Modern Research Computing

Open Source Tools for Economics, Finance, and Accounting

Ian D. Gow

Contents

Chapter 1

Introduction

This books describes a computing platform for research computing. The major characters in this book include a relational database (I focus on PostgreSQL), version control (using Git and hosting by GitHub or Bitbucket), and open source programming languages (I focus on R and Python). Other characters with minor roles include Google Sheets, Mechanical Turk, and Fiverr.

In addition to these characters, this book will emphasize a number of habits and processes that these all together. In some ways, the value is not in the tools listed above, but in how they are brought to bear on research problems.

1.1 A taxonomy of data

The emphasis in this book is on empirical research, which is largely about data. One challenge with understanding alternatives approaches to data is the fact that data comes in many forms and has a number of types. While many textbook treatments cover critical aspects of working with data, some of the complexity of data is often omitted from such discussions. To understand some of the issues, I will discuss a number of dichotomies that data can be understood to fall into.

1.1.1 Project-specific versus general data

For many researchers, the world might be divided into projects and each project might be understood as comprising a primary final output in the form of a paper. In many discussions, it is recommended that researchers keep their data separate from code by having, say, a subdirectory of the main project directory labelled `data` and another labelled `code` (or something similar).

My experience is that, while there is often data that is specific to a particular project, much of the data I work with has applications across multiple projects. Clearly data from the Center for Research in Security Prices (CRSP), whose

"US Stock Databases contain daily and monthly market and corporate action data for securities with primary listings on the NYSE, NYSE MKT, NASDAQ, and Arca exchanges and include CRSP broad market indexes", have applications across multiple projects.[1]

Some researchers handle general data by maintaining project-specific copies of the data. There is the download of part of the CRSP daily stock file that I made in 2015 for the paper on earnings management around IPOs. Then there is the download of the same data set that I made a few months later for the paper on the new lease accounting standard.

But other data sets might start life as project-specific data

1.2 Why a relational database?

A key point of this book regards the benefits of using a relational database as the primary data repository.[2] To discuss the benefits of a relational database, it is helpful to understand the common alternatives.

1.2.1 Alternative 1: Data files in statistical package of choice

One common approach is to primarily use one statistical programming package (e.g., Stata) and to keep data in the native format of that package.

1.3 Why PostgreSQL?

Much of this book involves the use of PostgreSQL as the backbone of a setup for empirical research in accounting, finance, economics, and fields with similar kinds of data.[3] You may wonder why I emphasize PostgreSQL and not, say, MySQL or SQLite or even something hosted in the cloud, such as Google's Redshift. I do so for a number of reasons:

- **To provide concrete examples.** Any expert in SQL or relational databases knows that, while there are standards such as SQL 92 and SQL-2003, no system closely follows these standards.[4] As such, to provide concrete working examples, I would often need to provide multiple versions for the different implementations.[5] Instead, by focusing on PostgreSQL,

[1] For many researchers in finance and accounting, it might be hard to imagine a study that did not use CRSP data.

[2] In ways I will discuss, it is effectively my exclusive data store.

[3] While PostgreSQL is the backbone, the research examples I will provide will generally use R or Python as the main coding language.

[4] My understanding is that PostgreSQL is one of the more standards-compliant implementations available.

[5] This is the approach taken by the *SQL Cookbook*.

I can provide working code without boring readers with the details of different implementations.

