6. Microclimate Workflow

Cob Staines

2025-03-07

Table of contents

This tutorial is available as a .qmd on Github.

Motivation

- Familiarize ourselves with navigating and joining data across multiple schemas
- Demonstrate a workflow to explore, join, pull, and manipulate RIBBiTR microclimate data

R

So far we have only worked with data in the `survey_data` schema. In this example we will expand our skills and familiarity to connect data between schemas in the database.

For this example, suppose we are interested in learning what microclimate time-series data exist from Panama, and then pulling available air temperature data from 2023.

Setup

In this case, we will load metadata for all schemas by dropping the `filter()` commands from previous tutorials.

```
# minimal packages for RIBBiTR DB data discovery
librarian::shelf(tidyverse, dbplyr, RPostgres, DBI, RIBBiTR-BII/ribbitrrr)

# establish database connection
dbcon <- hopToDB("ribbitr")
```

```
Connecting to 'ribbitr'... Success!
```

```
# load table metadata
mdt <- tbl(dbcon, Id("public", "all_tables")) %>%
  collect()

# load column metadata
mdc <- tbl(dbcon, Id("survey_data", "metadata_columns")) %>%
  collect()
```

Data discovery and pulling

Lets take a look at the `microclimate_data` schema diagram, we can browse to see which tables and columns we want. We can also consult the table or column metadata for the `microclimate_data` schema. This schema is pretty simple compared to the `survey_data` schema, with only a handful of tables: `logger`, `sensor`, and a number of time-series tables with names `ts_*` (e.g. `ts_temperature`). In this tutorial we will focus on temperature data, but the same workflow applies for data of other sensor types (e.g. relative humidity).

Our column metadata also contains a clue which is not obvious from the schema diagram: `microclimate_data.logger` has a foreign key which points to `survey_data.site`. knowing this, let's pull out all the ingredients we may want to work with:

Point to tables of interest

```
# pointers for all tables of interest

## observation table
db_ts_temp = tbl(dbcon, Id("microclimate_data", "ts_temperature"))

## lookup tables
db_sensor = tbl(dbcon, Id("microclimate_data", "sensor"))
db_logger = tbl(dbcon, Id("microclimate_data", "logger"))
db_site = tbl(dbcon, Id("survey_data", "site"))
db_region = tbl(dbcon, Id("survey_data", "region"))
db_country = tbl(dbcon, Id("survey_data", "country"))
```

Explore the data

While we could dive right into the time-series data to see what is there, this may mean loading lots of observations without knowing exactly what we are looking for. Instead, let's begin by exploring the lookup tables `logger` and `sensor`, to see what context we can bring without doing any heavy lifting yet.

```
# pointers for all tables of interest
pa_logger = db_logger %>%
  inner_join(db_site, "site_id") %>%
  left_join(db_region, by = "region_id") %>%
  left_join(db_country, by = "country_id") %>%
  filter(country == "panama") %>%
  collect()
```

Looking at the `pa_logger`, we see the Panama microclimate loggers for a number of sites, as well as for a number of microhabitats ("soil", "water", "sun", "shade"). For our sake, let's consider the "sun" and "shade" series only. Building on our code above:

```
pa_ss_sensor = db_sensor %>%
  left_join(db_logger, by = "logger_id") %>%
  left_join(db_site, "site_id") %>%
  left_join(db_region, by = "region_id") %>%
  left_join(db_country, by = "country_id") %>%
  filter(country == "panama",
         microhabitat %in% c("sun", "shade")) %>%
  collect()
```

As we explore the sensor data, we see variables of `sensor_type` (with values "dew_point", "temperature", "illuminance", and "relative_humidity") and `height_cm` (with values "5" and "100"). For our interest, let's consider temperature time series with sensors located at near-ground-level. Building on our code above:

```
sensors_of_interest = db_sensor %>%
  left_join(db_logger, by = "logger_id") %>%
  left_join(db_site, "site_id") %>%
  left_join(db_region, by = "region_id") %>%
  left_join(db_country, by = "country_id") %>%
  filter(country == "panama",
         microhabitat %in% c("sun", "shade"),
         sensor_type == "temperature",
         height_cm == 5) %>%
  collect()
```

