Data management ideas for researchers

Ian D. Gow

25 February 2026

Abstract

My sense is that data management is a challenge for researchers. In an academic context, some fields may receive greater institutional support than others. My experience in business schools was that there was very little support for data curation and management. While many of the ideas I discuss here are general in nature, for concreteness, I focus on the special case of a WRDS user maintaining a local Parquet data library of the kind discussed in Appendix E of *Empirical Research in Accounting: Tools and Methods* and provide examples using my Python package `db2pq`.

1 Introduction

My sense is that data management is a challenge for researchers. In an academic context, some fields may receive greater institutional support than others. My experience in business schools was that there was very little support for data curation and management. While data curation is often inextricably linked with data management, in this note I assume that we are working with curated data and just need to "manage" it in some way. A classic example of (mostly) curated data in an academic context is WRDS data, which my Python package `db2pq` provides tools for managing a local library of such data.

2 Data management principles

2.1 Caveats

At the outset, I should note some limitations to my discussion here.

First, this note does not data at the scale of multiple terabytes. Researchers working with data at the scale that one starts thinking about Spark clusters and immense cloud storage will not find any answers here. That said, I think the approaches I cover here scale up to a "low terabyte" scale, at least for aggregate data.

Second, with very few exceptions, I have not really dealt with data with confidentiality issues. Readers dealing with sensitive data would need to overlay the necessary safeguards and protocols associated with such data onto the discussion I provide here.

2.2 Some concepts in data management

2.2.1 Scope

Many datasets are **project-specific** datasets, meaning that only have use within a single project (e.g., paper). Examples of project-specific would include experimental data generated in a particular study.

Other datasets are **general-purpose** datasets, meaning that they contain data that might be relevant to many studies. Classic examples in a business-school context would be the US stock price files offered by the Center for Research in Security Prices, LLC (CRSP) or financial statement data provided by Compustat, or economic time-series data provided by various statistical offices around the world.

Other datasets are **project-level** datasets, meaning that the particular data sets are somehow frozen for a particular project, even though the nature of the data otherwise puts them in the category of general-purpose datasets. For example, I might want to fix on a particular version of `comp.g_funda`, Compustat's global dataset for annual financial statements for my project, even though this dataset has relevance beyond a specific project.[1]

There are two reasons for having project-level that I can think of. The first reason arises in the context of reproducibility. If I have published a paper, then the replication package for that paper should ideally contain the data used to produce the exact results in the paper. For this purpose, if the paper used `comp.g_funda` data, then the ideal replication package would include the precise *project-level* version of that data set used to produce the paper. Of course, in reality, one cannot simply post the project-level version of `comp.g_funda` as part of a public replication package. Nonetheless, the authors themselves should have a project-level version of the dataset that *they* retain. This much aligns with the views of Welch (2019), who suggests that "the author should keep a private copy of the full data set with which the results were obtained."

The second reason is related to the first, but in some ways opposite in spirit. Some authors do not want their results to be upset by updates to any of datasets used to produce them. On one research project, I was responsible for supplying a curated data set of significant scale and complexity. Unfortunately, my understanding of variable scoping in Perl meant that about 2% of the data were simply corrupted and I felt I had to fix this. At my end, I wanted to manage the data as a general-purpose data set, but my co-author wanted to stick to the earlier project-level data.[2]

1. As academic researchers generally get Compustat data through Wharton Research Data Services (WRDS), I refer to this dataset using the nomenclature used by WRDS. Here "global" means "not the United States (or Canada)".
2. Thre can be reasonable explanations for my co-author's stance. From some discussions I've had, it seems that many authors are worried about reviewers querying any change in any number in any table. While I do not understand why "because I updated Compustat" wouldn't be an adequate response to "why did the coefficients change?" query, I guess many authors put a high priority on triggering as few questions as possible in a far-from-perfect review process. Another reason for this stance is that many researchers have a very manual research process, so changing an input data set

As far as WRDS data are concerned, my `db2pq` package aims to facilitate managing WRDS data either as general-purpose data or as project-level data.[3] On my computers, I have the environment variable `DATA_DIR` set to a location inside Dropbox. So, by default, new WRDS data will go in the matching schema (i.e., subdirectory) of the directory indicated by `DATA_DIR`. In Python, I can inspect the value in `DATA_DIR`:

```
import os
os.environ['DATA_DIR']
```

```
'/Users/igow/Dropbox/pq_data'
```

For this note, I'm using the very latest version of `db2pq`, which you can install by running `pip install --upgrade db2pq` from the command line. If you do not have pandas or `paramiko` installed, you should do `pip install --upgrade "db2pq[sas,pandas]"`. See the db2pq PyPI homepage for more details.