- **Other systems are similar.** My experience is that if you understand one SQL-based system, it is pretty easy to apply that knowledge to another system. This is particularly true when using packages such as `dplyr` (for R) or SQLAlchemy (for Python), as these packages attempt to abstract from platform-specific details. My hope is that if you use, say, MySQL, then much of what I say will carry over to that platform.[6]
- **PostgreSQL is the system I use.** A prosaic justification for featuring PostgreSQL is that it's what I use. Trying to explain something I don't use would be problematic, and setting up MySQL just to write this does not seem to be a good use of limited time.
- **PostgreSQL is very robust.** When I decieded to migrate my data into a relational database in 2010 and 2011, I initially tried SQLite. This is incredibly easy to set up and is available everywhere, but after the second time a 20GB data table was corrupted, I decided it wasn't quite what I was looking for. Since switching to PostgreSQL in 2011, I haven't experience data corruption or database crashes.[7]
- **PostgreSQL has some great features.** I also dabbled with MySQL alongside PostgreSQL for a while, but in the end was captivated by some features of PostgreSQL, some of which are compelling for researchers. These include the very rich set of data types supported by PostgreSQL, the availability of procedural languages for in-database programming, and the support for modern SQL features such as common-table expressions and window functions. I will discuss what these are and how they are useful for researchers in Chapter 3. While PostgreSQL has many compelling features for a wide range of situations, in the system I describe, PostgreSQL is largely functioning as a data repository. As such, many of the benefits could be obtained using another platform, such as MySQL.
- **PostgreSQL is good value for money.** All this for nothing. PostgreSQL is supported by an enthusiastic and active group of users and developers.
- **A PostgreSQL server is now available from WRDS.** Wharton Research Data Services (WRDS) is a major sourc of data for researchers in finance, accounting and other fields. WRDS recently began offering a server on `wrds-pgdata.wharton.upenn.edu` running on port `9737`. This is some kind of a vote of confidence in PostgreSQL and also creates additional reasons to use PostgreSQL as doing so creates multiple ways to access WRDS data from within PostgreSQL.

[6] Another possibility is that you might like what you see in PostgreSQL as make the switch.

[7] Sometimes I have crashed the machine on which the database has been running, but that has never been PostgreSQL's fault.

Chapter 2

Bad practices in research

2.1 Manual steps in analysis

A mantra in some areas of statistics and biomedical research is "reproducibility". In some contexts, reproducibility of results means that one researcher could run exeriments that are essentially identical to another researcher's (presumably with different subjects, etc.) and get similar results. But in the context of research computing, especically with archival data sets, the idea is that one could take the steps outlined in a paper, using the (sometimes public) data sets, and produce similar results to those in the paper. In the limit, if the data sets are publicly available and the processes for transforming those for the paper are described precisely enough, then it should be possible to obtain precisely the coefficient estimates, etc., provided in the original paper.

2.2 Manual modification of data

2.3 Bad (or no) documentation

2.4 Poor version control

2.5 Limited sharing of code and data

2.6 No data exploration

2.7 Casual approach to merging data sets

Chapter 3

The backbone: The relational database

The approach I take in this chapter is to start by illustrating the kinds of things that a relational database platform allows me to do. Initially, I will gloss over the messy details of setting up a database, getting data into it, etc. But rest assured that if you like what you see, it is fairly straightforward to set up the infrastructure I describe.[1]

In addition, I will often illustrate ideas using code that I only fully explain later in the book. For example, I find R packages such as `dplyr` to be particularly slick

3.1 Handling large-ish data sets

One nice feature of a relational database is that it makes it easy to access data from large-ish data sets.

Suppose I was interested in the stock returns of Amazon.[2]

The CRSP daily stock file (denoted as `crsp.dsf` on the WRDS system) contains about 18 GB of data in about 90 million rows. While someone dealing with the log files of a busy website or managing the geospatial data of Uber would chuckle at the idea of this being "big data", the reality is that 18GB is a pain to work with in a number of situations:

[1] Much of the code I use to maintain my platform, along with instructions for using it, is available on my GitHub page.

[2] This is admittedly not a particularly realistic *research* problem, but many of the benefits described here do accrue in real research applications.

- When the data aren't local. Downloading 18GB over the internet is time-consuming with even the fastest broadband connections.
- When you have limited RAM. For a typical laptop in 2017, loading in 18GB into memory will exhaust available RAM.
- When you're only interested in a small sliver of the data, such in our hypothetical "Amazon stock returns" case.