Filter and pull associated time series data

Each sensor in `sensors_of_interest` has an associated time series in our database, the table in which it is stored corresponds to values of `sensor_type`. For example, all time series from sensors with `sensor_type == 'temperature'` are stored in the corresponding `ts_temperature` table. Let's pull temperature data which correspond to the `sensor_id`s in our list, filtering to the date range of interest.

NOTE ON TIMESTAMPS: The `timestamp_utc` columns contains date, time, and timezone information for each corresponding given sensor readings, corresponding to the time *in UTC* of the observations. When we filter our data to 2023, we may wish to specify the filter with start and end times in the local timezone to be clear about where our data of interest begins and ends. The `region` table provides a local `region.time_zone`, which we can refer to in our filter. If we want our data in local time, we will need to convert it to the regional timezone after pulling.

```
# pull local time zone
local_tz = unique(sensors_of_interest$time_zone)

# specify window of interest in local time, convert to utc fir query
start_datetime <- with_tz(force_tz(ymd_hms("2023-01-01 00:00:00"), tzone = local_tz), tzone
end_datetime <- with_tz(force_tz(ymd_hms("2024-01-01 00:00:00"), tzone = local_tz), tzone =

# filter and pull
ts_of_interest_long = db_ts_temp %>%
  filter(sensor_id %in% sensors_of_interest$sensor_id,
         timestamp_utc >= start_datetime,
         timestamp_utc < end_datetime) %>%
```

```
  collect() %>%
  mutate(timestamp_local = with_tz(timestamp_utc, tzone = local_tz)) # transform timestamp_u

head(ts_of_interest_long)
```

```
# A tibble: 6 x 4
  sensor_id          timestamp_utc       temperature_c_01_raw timestamp_local
  <chr>              <dttm>                             <dbl> <dttm>
1 1e412c83-f3ba-45~ 2023-01-01 05:00:00                 19.7 2023-01-01 00:00:00
2 1e412c83-f3ba-45~ 2023-01-01 05:30:00                 19.7 2023-01-01 00:30:00
3 1e412c83-f3ba-45~ 2023-01-01 06:00:00                 19.7 2023-01-01 01:00:00
4 1e412c83-f3ba-45~ 2023-01-01 06:30:00                 19.8 2023-01-01 01:30:00
5 1e412c83-f3ba-45~ 2023-01-01 07:00:00                 19.8 2023-01-01 02:00:00
6 1e412c83-f3ba-45~ 2023-01-01 07:30:00                 19.8 2023-01-01 02:30:00
```

Pivot wider

We have our data! Taking a closer look, we find our temperature data in the column `temperature_c_01_raw`, where '01_raw' tells us that these are the raw data pulled from the sensor (i.e. no filtering or gap filling).

Now, depending on our application, we may want to reformat this data so that time series are show in in parallel, each sensor having a respective column. We can use the `tidyr::pivot_wider()` function to rearrange our data to a wide format, with a few options regarding how we name the columns:

```
# column names from sensor_id
ts_of_interest_temp_c_id = ts_of_interest_long %>%
  pivot_wider(id_cols = c(timestamp_utc,
                          timestamp_local),
              names_from = sensor_id,
              values_from = temperature_c_01_raw)

# descriptive column names
ts_of_interest_temp_c_desc = sensors_of_interest %>%
  select(sensor_id,
         site_microclimate,
         microhabitat) %>%
  inner_join(ts_of_interest_long, by = "sensor_id") %>%
  select(-sensor_id) %>%
  pivot_wider(id_cols = c(timestamp_utc,
                          timestamp_local),
              names_from = c(site_microclimate,
```

```
                              microhabitat),
              values_from = temperature_c_01_raw)
```

These data are ready to be analyzed and visualized!

