The data model

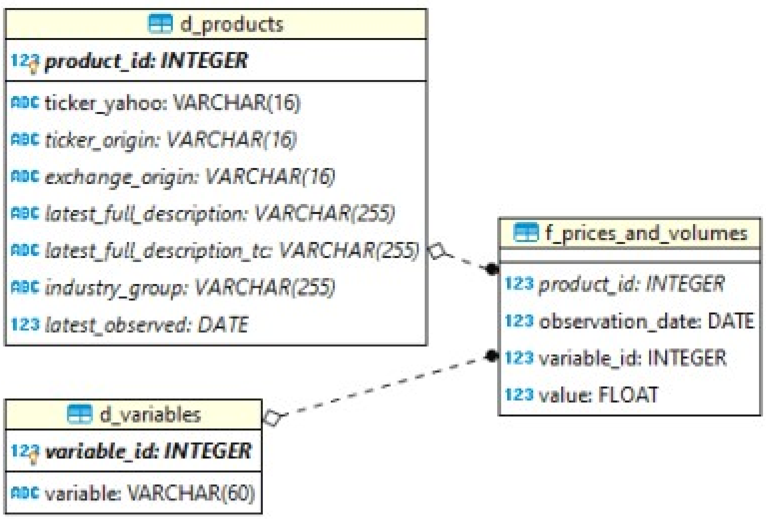
For my purposes I am happy with daily data. The data I want to store has an obvious grain of date-security ie one row for each day for each security that I have data for. This implies just two dimensions – date and security (or product, as I call it below, following ASIC's terminology for their published data on short positions). The facts that I'm interested in are the open, high, low, close and adjusted price for each day; the volume of transactions; the short position; the total float; and the short position as a proportion of the total float. The price and volume data can come from Yahoo Finance, and the short positions data can be downloaded from the Australian Securities and Investment Commission, who collect self-reports from short sellers.

My initial plan was just two tables:

- one for the product dimension with unchanging (or slowly changing) information like its ASX ticker, the ticker on Yahoo Finance, its latest name with various alternative approaches to punctuation
- one for the facts, with separate columns for the nine facts (open price, volume, etc) listed in the previous paragraph.

I think this is indeed probably the correct structure for a Kimball-style dimensionally modelled analytical datamart. However, it proved unpleasant to write the extract-transform-load for my two data sources into that wide fact table. It would have involved a lot of `UPDATE` operations to add data from one source to columns for rows that are partly populated by the other). SQLite in particular does not really support `UPDATE` in combination with a `JOIN` and getting around this would have been awkward. So to simplify things I normalised the data one step further and made my fact table more "long and thin". This meant adding a variable dimension so that when extra data comes in from another source I am just adding new rows to the data for a subset of variables rather than filling in empty columns for existing rows.

Here's how the data model looks:



I used the excellent (and free) DBeaver universal database tool to make that diagram, and to develop any non-trivial SQL code used in this blog.

And this is the SQL that creates it. I've used `GO` (in the style of Microsoft T-SQL) to separate each command, not because SQLite understands it (it doesn't) but because I have R functions that do, which I'll be using in a minute.

Post continues below SQL code

```
-- Drop existing version of database (careful!):
DROP TABLE IF EXISTS f_prices_and_volumes;
GO
DROP TABLE IF EXISTS d_variables;
GO
DROP TABLE IF EXISTS d_products;
GO

-- Define the various tables:
CREATE TABLE d_products (
  product_id INTEGER PRIMARY KEY,
  ticker_yahoo VARCHAR(16) NOT NULL UNIQUE,
  ticker_origin VARCHAR(16),
  exchange_origin VARCHAR(16),
  latest_full_description VARCHAR(255),
  latest_full_description_tc VARCHAR(255),
  industry_group VARCHAR(255),
  latest_observed DATE

)
GO

CREATE TABLE d_variables (
  variable_id INTEGER PRIMARY KEY,
  variable VARCHAR(60) NOT NULL UNIQUE
)
GO

CREATE TABLE f_prices_and_volumes (
  product_id INTEGER,
  observation_date DATE NOT NULL,
  variable_id INTEGER NOT NULL,
  value FLOAT NOT NULL,

  FOREIGN KEY (product_id) REFERENCES d_products
(product_id),
  FOREIGN KEY (variable_id) REFERENCES d_variables
(variable_id)
)
GO

CREATE UNIQUE INDEX one_obs_per_var
    ON f_prices_and_volumes(variable_id, product_id,
observation_date);
GO

-- Populate the variables dimension:
```

```
INSERT INTO d_variables (variable) VALUES
        ('open'),
        ('high'),
        ('low'),
        ('close'),
        ('volume'),
        ('adjusted'),
        ('short_positions'),
        ('total_product_in_issue'),
        ('short_positions_prop')
```