So how would I access these data? First, I would establish a connection to my database:

```
library(dplyr, warn.conflicts = FALSE)
pg <- src_postgres()
```

Here I am using R (and, in particular, the package `dplyr`). The first line loads the package.[3] The second line connects to my database.[4]

The next thing I need to do is to identify the `permno` (CRSP's firm identifier), which I can get by looking at the `crsp.stocknames` table.

```
stocknames <- tbl(pg, sql("SELECT * FROM crsp.stocknames"))

stocknames %>%
  filter(comnam %~*% 'AMAZON') %>%
  select(permno, comnam) %>%
  collect()
```

```
## # A tibble: 1 x 2
##   permno comnam
## *  <int> <chr>
## 1  84788 AMAZON COM INC
```

The first line of the code creates a variable `stocknames` that I can use to refer to the underlying data table `crsp.stocknames` in my database.[5]

Here I am using the `%>%` "pipe" operator, which is comes from the `magrittr` package and is re-exported by `dplyr`. The easiest way to understand this in the context of `dplyr` is that thing to the left of `%>%` is typically the object of the thing to the right of `%>%`, which is typically a verb. So we take stocknames, and `filter` to retain only observations where the company name (`comnam`) matches `AMAZON`.[6] Now that I know that the PERMNO for Amazon is `84788`, I can go to the daily stock file (`crsp.dsf`) to get stock returns.[7]

[3] You may have to `install.packages("dplyr")` if you do not have it installed already.

[4] I have omitted connection information here. As I will discuss later, I think it is good practice to supply that elsewhere, which is what I have done here.

[5] This syntax requires the use of a full SQL statement, `SELECT * FROM crsp.stocknames`, which isn't particularly elegant, but also isn't particularly problematic.

[6] `%~%` is PostgreSQL's regular expression matching operator. More will be said on all this later on. For now, just accept that it's doing some kind of string matching.

[7] An alternative approach would involve joining the query above with the one below. But I assume that it is necessary to visually inspect the query above to verify that we have the right

The result from that `filter` operation is another (smaller) data set that becomes the object for the next verb: `select`. Like its SQL counterpart (typically written `SELECT` in SQL circles), `select` chooses the columns of the data set that are to be retained. For this query, we want the `permno` and, to check we have the right company, the company's name (`comnam`).

```
dsf <- tbl(pg, sql("SELECT * FROM crsp.dsf"))

amzn_rets <-
  dsf %>%
  filter(permno == 84788L) %>%
  select(permno, date, ret, prc) %>%
  mutate(prc = abs(prc)) %>%
  collect()

amzn_rets
```

```
## # A tibble: 5,443 x 4
##    permno date           ret   prc
##  *  <int> <date>       <dbl> <dbl>
##  1  84788 1997-05-15 NA       23.5
##  2  84788 1997-05-16 -0.117   20.8
##  3  84788 1997-05-19 -0.0120  20.5
##  4  84788 1997-05-20 -0.0427  19.6
##  5  84788 1997-05-21 -0.127   17.1
##  6  84788 1997-05-22 -0.0219  16.8
##  7  84788 1997-05-23  0.0746  18
##  8  84788 1997-05-27  0.0556  19
##  9  84788 1997-05-28 -0.0329  18.4
## 10  84788 1997-05-29 -0.0170  18.1
## # ... with 5,433 more rows
```

Now, we set up a variable to refer to `crsp.dsf`, `filter` to get data just for Amazon, `select` the variables of interest, then `mutate` the stock price variable (`prc`) to correct negative values, before using `collect` to bring the data from the database into R.[8]

Note that this all took about a tenth of a second. So the fact that the data are stored in an 18GB table on a different computer was not at all problematic. PostgreSQL is able to grab the data of interest quickly and send (over the internet) just what I need. The resulting data is a mere 150.2 Kb.