Disconnect

```
dbDisconnect(dbcon)
```

Python

So far we have only worked with data in the `survey_data` schema. In this example we will expand our skills and familiarity to connect data between schemas in the database.

For this example, suppose we are interested in learning what microclimate time-series data exist from Panama, and then pulling available air temperature data from 2023.

Setup

In this case, we will load metadata for all schemas by dropping the `filter()` commands from previous tutorials.

```
# minimal packages for RIBBiTR DB Workflow
import ibis
from ibis import _
import pandas as pd
import dbconfig
import db_access as db

# establish database connection
dbcon = ibis.postgres.connect(**dbconfig.ribbitr)
# recommended to set timezone to utc when working with IBIS, to keep query results consisten
# dbcon.raw_sql("SET TIME ZONE 'UTC';")
#
# peace = dbcon.raw_sql("show timezone;").fetchall()

# load table metadata
mdt = dbcon.table(database = "public", name = "all_tables").to_pandas()

# load column metadata
mdc = dbcon.table(database="public", name="all_columns").to_pandas()
```

Data discovery and pulling

Lets take a look at the `microclimate_data` schema diagram, we can browse to see which tables and columns we want. We can also consult the table or column metadata for the `microclimate_data` schema. This schema is pretty simple compared to the `survey_data` schema, with only a handful of tables: `logger`, `sensor`, and a number of time-series tables with names `ts_*` (e.g. `ts_temperature`). In this tutorial we will focus on temperature data, but the same workflow applies for data of other sensor types (e.g. relative humidity).

Our column metadata also contains a clue which is not obvious from the schema diagram: `microclimate_data.logger` has a foreign key which points to `survey_data.site`. knowing this, let's pull out all the ingredients we may want to work with:

Point to tables of interest

```
# pointers for all tables of interest

## observation tables
db_ts_temp = dbcon.table('ts_temperature', database='microclimate_data')

## lookup tables
db_sensor = dbcon.table('sensor', database='microclimate_data')
db_logger = dbcon.table('logger', database='microclimate_data')
db_site = dbcon.table('site', database='survey_data')
db_region = dbcon.table('region', database='survey_data')
db_country = dbcon.table('country', database='survey_data')
```

Explore the data

While we could dive right into the time-series data to see what is there, this may mean loading lots of observations without knowing exactly what we are looking for. Instead, let's begin by exploring the lookup tables `logger` and `sensor`, to see what context we can bring without doing any heavy lifting yet.

```
# Recursive joins
pa_logger = (
    db_logger
    .inner_join(db_site, db_logger.site_id == db_site.site_id)
    .left_join(db_region, db_site.region_id == db_region.region_id)
    .left_join(db_country, db_region.country_id == db_country.country_id)
    .filter(_.country == 'panama')
    .to_pandas()
)
```

Looking at `pa_logger`, we see the Panama microclimate loggers for a number of sites, as well as for a number of microhabitats ("soil", "water", "sun", "shade"). For our sake, let's consider the "sun" and "shade" series only. Building on our code above:

```
# Recursive joins
pa_ss_logger = (
    db_logger
    .left_join(db_site, db_logger.site_id == db_site.site_id)
    .left_join(db_region, db_site.region_id == db_region.region_id)
    .left_join(db_country, db_region.country_id == db_country.country_id)
    .filter(_.country == 'panama',
            _.microhabitat.isin(['sun','shade']))
    .to_pandas()
)
```

As we explore the sensor data, we see variables of `sensor_type` (with values "dew_point", "temperature", "illuminance", and "relative_humidity") and `height_cm` (with values "5" and "100"). For our interest, let's consider temperature time series with sensors located at near-ground-level. Building on our code above:

```
# Recursive joins
sensors_of_interest = (
    db_sensor
    .left_join(db_logger, db_sensor.logger_id == db_logger.logger_id)
    .left_join(db_site, db_logger.site_id == db_site.site_id)
    .left_join(db_region, db_site.region_id == db_region.region_id)
    .left_join(db_country, db_region.country_id == db_country.country_id)
    .filter(_.country == 'panama',
            _.microhabitat.isin(['sun','shade']),
            _.sensor_type == 'temperature',
            _.height_cm == 5)
    .to_pandas()
)
```