Within Python, you can check the version using the following.[4]

```
import db2pq
db2pq.__version__
```

```
'0.2.1.dev0'
```

The core function of `db2pq` is `wrds_update_pq()` If I ask `wrds_update_pq()` to update my *general-purpose* version `crsp.dsi`, I can see that the latest data on WRDS are no more recent than what I already have, so no update occurs.

```
from db2pq import wrds_update_pq
wrds_update_pq("dsi", "crsp")
```

```
crsp.dsi already up to date.
```

But, if I wanted a *project-level* version of `crsp.dsi`, I can specify the project-level data directory (`"data"`) and WRDS will update the data there. As I don't have any data in that folder to begin with, an "update" occurs.

means changing many other things, including re-typing the coefficients in the Word document containing the paper or re-exporting the data to Excel to make any plots.

3. WRDS data are not project-specific data sets pretty much by definition.
4. This will not work if you have a version of `db2pq` older than `0.2.0`, in which case you should reinstall it.

```
from db2pq import wrds_update_pq
wrds_update_pq("dsi", "crsp", data_dir="data")
```

```
Updated crsp.dsi is available.
Beginning file download at 2026-02-25 16:38:54 UTC.
Completed file download at 2026-02-25 16:38:56 UTC.
```

2.2.2 Version control

Version control is a thorny issue with data. As far as I know there is no equivalent of Git for datasets.[5] While I am sure that version control of data is a big issue in many contexts (e.g., data for regulated bodies), many data providers, even commercial vendors, often do a poor job of version control.

Many data sources will provide the current version of any given dataset and nothing else. For example, there is no way to get the version of the data you downloaded from WRDS two years ago if you want to understand why results have changed. In practice, researchers need to do any archiving of WRDS data sets themselves.

My `db2pq` Python package provides some functions that make it more convenient to maintain archives of previous version of tables from WRDS. The core function for maintaining a local repository of Parquet files based on WRDS data is `wrds_update_pq()`. This function has an `archive` argument that, if set to `True`, causes any existing data in the repository to be archived when an update is available and is applied:

```
wrds_update_pq("company", "comp", archive=True)
```

```
Updated comp.company is available.
Beginning file download at 2026-02-25 16:38:57 UTC.
Completed file download at 2026-02-25 16:38:59 UTC.
```

For most tables on the WRDS database, it appears that "last updated" metadata is included in table comments. The `wrds_update_pq()` function will, by default, extract that metadata and embed it as metadata in the Parquet files.

The `pq_last_modified()` function, if given a `table_name` argument, will by default return the metadata embedded in the Parquet file.

```
from db2pq import pq_last_modified
pq_last_modified(table_name="company", schema="comp")
```

5. I'd guess that such a thing would amount to the equivalent of SQL's `INSERT`, `UPDATE`, and `DELETE` commands.

```
'Company (Updated 2026-02-25)'
```

But if I specify `archive=True`, then `pq_last_modified()` will instead return a pandas data frame containing information about (possibly several) files matching the specified `table_name` in the archive.[6] Here, we see that I have four previous versions of `comp.company` in my archive.

```
pq_df = pq_last_modified(table_name="company", schema="comp", archive=True)
pq_df[["file_name", "last_mod"]]
```

	file_name	last_mod
0	company_20240614T062835Z	2024-06-14 02:28:35-04:00
1	company_20260105T070000Z	2026-01-05 02:00:00-05:00
2	company_20260107T070000Z	2026-01-07 02:00:00-05:00
3	company_20260209T070000Z	2026-02-09 02:00:00-05:00
4	company_20260218T070000Z	2026-02-18 02:00:00-05:00
5	company_20260224T070000Z	2026-02-24 02:00:00-05:00
6	company_20260225T000000Z	2026-02-24 02:00:00-05:00
7	company_20260225T070000Z	2026-02-25 02:00:00-05:00

I can use the function `pq_restore()` to make the one from `2024-06-14` the one that I am using for my data repository.

```
from db2pq import pq_restore
pq_restore("company_20240614T062835Z", "comp")
```

```
'/Users/igow/Dropbox/pq_data/comp/company.parquet'
```

I now see that this is the version used when I look for `company` in the `comp` schema:

```
pq_last_modified(table_name="company", schema="comp")
```

```
'Last modified: 06/14/2024 02:28:35'
```

One thing you will notice is that the format of the "last modified" string has changed from the one above. This could be for one of three reasons:

6. If `table_name` is omitted and `schema` is specified, then the function will return a data frame with information on the files in the data directory for the schema (if `archive` is `False`, as is the default) or in the archive directory (if `archive` is `True`).

