Querying the MFE Sensor Database

Kaija Gahm

Last compiled on 2021-04-22

Introduction

Motivation

The MFE sensor database is much larger than the regular MFE database, since it contains high-frequency data. (As I'm writing this, the file is 2.81 GB, vs. 125.7 MB for the regular database).

As a result, querying data from the sensor database can be complicated. The simplest way is to load an entire table into R's memory and then perform any operations on the resulting object; this is intuitive if you're used to working with relatively small datasets in R. But because the sensor database tables are so large, loading and working with a full table can take a prohibitively long time. Sometimes, you might not be able to load the full table at all because it's too large to fit in R's memory.

The `sensordbTable` function

The `sensordbTable()` wrapper function can also handle slightly more complicated queries. It can be found on GitHub, and it takes the following arguments:

- `table` the data table to be returned
- `dbname` name of the database file
- `lakeID` vector of lakeID's that you want
- `depthClass` vector of depthClasses that you want
- `minDepth_m` numeric value of minimum sensor depth
- `maxDepth_m` numeric value of maximum sensor depth
- `minDate` minimum sample date in standard unambiguous format
- `maxDate` maximum sample date in standard unambiguous format
- `dateFormat` date format, if not in standard unambiguous format

These arguments can help you send more specific queries to the database to pull out just the data you need from a given sensor database table, instead of grabbing the whole table from the start.

However, these arguments don't cover all specific queries that you might want to write, so depending on what question you're answering, you will still have to do some data subsetting after pulling the initial dataset into R.

Working with the database through the tidyverse

I (Kaija) am a big fan of the tidyverse suite of R packages. But I'm not here to evangelize! If you don't like using the tidyverse, there's no pressure to use this method. I just wanted to demonstrate it in case it's useful to someone.

There's a package in the tidyverse called `dbplyr`, which provides some backend functions for working with databases through `dplyr`, one of the most common data wrangling packages in the tidyverse. You don't actually have to load `dbplyr` itself–it will just work in the background when you use what look like `dplyr` functions. In a nutshell, `dbplyr` allows you to perform data subsetting operations on remote database tables

before pulling the data into R's memory, using syntax that will be familiar if you're used to `dplyr`. Behind the scenes, it translates that code into SQL queries, which it sends to the database.

You can read more about the technicalities of `dbplyr` here, but I'm just going to demonstrate a workflow that works for me. Hopefully this will allow you to customize the functions to your own queries.

Step by step

1. Install packages

You'll need to install `RSQLite` and `dplyr`. Or, as I usually do, `RSQLite` and the entire `tidyverse`. I also like to install `here` to manage my file paths.

```
library(RSQLite)  # for the database connection
library(tidyverse)  # for dplyr/dbplyr
library(here)  # for file paths
```

2. Connect to the database

You probably aren't used to manually connecting to the database, since the `dbTable` and `sensordbTable` functions do this behind the scenes. But luckily, it's just one line of code, and you can copy and paste it until you memorize it.

```
con <- dbConnect(SQLite(), # this is the "drv", or "driver" argument. I...
                 # don't fully understand what it does.
                 here("MFEsensordb_20200526.db")) # this is a path to my
# database file, MFEsensordb_20200526.db. This path is written relative to
# the R project root directory, aka the folder where my .Rproj folder is
# stored--that's the magic of the `here` package. So in this case, my sensor
# database file is stored in the main project directory. You might have to
# customize this file path to point to your own file, depending on
# where it's stored.
```

For more info on the `here` package and relative file paths, see https://here.r-lib.org/.

For more information on RStudio "projects", the .Rproj file, and a project-oriented workflow, see https://www.tidyverse.org/blog/2017/12/workflow-vs-script/.