Now that I have the data, suppose that I wanted to plot cumulative stock

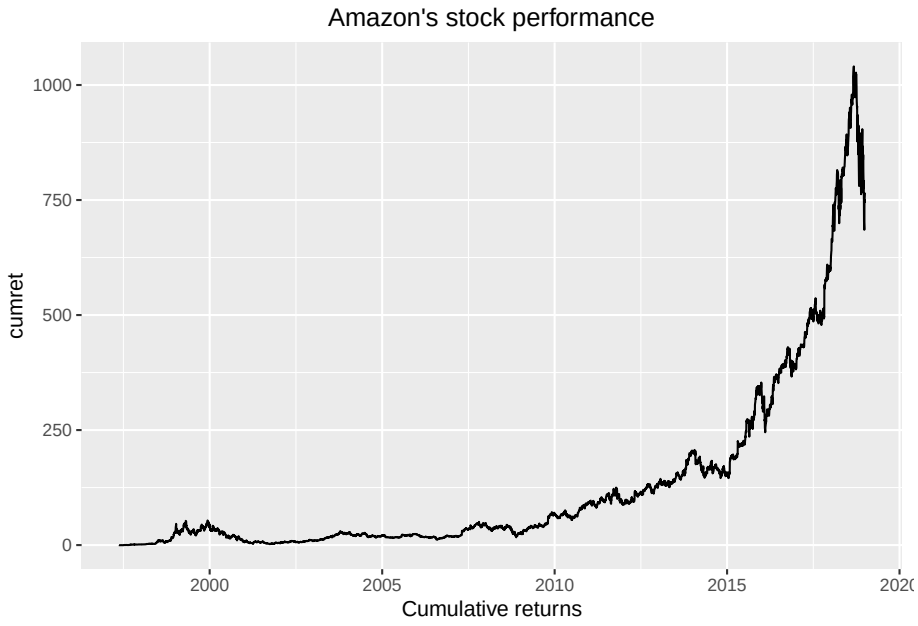
"Amazon" here.

[8]Stock prices should be positive, but André de Souza of NYU points out that "closing prices in the `crsp.msf` and `crsp.dsf` files are negative if the price is not the price from an actual trade, but the average of bid-ask spreads. I always set `prc = abs(prc)` before I do anything else with price."

performance. Many researchers would take the data above and dump to a file that could be opened in Excel, open that file in Excel, use the mouse to select the data of interest, and then make a plot. That plot could be copied and pasted into, say, a Word document. But here we have taken several manual steps of the kind I argued against in Chapter 2.

How else might we do this? Well, we could use the excellent `ggplot2` package to make a plot following on from our steps so far. Here is the code to do so:

```
library(ggplot2)
amzn_rets %>%
  mutate(cumret = cumprod(1+coalesce(ret, 0)) - 1) %>%
  ggplot(aes(x = date, y = cumret)) +
  geom_line() +
  xlab("Cumulative returns") +
  ggtitle("Amazon's stock performance") +
  theme(plot.title = element_text(hjust = 0.5))
```



Chapter 4

Getting data into PostgreSQL

In this chapter I sketch some aspects of getting data into PostgreSQL from a variety of sources. Here are examples I consider

- Data from Wharton Research Data Services (WRDS)
- Conference call transcripts from StreetEvents
- Hand-collected data from Google Sheets
- Excel worksheets from third parties

4.1 Data from WRDS

A major source of data for finance and accounting research is WRDS. One way to access WRDS data is via WRDS's new PostgreSQL database. You can use your WRDS ID to connect to this database. Here is how to connect using R:[1]

```
library(dplyr, warn.conflicts = FALSE)
library(DBI)
pg_wrds <- dbConnect(RPostgres::Postgres(),
                     host = "wrds-pgdata.wharton.upenn.edu", port = 9737,
                     user = "iangow", dbname = "wrds")
```

Having connected to the database, one can access the WRDS data you have access to in much the same way as you might via SAS. For example, the CRSP daily stock file is in the table `dsf` in the `crsp` schema. Here we look at a few rows of this.