1. The Parquet file that has been restored was created using my other Python package, `wrds2pg`, which extracts data from WRDS's SAS data files. Naturally, it uses information returned by the SAS command `PROC CONTENTS`.
2. The Parquet file that has been restored was created using an earlier version of `db2pq`. Because WRDS did not initially store "last modified" information with its PostgreSQL tables, earlier version of `db2pq` retrieved information from the matching SAS file on the assumption that the SAS and PostgreSQL data sets would generally be aligned.
3. The Parquet file that has been restored was created using a recent version of `db2pq`, but with `use_sas=True`. In this case, `wrds_update_pq()` will retrieve the "last modified" information from the SAS file.[7]

By default, the `pq_restore()` function has `archive=True`, which means that any existing data file is archived.[8] We can see that the file that we created just moments ago using `wrds_update_pq()` is now in the archive:

```
pq_df = pq_last_modified(table_name="company", schema="comp", archive=True)
pq_df[["file_name", "last_mod"]]
```

	file_name	last_mod
0	company_20260105T070000Z	2026-01-05 02:00:00-05:00
1	company_20260107T070000Z	2026-01-07 02:00:00-05:00
2	company_20260209T070000Z	2026-02-09 02:00:00-05:00
3	company_20260218T070000Z	2026-02-18 02:00:00-05:00
4	company_20260224T070000Z	2026-02-24 02:00:00-05:00
5	company_20260225T000000Z	2026-02-24 02:00:00-05:00
6	company_20260225T070000Z	2026-02-25 02:00:00-05:00

If we update again with `archive=True`, we will effectively put the `2024-06-14` back in the archive and replace it with the current version on WRDS.

```
from db2pq import pq_archive
wrds_update_pq("company", "comp", archive=True)
```

```
Updated comp.company is available.
Beginning file download at 2026-02-25 16:39:00 UTC.
Completed file download at 2026-02-25 16:39:02 UTC.
```

7. The information returned by `PROC CONTENTS` is assumed to be expressed in US Eastern local time (i.e., `America/New_York`). The PostgreSQL comments generally only indicate a date, and the `db2pq` assumes that the update occurred at 02:00 US Eastern time.
8. In addition to `pg_restore()`, the `db2pq` package also offers `pq_archive()` and `pq_remove()` functions.

Some WRDS PostgreSQL tables appear not (yet) to have "last modified" information. For example, some RavenPack data tables appear not to have this information. In the following, I set `obs=100` and `data_dir="data"`, as I am doing this "update" purely for the purposes of this note, so only get 100 observations to keep things fast.

```
wrds_update_pq("rpa_entity_mappings", "ravenpack_common", obs=100,
               data_dir="data")
```

```
No comment found for ravenpack_common.rpa_entity_mappings.
ravenpack_common.rpa_entity_mappings already up to date.
```

We can confirm this using `pq_last_modified()`:

```
pq_last_modified(table_name="rpa_entity_mappings", schema="ravenpack_common",
                 data_dir="data")
```

```
''
```

In such cases, any subsquent call to `wrds_update_pq()` will not trigger an "update" because there is effectively nothing to allow it to confirm that the local data are not current.[9]

```
wrds_update_pq("rpa_entity_mappings", "ravenpack_common", obs=100,
               data_dir="data")
```

```
No comment found for ravenpack_common.rpa_entity_mappings.
ravenpack_common.rpa_entity_mappings already up to date.
```

In such cases, it makes sense to use the SAS data to determine the vintage of the data. However, a wrinkle in this case is that there is no SAS library called `ravenpack_common`. Instead the data are stored in the SAS library named `rpa`. So we also need to tell `wrds_update_pq()` where to get the SAS data.

```
wrds_update_pq("rpa_entity_mappings", "ravenpack_common", obs=100,
               data_dir="data", use_sas=True, sas_schema="rpa")
```

```
Updated ravenpack_common.rpa_entity_mappings is available.
Beginning file download at 2026-02-25 16:39:06 UTC.
Completed file download at 2026-02-25 16:39:09 UTC.
```

Now we have valid "last modified" data:

9. An update can always be forced using `force=True`.

```
pq_last_modified(table_name="rpa_entity_mappings", schema="ravenpack_common",
                 data_dir="data")
```

```
'Last modified: 02/09/2026 16:11:10'
```

So a subsequent call to `wrds_update_pq()` does not trigger an update, but for the correct reasons.

```
wrds_update_pq("rpa_entity_mappings", "ravenpack_common", obs=100,
               data_dir="data", use_sas=True, sas_schema="rpa")
```

```
ravenpack_common.rpa_entity_mappings already up to date.
```

2.2.3 Storage formats

While there are many storage formats available for data, I think a strong case can be made for Parquet being the default storage format for many users. If you use R or Python, I think the case is easy to make. Many software packages can read Parquet data and some of them (e.g., Polars or DuckDB) will absolutely fly with Parquet data.