Okay, now we've created a *database connection* object called `con`. We can double-check that we correctly connected to the database by getting a list of all the database tables.

```
# (I'm printing just the first 10 for the sake of brevity)
dbListTables(con)[1:10]  # great, that looks like what we'd expect.
```

```
##  [1] "CHANGES_CODE_KEY"     "DO_CORR"              "DO_RAW"
##  [4] "HOBO_METSTATION_CORR" "HOBO_METSTATION_RAW"  "HOBO_PRESS_CORR"
##  [7] "HOBO_PRESS_RAW"       "HOBO_TCHAIN_CORR"     "HOBO_TCHAIN_RAW"
## [10] "METADATA"
```

3. Write a query using `tbl()` and `collect()`

Now, we're going to write a simple query to the database, mirroring a query that we might be able to write using the `sensordbTable` function.

Let's say we'd like to get data from the DO_CORR table, restricting the lake to WL.

The query would look like this:

```
dat <- tbl(con, "do_corr") %>%
    filter(lakeID == "WL") %>%
    collect()
```

Let's break down the steps here. `tbl(con, "do_corr")` establishes that we're going to be pulling data from the DO_CORR table of the data source `con`. Then, `filter(lakeID == "WL")` restricts the query to grab only data from WL. Finally, `collect()` tells R that we're done writing our query, and it should go ahead and pull the requested data.

Importantly, this is different than loading in the entire DO_CORR table and *then* filtering it to just WL rows, because the filtering is done as part of the query itself. The data isn't loaded into R's memory until *after* we've filtered it, meaning that we end up loading a smaller dataset.

So far, what I've shown you is doable with `sensordbTable()` as well. But what if you want to go beyond the arguments to that wrapper function?

3. A more complicated query

Now I'd like to pull data from EL_DeepHole_, between 0.5 and 5 meters depth. Because of the analysis I'm doing, I know I won't need information about the B3 change codes, and I also won't need any temperature information, so I can eliminate those columns from my query in order to make the final dataset even smaller.

Here's how I'd write that query:

```
dat2 <- tbl(con, "do_corr") %>% # tell R which table we want to pull data from
  # remove the unwanted columns
  select(-c("changesCodeTemp", "changesCodeDO", "cleanedTemp_C")) %>%
  filter(lakeID == "WL", # we only want data from WL
         location == "DeepHole", # and DeepHole
         depth_m <= 5, # max depth 5m
         depth_m >= 0.5) %>% # min depth 0.5m
  collect() # send that query to the database
```

Once again, the full query, incorporating our selection of columns, lake, site, location, and depths, is written before being sent to the database. The final dataset that's pulled into R is 64155 rows by 8 columns, instead of 64155 rows by 11 columns–which is the size of the WL-only data we pulled in our first query. So sure, we could have first loaded that WL-only data and *then* run these filtering operations in R after loading it, but this is more efficient.

4. Some more query examples

I can't anticipate all the queries you might want to run, so here are a bunch of examples that you can copy if they look useful.

A. List all the tables in the database

```
dbListTables(con)
```

```
##  [1] "CHANGES_CODE_KEY"     "DO_CORR"              "DO_RAW"
##  [4] "HOBO_METSTATION_CORR" "HOBO_METSTATION_RAW"  "HOBO_PRESS_CORR"
##  [7] "HOBO_PRESS_RAW"       "HOBO_TCHAIN_CORR"     "HOBO_TCHAIN_RAW"
## [10] "METADATA"             "PRECIP_CORR"          "PRECIP_RAW"
## [13] "SITES"                "UPDATE_METADATA"      "YSI_CORR"
## [16] "YSI_RAW"
```

B. List the column names in a table

```
dbListFields(con, "hobo_tchain_raw")  # case doesn't matter
```

```
## [1] "lakeID"     "location"   "depth_m"    "dateTime"   "dayfrac"
## [6] "temp_C"     "light_lux"  "metadataID" "updateID"
```