Populating with data

I used R for accessing the origin data from the web and sending SQL commands to set up the database. Here's the first chunk of code that creates the empty database, runs the SQL above to set up tables, and makes an initial dump of ASX listed companies into the `d_products` table. I adapted some of this and subsequent code from this blog post by Michael Plazzer. I'm not sure how definitive is that list of ASX listed companies referred to in the below code.

Post continues below R code

```
library(tidyverse)
library(glue)
library(lubridate)
library(frs) # for download_if_fresh and execute_sql.
Available from GitHub.
library(janitor)
library(ggrepel)
library(kableExtra)
library(quantmod)
library(RSQLite)
library(DBI)
library(tools)

#=============database setup============================


#-----------Define database------------------
if(!"stocks.sqlite" %in% list.files()){
  # if this is the very first run we need to create the empty
database and its tables
  con <- dbConnect(RSQLite::SQLite(), "stocks.sqlite")

  # run the SQL script that defines the table - needs to be
saved in seaprate file:
  execute_sql(con, "0199-stocks-db-setup.sql", log_table =
NULL)

} else {
  con <- dbConnect(RSQLite::SQLite(), "stocks.sqlite")
}
```

```
#------------set up product dimension----------

asx_cos <- read.csv("http://www.asx.com.au/asx/research/
ASXListedCompanies.csv",skip=1) %>%
  mutate(ticker_yahoo = paste0(ASX.code, ".AX"),
         exchange_origin = "ASX",
         latest_observed = as.Date(NA),
         latest_full_description_tc =
tools::toTitleCase(tolower(Company.name))) %>%
  select(
    ticker_yahoo,
    ticker_origin = ASX.code,
    exchange_origin,
    latest_full_description = Company.name,
    latest_full_description_tc,
    industry_group = GICS.industry.group,
    latest_observed
  )

RSQLite::dbWriteTable(con, "d_products", asx_cos, row.names
=FALSE, append = TRUE)
```

Loading the short positions data

The short positions data from the ASIC website includes many products that aren't in the list of listed companies on the ASX site. In general, I want to be able to update my list of products/securities. Getting data from Yahoo Finance, where I specify a security ticker code and then get the data, won't let me do this (unless I tried codes at random). Because of all this, in my initial bulk upload I do the ASIC data first, hoping (without really checking how it happens, which of course I would for a more formal use) that this will surface new (or old) securities that aren't in the spreadsheet I downloaded from the ASX.

The code below is in two chunks. It downloads all the CSVs of short positions data from ASIC, taking care not to re-download data it already has. Each CSV represents one day of three facts on each product. Then (somewhat more complex), it reads all the CSVs one at a time (if it hasn't already processed this particular CSV); identifies missing products/securities which it then adds to the `d_products` table; then populates the fact table with the facts for all the products it's found in this particular CSV, having matched them to the `product_id` field in the `d_products` table.

Other than the three-card shuffle with adding new products to `d_products` as it goes, and some annoying complications with different formats and encoding of the CSV files on the ASIC page (see comments in the code), this is fairly straightforward data-wrangling stuff for R.

