or inconsistencies between packages (e.g., for a time, one package required an older version of another). But these issues seem to have diminished over time as (for example) the Tidyverse has matured and become quite stable. And my impression is that these issues are more acute with Python.

  Almost any resource on Python will talk about setting up virtual environments and how to deal with multiple versions of Python on your computer. I feel that R is easier to navigate for the more casual user who wants to run an analysis and isn't looking to put code into a production system or into a time capsule. One *can* worry about these things, but I think that one has much

7. In this regard, I strongly disagree with the recommendation of Philipp Hauber, a data scientist at *The Economist*, to start with the "no-frills tutorial by Norm Matloff" given the orientation of that resource to "base R". I think *Empirical Research in Accounting: Tools and Methods* demonstrates that you can go quite far with very little base R–specific knowledge.

less need to do so with R. Roughly 99% of the time one can have just use system-level versions of packages that are whatever is the latest on CRAN. For one there's no worry about messing up the "system version" of R as there is with Python.[8]

Note that the above is by no means meant to claim that R is uniformly superior to Python, but simply that R has advantages and that these advantages are greater in certain situations. While I have never taken a course in computer science or data analysis, I have experience with both R and Python and find myself reaching for R in many cases and for Python in other cases. Sometimes I might use Python for one part of a problem and R for another part.

8. Though MacOS, for one, makes this kind of problem much harder to cause than it used to be.