I believe that recent editions of Stata can read Parquet files, though the way Stata operates means that Stata users are unlikely to see the performance benefits Parquet offers.[10] SAS users might find the case for Parquet less compelling, though there are probably benefits in moving to a storage medium that is more compact, less proprietary, and more likely to be supported in a few years' time.

Of course, an alternative to using Parquet files would be using a database, such as PostgreSQL. I think such systems have a lot of merit (and I have used PostgreSQL to store WRDS data since 2011), but I think they are more complicated for most users' needs and their benefits (e.g., shared access to data and rock-solid assurance) are less meaningful for most.

Another alternative is the CSV file, perhaps compressed. I think if one were sending data on the next Voyager mission, then CSV might be the chosen format.[11] Or if you really, really wanted data novices to inspect your data in Excel or Word, then CSV might be the go-to option. Or perhaps you want to put your data in a written form in a book for users to type in. For any other purpose with serious data needs, I think Parquet dominates.

One issue with CSV is that one is always dealing with type inference (string, integer, timestamp) and I think that type inference is one of those problems you want to solve once for any given dataset. For the WRDS data that is the focus of this note, I think CSV is to be avoided.

10. Of course, if a user cared about performance with data manipulation, he probably wouldn't be using Stata to begin with.
11. Each of the two Voyager spacecraft, launched by NASA in 1977, carry the Voyager Golden Record, a gold-plated copper phonograph record intended as a message to any intelligent extraterrestrial lifeforms that might encounter the probes. If we wanted to give data to such lifeforms, I think it would be (quoted) CSV data and written on paper.

2.2.4 Timestamps

Speaking of type inference, one bane of the existence of any data analyst might be timestamps. The usual purpose of timestamps is to identify a moment in time. For example, I want to know the precise time at which an earnings conference call happened, so I can turn to a dataset with intra-day data on quotes and trades to see how the market reacted. If the data on earnings conference call use UTC and the trade-and-quote data use US Eastern time and I ignore these differences, then I will be looking at times that are off by four or five hours (depending on the time of year).

To examine this issue, I'm going to look revisit the Call Report data I wrote about recently. I have these data in my (general-purpose) data repository and I can use the following function to load it into Polars.

```
import polars as pl
from pathlib import Path
import os

def ffiec_scan_pqs(schedule=None, *,
                   schema="ffiec", data_dir=None):
    if data_dir is None:
        data_dir = Path(os.environ["DATA_DIR"]).expanduser()

    path = data_dir / schema if schema else data_dir

    if schedule is None:
        raise ValueError("You must supply `schedule`.")
    files = list(path.glob(f"{schedule}_*.parquet"))
    if not files:
        raise FileNotFoundError(
          f"No Parquet files found for schedule '{schedule}' in {path}"
        )

    return pl.concat([pl.scan_parquet(f) for f in files])
```

In my earlier note, I discussed how I managed to infer that the timestamps on that dataset are in US Eastern time (`America/New_York`). We can inspect the data I have using the function above and focused on a single observation:

```
(
    ffiec_scan_pqs("por")
    .select("IDRSSD", "date", "last_date_time_submission_updated_on")
    .filter(pl.col("IDRSSD") == 37,
            pl.col("date") == pl.date(2023 , 12, 31))
```

```
    .collect()
)
```

IDRSSD	date	last_date_time_submission_updated_on
i32	date	datetime[μs, UTC]
37	2023-12-31	2024-01-10 18:43:43 UTC

WRDS offers essentially the same data in its `bank` schema. We can use `wrds_update_pq()` to get a sample of these data.[12]

```
wrds_update_pq("wrds_call_rcfa_1", "bank", data_dir = "data", obs=100)
```

```
Updated bank.wrds_call_rcfa_1 is available.
Beginning file download at 2026-02-25 16:39:13 UTC.
Completed file download at 2026-02-25 16:39:15 UTC.
```

To load the data into Polars, I will use the following small function:

```
def load_parquet(table, schema, *, data_dir=None):
  if data_dir is None:
    data_dir = Path(os.environ["DATA_DIR"]).expanduser()
  else:
    data_dir = Path(data_dir)

  path = data_dir / schema / f"{table}.parquet"
  return pl.scan_parquet(path)
```