This took three or four hours each for the two bits of functionality (bulk download and bulk import) to run.

```
#===============Get the short positions data=============

#----------Download-------------
# From:
# https://asic.gov.au/regulatory-resources/markets/short-selling/short-position-
reports-table/
```

```
all_dates <- seq(from = as.Date("2010-06-16"), to =
Sys.Date(), by = 1)
dir.create("asic-shorts", showWarnings = FALSE)

i = 1
for(i in i:length(all_dates)){
  the_date <- all_dates[i]

  # Don't bother trying to download on weekends:
  if(wday(the_date, label = TRUE) %in% c("Sat", "Sun")){
    next()
    }

  m <- str_pad(month(the_date), width = 2, side = "left", pad
= "0")
  y <- year(the_date)
  ch <- format(the_date, "%Y%m%d")
  fn <- glue("RR{ch}-001-SSDailyAggShortPos.csv")

  url <- glue("https://asic.gov.au/Reports/Daily/{y}/{m}/{fn}")

  # Only exists for trading days. Rather than bother to work
out exactly the trading days,
  # we will just skip over any 404 error
  try(download_if_fresh(url, glue("asic-shorts/{fn}")))
}


#----------------------Import---------------------------

all_products <- dbGetQuery(con, "SELECT product_id,
ticker_yahoo
                                 FROM d_products
                                 WHERE exchange_origin =
'ASX'") %>%
  as_tibble()

already_done_dates <- dbGetQuery(con, "SELECT DISTINCT
observation_date AS od
                                      FROM
f_prices_and_volumes AS a
                                      INNER JOIN d_variables
AS b
                                        On a.variable_id =
b.variable_id
                                      WHERE b.variable =
'short_positions'
                                      ORDER BY
observation_date")$od

d_variables <- dbGetQuery(con, "select variable_id, variable
```

```
from d_variables")

all_csvs <- list.files("asic-shorts", pattern =
"DailyAggShortPos", full.names = TRUE)

# we are going to do this backwards so product names are the
latest ones
i =length(all_csvs)
for(i in i:1){

  the_csv <- all_csvs[i]
  the_date <- as.Date(str_extract(the_csv, "[0-9]+"), format=
"%Y%m%d")

  # if we've already got short positions observations in the
data for this date,
  # then break out of the loop and go to the next iteration
  if(as.character(the_date) %in% already_done_dates){next()}

  # The first 1400 files are actually tab-delimited and
UTF-16, the
  # rest are genuine comma separated files and more standard
encoding.
  # Couldn't get read_csv to work with the various encoding
here.
  if(i <= 1400){
    d1 <- read.csv(the_csv, fileEncoding = "UTF-16", sep =
"\t")
  } else {
    d1 <- read.csv(the_csv)
    # sometimes this still doesn't work and we go back to the
other method:
    if(nrow(d1) < 10){
      d1 <- read.csv(the_csv, fileEncoding = "UTF-16", sep =
"\t")
    }
  }

  d2 <- d1 %>%
    clean_names() %>%
    as_tibble() %>%
    mutate(observation_date = the_date,
           ticker_yahoo = paste0(str_trim(product_code),
".AX")) %>%
    left_join(all_products, by = "ticker_yahoo") %>%
    select(
      product_id,
      observation_date,
      short_positions = reported_short_positions,
      total_product_in_issue,
      short_positions_prop = x_of_total_product_in_issue_
reported_as_short_positions,
```

```
      ticker_yahoo,
      product,
      product_code
    ) %>%
    # convert from percentage to proportion:
    mutate(short_positions_prop = short_positions_prop / 100)

  non_match <- sum(is.na(d2$product_id))
  if(non_match > 0){
    message(glue("Found {non_match} products in the short
data not yet in the database"))
    print(select(filter(d2, is.na(product_id)), ticker_yahoo,
product, observation_date))

    new_products <- d2 %>%
      filter(is.na(product_id)) %>%
      mutate(exchange_origin = 'ASX',
             latest_full_description_tc =
tools::toTitleCase(tolower(product)),
             industry_group = NA,
             latest_observed = NA,
             ticker_origin = str_trim(product_code)) %>%
      select(ticker_yahoo,
             ticker_origin,
             exchange_origin,
             latest_full_description = product,
             latest_full_description_tc,
             industry_group,
             latest_observed)

    RSQLite::dbWriteTable(con, "d_products", new_products,
row.names =FALSE, append = TRUE)

    all_products <- dbGetQuery(con, "SELECT product_id,
ticker_yahoo
                                 FROM d_products
                                 WHERE exchange_origin =
'ASX'") %>%
      as_tibble()
  }

  upload_data <- d2 %>%
    select(observation_date:ticker_yahoo) %>%
    left_join(all_products, by = "ticker_yahoo") %>%
    select(-ticker_yahoo) %>%
    gather(variable, value, -product_id, -observation_date)
%>%
    left_join(d_variables, by = "variable") %>%
    select(product_id,
         observation_date,
         variable_id,
```

```
          value) %>%
    # a small number of occasions the short_positions_prop is
NA or Inf because
    # the totla product in issue is 0 even though there are
some short positions.
    # we will just filter these out
    filter(!is.na(value)) %>%
    mutate(observation_date = as.character(observation_date))

  dbWriteTable(con, "f_prices_and_volumes", upload_data,
append = TRUE)

  # progress counter so we know it isn't just stuck
  if(i %% 100 == 0){cat(i)}
}
```

