6. Microclimate Workflow

Cob Staines

2024-12-18

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 database... 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, with only three substantive data tables: `logger`, `sensor`, and `time_series_01_raw`.

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 ingrediants we may want to work with:

Point to tables of interest

```
# pointers for all tables of interest
db_ts01 = tbl(dbcon, Id("microclimate_data", "time_series_01_raw"))
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 exatly what we are looking for. Instead, let's begin by exploring the support 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") %>%
  inner_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_c", "temperature_c", "intensity_lux", and "relative_humidity_percent") 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") %>%
  inner_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_c",
         height_cm == 5) %>%
  collect()
```

Filter and pull associated time series data

Each sensor in `sensors_of_interest` has an associated time series, with values in the `time_series_01_raw` table. Let's pull data which correspond to the `sensor_id`s in our list, filtering to the date range of interest.

NOTE ON TIMESTAMPS: The `timestamptz` columns contains date, time, and timezone information for each corresponding given sensor readings, corresponding to the local time and timezone of the observations. When we filter our data to 2023, we need to specify 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.

```
patz = unique(sensors_of_interest$time_zone)
start_datetime <- with_tz(ymd_hms("2023-01-01 00:00:00"), patz)
end_datetime <- with_tz(ymd_hms("2024-01-01 00:00:00"), patz)

ts_of_interest_long = db_ts01 %>%
  filter(sensor_id %in% sensors_of_interest$sensor_id,
         timestamptz >= start_datetime,
         timestamptz < end_datetime) %>%
  collect()
```

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 `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_id = ts_of_interest_long %>%
  pivot_wider(id_cols = timestamptz,
```

```
              names_from = sensor_id)

# descriptive column names
ts_of_interest_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 = timestamptz,
              names_from = c(site_microclimate,
                             microhabitat))
```

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.

Setup

```
# 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)

# 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, with only three substantive data tables: `logger`, `sensor`, and `time_series_01_raw`.

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 ingrediants we may want to work with:

Point to tables of interest

```
# pointers for all tables of interest
db_ts01 = dbcon.table('time_series_01_raw', database='microclimate_data')
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 exatly what we are looking for. Instead, let's begin by exploring the support 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
    .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',
            _.microhabitat.isin(['sun','shade']))
    .to_pandas()
)
```

As we explore the sensor data, we see variables of `sensor_type` (with values "dew_point_c", "temperature_c", "intensity_lux", and "relative_humidity_percent") 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)
    .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',
            _.microhabitat.isin(['sun','shade']),
            _.sensor_type == 'temperature_c',
            _.height_cm == 5)
    .to_pandas()
)
```

Filter and pull associated time series data

Each sensor in `sensors_of_interest` has an associated time series, with values in the `time_series_01_raw` table. Let's pull data which correspond to the `sensor_id`s in our list, filtering to the date range of interest.

NOTE ON TIMESTAMPS: The `timestamptz` columns contains date, time, and time-zone information for each corresponding given sensor readings, corresponding to the local time and timezone of the observations. When we filter our data to 2023, we need to specify 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.

```
patz = sensors_of_interest['time_zone'].unique()[0]

start_datetime = pd.to_datetime("2023-01-01 00:00:00").tz_localize(patz)
end_datetime = pd.to_datetime("2024-01-01 00:00:00").tz_localize(patz)

ts_of_interest_long = (
    db_ts01
    .filter(_.sensor_id.isin(sensors_of_interest['sensor_id'].astype(str).tolist()),
            _.timestamptz >= start_datetime,
            _.timestamptz < end_datetime)
    .to_pandas()
)
```

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='timestamptz',
    columns='sensor_id',
    values='value'
).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='timestamptz',
        columns=['site_microclimate', 'microhabitat'],
        values='value'
    )
    .reset_index()
)
```

These data are ready to be analyzed and visualized!

Disconnect

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

<- 5. Bd-Capture Workflow |