As can be seen from the output below, the timestamp is off by five hours. That is because `wrds_update_pq()` assumes that timestamps are in UTC, which is a correct assumption for some data sets on WRDS, but is incorrect in this case.

```
rcfa_1 = load_parquet("wrds_call_rcfa_1", "bank", data_dir = "data")
(
    rcfa_1
    .select("rssd9001", "wrdsreportdate", "rssdsubmissiondate")
    .filter(pl.col("rssd9001") == 37,
            pl.col("wrdsreportdate") == pl.date(2023, 12, 31))
    .collect()
)
```

12. Because the data appear to be sorted by bank ID, I should retrieve the observation above, even though I'm only getting 100 rows of data.

rssd9001 i32	wrdsreportdate date	rssdsubmissiondate datetime[μs, UTC]
37	2023-12-31	2024-01-10 13:43:43 UTC

WRDS generally stores timestamps in PostgreSQL with type `TIMESTAMP WITHOUT TIME ZONE`, which is equivalent to saying "you figure out the time zone, user."[13] Because we know that the timestamps provided by the FFIEC are expressed in US Eastern time, we can tell `wrds_update_pq()` this using the `tz` argument:

```
# cache: true
wrds_update_pq("wrds_call_rcfa_1", "bank",
               data_dir = "data", obs=100,
               force=True, tz="America/New_York")
```

```
Forcing update based on user request.
Completed file download at 2026-02-25 16:39:17 UTC.
```

Now, we see that the data are correct.

```
rcfa_1 = load_parquet("wrds_call_rcfa_1", "bank")
(
    rcfa_1
    .select("rssd9001", "wrdsreportdate", "rssdsubmissiondate")
    .filter(pl.col("rssd9001") == 37,
            pl.col("wrdsreportdate") == pl.date(2023, 12, 31))
    .collect()
)
```

rssd9001 i32	wrdsreportdate date	rssdsubmissiondate datetime[μs]
37	2023-12-31	2024-01-10 13:43:43

In other cases, WRDS doesn't even bother to store the data as `TIMESTAMP WITHOUT TIME ZONE`, but instead the data are stored as strings. Here's one example.[14]

13. The only option that should be used with PostgreSQL is `TIMESTAMP WITH TIME ZONE`.
14. Here I use `keep` to limit my download to the fields of interest in this case.

```
# cache: true
wrds_update_pq("feed03_audit_fees", "audit",
               keep=["auditor_fkey", "file_accepted"],
               obs=5, data_dir="data")
```

```
Updated audit.feed03_audit_fees is available.
Beginning file download at 2026-02-25 16:39:18 UTC.
Completed file download at 2026-02-25 16:39:20 UTC.
```

But here we see that `file_accepted` is stored as a string (and `auditor_fkey` is a floating-point value).

```
load_parquet("feed03_audit_fees", "audit",
             data_dir="data").collect()
```

auditor_fkey f64	file_accepted str
5.0	”2001-03-28 00:00:00”
5.0	”2002-03-25 00:00:00”
4.0	”2003-03-31 09:19:45”
6.0	”2004-04-06 14:34:14”
6.0	”2005-04-04 11:55:05”

Fortunately, with `wrds_update_pq()`, I can specify the (Arrow) data types for selected columns and, in the case of `timestamp`, the time zone.

```
wrds_update_pq("feed03_audit_fees", "audit",
               col_types={"auditor_fkey": "int32",
                          "file_accepted": "timestamp"},
               tz="America/New_York", force=True,
               keep=["auditor_fkey", "file_accepted"],
               data_dir="data")
```

```
Forcing update based on user request.
Completed file download at 2026-02-25 16:39:22 UTC.
```

Now things look much better.

```
load_parquet("feed03_audit_fees", "audit",
             data_dir="data").head().collect()
```

auditor_fkey i32	file_accepted datetime[μs, UTC]
5	2001-03-28 05:00:00 UTC
5	2002-03-25 05:00:00 UTC
4	2003-03-31 14:19:45 UTC
6	2004-04-06 18:34:14 UTC
6	2005-04-04 15:55:05 UTC

2.3 Other ideas

There are several ideas not covered by this note currently, but that might be added later:

- Back up your data
- Modification of raw data files
- The WRDS web query

In the last case, don't use it! (I will explain why, but one issue is reproducibility.)

Welch, I., 2019. Editorial: An opinionated FAQ. Critical Finance Review 8, 19–24. https://doi.org/10.1561/104.00000077