Loading the price and volumes data

Next step is to get some data from Yahoo Finance on price and volumes. Overall, this is more straightforward. The `quantmod` R package describes the functionality I'm about to use as "essentially a simple wrapper to the underlying Yahoo! finance site's historical data download".

I've tried in the code below to make this updateable, so in future I can run the same code without downloading all the historical data again. But I haven't fully tested this; it's more a working prototype than production-ready code (and I wouldn't use SQLite for production in this case). But here's code that works, at least for now. It gets all the Australian security ticker names in the format used by Yahoo (finishing with `.AX` for the ASX) and their matching `product_id` values for my database; finds the latest data data is available in the database; downloads anything additional to that; normalises it into long format and uploads it to the fact table in the database.

This took a couple of hours to run (I didn't time it precisely).

```
#================stock price and volume
information===========

all_stocks <- dbGetQuery(con, "SELECT product_id,
ticker_yahoo
                               FROM d_products
                               ORDER BY product_id") %>%
as_tibble()

i=1
for(i in i:nrow(all_stocks)){
  # display a counter ever 20 iterations so we know we're
making progress:
  if(i %% 20 == 0){
    cat(i, " ")
  }
  ax_ticker <- all_stocks[i, ]$ticker_yahoo

  latest <- as.Date(dbGetQuery(con, glue("select
```

```
max(observation_date) as x from f_prices_and_volumes
                        where product_id = {all_stocks[i,
]$product_id}"))$x)
  if(is.na(latest)){
    start_date <- as.Date("1980-01-01")
  } else {
    start_date <- latest + 1
  }

  if(start_date <= Sys.Date()){
    tryCatch({
      df_get <- data.frame(getSymbols(
        ax_ticker,
        src = 'yahoo',
        from = start_date,
        to = Sys.Date(),
        auto.assign = FALSE))

      if(nrow(df_get) > 0){
        df_get$observation_date <- row.names(df_get)

        row.names(df_get) <- NULL
        names(df_get) <- c("open", "high", "low", "close",
"volume", "adjusted", "observation_date")
        upload_data <- df_get %>%
          as_tibble() %>%
          mutate(product_id = all_stocks[i, ]$product_id) %>%
          gather(variable, value, -observation_date,
-product_id) %>%
          inner_join(d_variables, by = "variable") %>%
          select(product_id,
                 observation_date,
                 variable_id,
                 value) %>%
          filter(observation_date >= start_date) %>%
          mutate(observation_date =
as.character(observation_date))

        dbWriteTable(con, "f_prices_and_volumes",
upload_data, append = TRUE)
      }

    },
    error=function(e){})
  }
}
```