Filter and pull associated time series data

Each sensor in `sensors_of_interest` has an associated time series in our database, the table in which it is stored corresponds to values of `sensor_type`. For example, all time series from sensors with `sensor_type == 'temperature'` are stored in the corresponding `ts_temperature` table. Let's pull temperature data which correspond to the `sensor_id`s in our list, filtering to the date range of interest.

NOTE ON TIMESTAMPS: The `timestamp_utc` columns contains date, time, and timezone information for each corresponding given sensor readings, corresponding to the time *in*

UTC of the observations. When we filter our data to 2023, we may wish to specify the filter with start and end times in the local timezone to be clear about where our data of interest begins and ends. The `region` table provides a local `region.time_zone`, which we can refer to in our filter. If we want our data in local time, we will need to convert it to the regional timezone after pulling.

```
# pull local time zone
local_tz = sensors_of_interest['time_zone'].unique()[0]

local_tz = "Pacific/Honolulu"

# specify window of interest in local time, converted to utc for querying
start_datetime = pd.to_datetime("2023-01-01 00:00:00").tz_localize(local_tz).tz_convert("UTC
end_datetime = pd.to_datetime("2024-01-01 00:00:00").tz_localize(local_tz).tz_convert("UTC")

# filter and pull
ts_of_interest_long = (
    db_ts_temp
    .filter(_.sensor_id.isin(sensors_of_interest['sensor_id'].astype(str).tolist()),
            _.timestamp_utc >= start_datetime,
            _.timestamp_utc < end_datetime)
    .to_pandas()
)

# convert to local timezone
ts_of_interest_long['timestamp_local'] = ts_of_interest_long['timestamp_utc'].dt.tz_localize

ts_of_interest_long.head(6)
```

```
                              sensor_id  ...           timestamp_local
0  1e412c83-f3ba-453c-a70f-c2df54a0d5d6  ... 2023-01-01 00:00:00-10:00
1  1e412c83-f3ba-453c-a70f-c2df54a0d5d6  ... 2023-01-01 00:30:00-10:00
2  1e412c83-f3ba-453c-a70f-c2df54a0d5d6  ... 2023-01-01 01:00:00-10:00
3  1e412c83-f3ba-453c-a70f-c2df54a0d5d6  ... 2023-01-01 01:30:00-10:00
4  1e412c83-f3ba-453c-a70f-c2df54a0d5d6  ... 2023-01-01 02:00:00-10:00
5  1e412c83-f3ba-453c-a70f-c2df54a0d5d6  ... 2023-01-01 02:30:00-10:00

[6 rows x 4 columns]
```

Pivot wider

We have our data! Though depending on our application, we may want to reformat this data so that time series are show in in parallel, each sensor having a respective column. We can use the `pd.pivot()` function to rearrange our data to a wide format, with a few options regarding how we name the columns:

```
# column names from sensor_id
ts_of_interest_id = ts_of_interest_long.pivot(
    index='timestamp_local',
    columns='sensor_id',
    values='temperature_c_01_raw'
).reset_index()

# descriptive column names
ts_of_interest_desc = (
    sensors_of_interest[['sensor_id', 'site_microclimate', 'microhabitat']]
    .merge(ts_of_interest_long, on='sensor_id', how='inner')
    .drop(columns=['sensor_id'])
    .pivot(
        index='timestamp_local',
        columns=['site_microclimate', 'microhabitat'],
        values='temperature_c_01_raw'
    )
    .reset_index()
)
```

These data are ready to be analyzed and visualized!

Disconnect

```
# close connection
dbcon.disconnect()
```

<- 5. Bd-Capture Workflow |