C. See the first few rows of a table

```
tbl(con, "sites") %>%
    head(10) %>%
    collect()
```

```
## # A tibble: 10 x 6
##    lakeID location   lat      long      UTM                 updateID
##    <chr>  <chr>      <chr>    <chr>     <chr>               <chr>
##  1 BA     DeepHole   46.24    -89.49    16T 307371E 5124083W originaldb.20190228
##  2 BE     DeepHole   46.24    -89.51    16T 306355E 5124660W originaldb.20190228
##  3 BO     DeepHole   46.23    -89.49    16T 307651E 5122680W originaldb.20190228
##  4 BR     DeepHole   46.21    -89.47    16T 309118E 5121214W originaldb.20190228
##  5 CB     DeepHole   46.23    -89.57    16T 301832E 5123096W originaldb.20190228
##  6 CR     DeepHole   46.21    -89.47    16T 309216E 5120406W originaldb.20190228
##  7 CR     Outlet     NA       NA        NA                  originaldb.20190228
##  8 CR     P1         46.20614 -89.47259 16T 309256E 5119925W originaldb.20190228
##  9 CR     P6         46.20898 -89.47234 16T 309285E 5120239W originaldb.20190228
## 10 CR     StaffGauge NA       NA        NA                  originaldb.20190228
```

D. Perform summary calculations before even loading the data into R, using `group_by` and `summarize`

```
# tell R which table you're going to work with
tbl(con, "do_corr") %>%
  # choose columns
  select(lakeID, dateTime, location, depth_m, cleanedDO_mg_L) %>%
  # convert the DO column to numeric
  mutate(cleanedDO_mg_L = as.numeric(cleanedDO_mg_L),
         # create a new 'year' column
         year = year(dateTime)) %>%
  # group by lake, location, and year because we want a mean/sd value for
  # each lake/location/year group
  group_by(lakeID, location, year) %>%
  # compute means on each group, removing any NA values
  summarize(mean_DO = mean(cleanedDO_mg_L, na.rm = T),
            # compute standard deviation on each group, removing any NA values
            sd_DO = sd(cleanedDO_mg_L, na.rm = T)) %>%
  collect() # pull the final data into R
```

```
## # A tibble: 78 x 5
## # Groups:   lakeID, location [19]
##    lakeID location  year mean_DO sd_DO
##    <chr>  <chr>    <int>   <dbl> <dbl>
##  1 BA     DeepHole  2012    7.66 1.85
##  2 BA     DeepHole  2013    7.65 0.685
##  3 BA     DeepHole  2014    8.25 0.909
##  4 BA     DeepHole  2015    8.97 0.582
##  5 BA     DeepHole  2016    7.77 0.667
##  6 BA     DeepHole  2017    8.45 0.685
##  7 BA     DeepHole  2018    8.37 0.899
##  8 BE     DeepHole  2012    6.66 1.33
##  9 BO     DeepHole  2012    4.74 3.57
## 10 BO     DeepHole  2013    7.18 1.78
## # ... with 68 more rows
```

So here, the whole computation is done before ever pulling the data into R.

5. Caveats

There are some calculations that can't really be done before loading the data into R with `collect()`. From the `dbplyr` documentation:

> *`collect()` requires that database does some work, so it may take a long time to complete. Otherwise, `dplyr` tries to prevent you from accidentally performing expensive query operations:*
>
> *Because there's generally no way to determine how many rows a query will return unless you actually run it, `nrow()` is always NA.*
>
> *Because you can't find the last few rows without executing the whole query, you can't use `tail()`.*

You may run into other such problems when using `tbl()`. If something is mysteriously not working, you can always go the slightly less efficient route and just load in the data first and then do your filtering/subsetting after the fact.

Conclusion

I hope this was helpful! Generating queries with `tbl()` and `collect()` is a more powerful and flexible alternative to using the `sensordbTable()` wrapper function. But that wrapper function is of course still useful, and you shouldn't hesitate to use it if that's what you're comfortable with.

Happy databasing!