Updating the product dimension with some summary data

The final steps in the extract-transform-load process are some convenience additions to the database. First, I update the `d_products` table which has a `latest_observed` column in it with the most recent observation for each product:

```
#===============Update the observation dates in the dimension
table============
# This is much more complicated with SQLite than in SQL
Server because of the
# apparent inability of SQLite to elegantly update a table
via a join with
# another table. There may be a better way than the below but
I couldn't find it:

sql1 <-
  "CREATE TABLE tmp AS
  SELECT product_id, max(observation_date) as the_date
  FROM f_prices_and_volumes
  WHERE observation_date IS NOT NULL
  GROUP BY product_id;"

sql2 <-
  "UPDATE d_products
  SET latest_observed = (SELECT the_date FROM tmp WHERE
d_products.product_id = tmp.product_id)
  WHERE EXISTS (SELECT * FROM tmp WHERE d_products.product_id
= tmp.product_id);"

sql3 <- "DROP TABLE tmp;"


dbSendQuery(con, sql1)
dbSendQuery(con, sql2)
dbSendQuery(con, sql3)
```

Finally, I want to create a wider version of the data, closer to my original idea of a fact table with one row per product-date combination, and nine fact columns (for open price, volume, short position, etc). In another database I would use an indexed or materialized view for this sort of thing, but SQLite doesn't support that. I tried making a view (basically a stored query) but its performance was too slow. So I created a whole new table that will need to be created from scratch after each update of the data. This isn't as disastrous as it sounds - an indexed view does something similar in terms of disk space, and it only takes a minute or so to run this. And it is a convenient table to have.

So here's the final step in this whole data upload and update process, creating that wide table from scratch. Note the clunky (to R or Python users who are used to things like `spread()` or `pivot_wider()`) way that SQL pivots a table wide, with that use of the `SUM(CASE WHEN ...)` pattern. It looks horrible, but it works (so long as you know in advance all the column names you are trying to make in the wider version):

```
#============Create a denormalised (wide) version of the main
fact table=======
# In another database system this would be a materialized
view or indexed view or similar,
# but we don't have that in SQLite so it is a straight out
table. Note that this roughly
# doubles the size of the database on disk; and duplication
```

```
means we need to remember
# to re-create this table whenevedr the original fact table
updates.

sql1 <- "DROP TABLE IF EXISTS f_prices_and_volumes_w"

sql2 <- "
CREATE TABLE f_prices_and_volumes_w AS
SELECT
        observation_date,
        c.ticker_yahoo,
        c.ticker_origin,
        c.latest_full_description,
        SUM(CASE WHEN variable = 'open' THEN value END) AS
open,
        SUM(CASE WHEN variable = 'high' THEN value END) AS
high,
        SUM(CASE WHEN variable = 'low' THEN value END) AS
low,
        SUM(CASE WHEN variable = 'close' THEN value END) AS
close,
        SUM(CASE WHEN variable = 'volume' THEN value END) AS
volume,
        SUM(CASE WHEN variable = 'adjusted' THEN value END)
AS adjusted,
        SUM(CASE WHEN variable = 'short_positions' THEN value
END) AS short_positions,
        SUM(CASE WHEN variable = 'total_product_in_issue'
THEN value END) AS total_product_in_issue,
        SUM(CASE WHEN variable = 'short_positions_prop' THEN
value END) AS short_positions_prop
FROM f_prices_and_volumes AS a
INNER JOIN d_variables AS b
        ON a.variable_id = b.variable_id
INNER JOIN d_products AS c
        ON a.product_id = c.product_id
GROUP BY observation_date, ticker_yahoo, ticker_origin,
latest_full_description"

dbSendStatement(con, sql1)
dbSendStatement(con, sql2) # takes a minute or so
```

Exploratory analysis

Phew, now for the fun bit. But I'm going to leave substantive analysis of this for another post, as this is already long enough! I'll just do two things here.

First, let's look at a summary of how many data points we've got in the database

variable	n	number_products	number_dates
open	880775	1873	8491
high	880775	1873	8491

variable	n	number_products	number_dates
low	880775	1873	8491
close	880775	1873	8491
volume	880775	1873	8491
adjusted	880775	1873	8491
short_positions	1318329	3048	2694
total_product_in_issue	1318326	3047	2694
short_positions_prop	1318204	3046	2694

That all looks as expected. In total we have about 9 million observations. There are many securities with short positions reported to ASIC that I couldn't find prices and volumes for in Yahoo Finance, which is interesting and worth looking into, but not astonishing.