[1] You will also need the `RPostgreSQL` package installed; you also need to supply your WRDS password, which could be done by adding `password = "XXXXX"` to the arguments to `src_postgres` below.

```
dsf <- tbl(pg_wrds, sql("SELECT * FROM crsp.dsf"))
dsf %>%
  mutate(prc = abs(prc)) %>%
  select(permno, date, prc, ret)
```

```
## # Source:   lazy query [?? x 4]
## # Database: postgres [iangow@wrds-pgdata.wharton.upenn.edu:9737/wrds]
##    permno date         prc     ret
##     <dbl> <date>     <dbl>   <dbl>
##  1  56611 1984-04-04  23    0
##  2  56611 1984-04-05  22.5 -0.0217
##  3  56611 1984-04-06  22.5  0
##  4  56611 1984-04-09  22.5  0
##  5  56611 1984-04-10  23.5  0.0444
##  6  56611 1984-04-11  23.5  0
##  7  56611 1984-04-12  23.5  0
##  8  56611 1984-04-13  23.5  0
##  9  56611 1984-04-16  23.5  0
## 10  56611 1984-04-17  23.5  0
## # ... with more rows
```

While that's one option for accessing WRDS data, it has its limitations. First, a common scenario is that one wants to merge data from WRDS with a large-ish data set of one's own. Given that the WRDS database does not include your data, you'd need to pull the data from WRDS on one's own machine and then merge. One can pull data into a local dataframe using the `dplyr` verb `collect()`. But `dsf %>% collect()` is not something you want to be doing on a regular basis.

For WRDS data sets that you use in your research, I think it makes sense to have a copy of the data in your own database. While doing this uses up disk space and requires some time to download te data, the process for doing this is relatively painless.[2]

I use Python scripts to maintain my subset of WRDS, which includes data from Compustat, CRSP, and Capital IQ, among other data sets. These scripts are available at https://github.com/iangow/wrds_pg.

[2] An advantage of this approach is that the WRDS PostgreSQL database can be a little rough in places, with timestamps stored as text values and integers as floating point values.

Chapter 5

Identifiers

Here *identifiers* refer to things such as PERMNOs, tickers, GVKEYs, CUSIPs, CIKs, and the like. The idea behind these firm identifiers is that they uniquely identify a firm for a particular purpose.[1] Of course, identifiers apply not only to firms, but also people.[2] But most of this chapter will focus on firm identifiers, in part because of the importance of firsm as units of observation in finance and accounting research, but also becasue identifying firms is much harder than identifying people. While not specfic to the platform I describe in this book, the issue of *identifiers* is one that seems less perplexing and better-handled when a relational database provides the backbone of your data store as it does here.

5.1 Firm identifiers: A quiz

My sense is that researchers' understanding of firm identifiers is often sketchy. Here is a quick quiz to test your knowledge of firm identifiers.[3] Using a standard mapping such as the PERMNO-GVKEY mapping provided by CRSP:

- General Motors Corporation declated bankruptcy in June 2009? Does the "successor firm" General Motors Company have the same GVKEY? The same PERMNO?
- Can a CUSIP map to more than one PERMNO? To more than one GVKEY?
- Can a PERMNO map to more than one CUSIP?
- Can a GVKEY map to more than one PERMCO?
- Can a PERMCO map to different CIKs?

[1] Of course, PERMNOs, CUSIPs, and tickers (at best) identify *securities*, not firms. More on this below.

[2] In the United States, a Social Security Number (SSN) is a pretty robust identifier of people, though as researchers, we generally don't have access to SSNs.

[3] Obviously, I am assuming that you recognize the various identifiers. If not, read on.

- If I have two data sets, X and Y and CUSIP is a "firm" identifier on each, can you simply merge using CUSIPs?
- When would a "firm" change CUSIPs?
- When would a "firm" change CIKs?
- If the firm identifier on IBES is `ticker`, should I merge with CRSP using `ticker` from `crsp.stocknames`?