That table was created with this SQL (and a bit of R sugar around using `knitr` and `kableExtra`, not shown):

```
SELECT
        variable,
        COUNT(1) AS n,
        COUNT(DISTINCT(product_id)) AS number_products,
        COUNT(DISTINCT(observation_date)) AS number_dates
FROM f_prices_and_volumes AS a
INNER JOIN d_variables AS b
        ON a.variable_id = b.variable_id
GROUP BY variable
ORDER BY b.variable_id
```

Finally, some real analysis. What can we do with this database? Here's an example of the sort of thing that's possible with this asset that wasn't earlier. This is an answer to my hypothetical question I started with - what are the most traded and fastest growing (in price) securities on the ASX?

That chart was created with this code, which has three substantive bits:

- an SQL query (and R code to send it to the databse) that grabs the data we need, averaged by year for each product, from the wide table defined above
- a little function purely to change the upper/lower case status of product names for the chart
- `ggplot2` code to draw the chart.

```
#---------Recent growth-----------

sql <- "
WITH annual_vals AS
        (SELECT
            CAST(STRFTIME('%Y', observation_date) AS INT) AS
year,
                AVG(adjusted) AS avg_adjusted,
                sum(volume * adjusted) AS vol_val,
                latest_full_description,
                ticker_origin
        FROM f_prices_and_volumes_w
```

```
        GROUP BY latest_full_description, ticker_yahoo,
STRFTIME('%Y', observation_date))
SELECT
        SUM(CASE WHEN year = 2020 THEN avg_adjusted END) AS
adj_2020,
        SUM(CASE WHEN year = 2021 THEN avg_adjusted END) AS
adj_2021,
        SUM(CASE WHEN year = 2021 THEN vol_val END) AS
vol_val_2021,
        ticker_origin,
        latest_full_description
FROM annual_vals
WHERE year >= 2020
GROUP BY ticker_origin, latest_full_description
HAVING SUM(CASE WHEN year = 2020 THEN avg_adjusted END) > 0"

recent_growth <- dbGetQuery(con, sql) %>% as_tibble()

d <- recent_growth %>%
  mutate(gr = adj_2021 /  adj_2020 - 1) %>%
  filter(vol_val_2021 > 1e6) %>%
  arrange(adj_2021)

#' Convert upper case security names to title case
#'
#' @param x a character vector
#' @param rm_ltd whether or not to strip "Limited" and "Ltd"
from titles as unwanted clutter
#' @details A convenience function for labelling securities
on a chart, which
#'   converts to title case, keeps ETF (Exchange Trade Fund)
in capitals, and
#'   can remove 'Limited' altogether
better_case <- function(x, rm_ltd = TRUE){
  x <- toTitleCase(tolower(x))
  x <- gsub("Etf ", "ETF ", x)
  if(rm_ltd){
    x <- gsub("Limited", "", x)
    x <- gsub("Ltd", "", x)
  }
  x <- str_trim(x)
  return(x)
}

set.seed(123)
d %>%
  ggplot(aes(x = vol_val_2021 / 1e6, y = gr, colour =
ticker_origin)) +
  geom_point() +
  geom_text_repel(data = filter(d, gr > 2 | vol_val_2021 >
100e6),
                  aes(label = better_case(latest_full_
```

```
description)),
                  alpha = 0.9, size = 2.6) +
  scale_x_log10(label = dollar_format(suffix = "m")) +
  scale_y_continuous(label = percent) +
  theme(legend.position = "none") +
  labs(x = "Value of transactions in 2021, to 14 February",
       y = "Growth in price\nfrom average in 2020 to average
in 2021",
       title = "High volume and growth securities in the ASX,
2021",
       subtitle = "Labelled securities are those with volume
of trades > $100m or growth >200%",
       caption = "Source: Yahoo Finance via
http://freerangestats.info. This is not financial advice.")
```

La voila. Coming soon in a future blog post - exploring short positions of securities on the ASX.