5.2 CRSP's link tables

From WRDS:

> The WRDS-created linking dataset (ccmxpf_linktable) has been deprecated. It will continue to be created for a transition period of 1 year. SAS programmers should use the Link History dataset (ccmxpf_lnkhist) from CRSP.

And from here:

> LINKPRIM clarifies the link's relationship to Compustat's marked primary security within the related range. "P" indicates a primary link marker, as identified by Compustat in monthly security data. "C" indicates a primary link marker, as identified by CRSP to resolve ranges of overlapping or missing primary markers from Compustat in order to produce one primary security throughout the company history. "J" indicates a joiner secondary issue of a company, identified by Compustat in monthly security data.

So let's check it out. Given any GVKEY and a date, is there only one PERMNO that is matched with `linkprim IN ('P', 'C')`?

```
ccmxpf_lnkhist <-
    tbl(pg, sql("SELECT * FROM crsp.ccmxpf_lnkhist"))

gvkey_permno <-
    ccmxpf_lnkhist %>%
    filter(linkprim %in% c("C", "P"),
           !is.na(lpermno))

# Look for overlapping date ranges
gvkey_permno %>%
    group_by(gvkey) %>%
    arrange(gvkey, linkdt) %>%
    mutate(lead_linkdt = lead(linkdt),
           lag_linkenddt = lag(linkenddt)) %>%
    filter(linkenddt >= lead_linkdt | lag_linkenddt >= linkdt)
```

```
## # Source:     lazy query [?? x 10]
## # Database:   postgres 11.3.0 [igow@iangow.me:5432/crsp]
```

```
## # Groups:     gvkey
## # Ordered by: gvkey, linkdt
```

So there is just *one* case of overlapping dates, and it has `linkprim` equal to `P` in both rows. But the answer is effectively "no" because you get the same PERMNO (`77571`) in either row.

Note that the question here begins with a `GVKEY` and asks "given this GVKEY, which PERMNO provides the correct security-related information (return, stock price) for a given date?" The answer is given by `lpermno` above. Note that there is no mention of `datadate` (from Compustat) here. But it may be that I am interested in security information on `datadate` and thus that would drive the selection of `lpermno`.

Note that the vast majority of GVKEYs map to just one PERMNO:

```
gvkey_permno %>%
    select(gvkey, lpermno) %>%
    distinct() %>%
    group_by(gvkey) %>%
    mutate(num_permnos = n()) %>%
    ungroup() %>%
    count(num_permnos) %>%
    arrange(num_permnos)
```

```
## # Source:     lazy query [?? x 2]
## # Database:   postgres 11.3.0 [igow@iangow.me:5432/crsp]
## # Ordered by: num_permnos
##   num_permnos     n
##         <dbl> <dbl>
## 1           1 27795
## 2           2  1698
## 3           3   201
## 4           4     8
## 5           5     5
```

I think I've looked into the case with 5 PERMNOs before. If I recall correctly, it's a total mess with tracking stock, spin-offs, etc. But one observation doesn't matter much.

```
stocknames <- tbl(pg, sql("SELECT * FROM crsp.stocknames"))
gvkey_permno %>%
    select(gvkey, lpermno) %>%
    distinct() %>%
    group_by(gvkey) %>%
    mutate(num_permnos = n()) %>%
    filter(num_permnos==5L) %>%
    inner_join(ccmxpf_lnkhist) %>%
    rename(permno=lpermno) %>%
```

```
    inner_join(stocknames) %>%
    arrange(linkdt) %>%
    select(gvkey, permno, linkdt, linkenddt, ncusip, comnam)
```

Chapter 6

Application: Firm performance over time

Chapter 7

Application: Event returns

Chapter 8

Application: Processing textual data

Chapter 9

Application: Hand-collection of data

9.1 Google Sheets

9.2 Text annotation