3. Data Pulling

Cob Staines

2025-01-13

Table of contents

This tutorial is available as a .qmd on Github.

Motivation

- Download or "pull" data from the database to our local machine
- Pre- select and filter to data of interest

R

Let's set up our environment to get ready to pull data.

Load packages

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

Establish database connection

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

```
Connecting to database... Success!
```

Load metadata

We recommend always loading the column metadata (and perhaps the table metadata) along with any data you are pulling. Not only will this give you a quick reference to identify what the data represent, but it will also allow us to automate some data pulling processes (more on that later).

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

# load table "all_columns" from schema "public", filtering to schema "survey_data"
mdc <- tbl(dbcon, Id("public", "all_columns")) %>%
  filter(table_schema == "survey_data") %>%
  collect()
```

Pulling data

Let's construct our first data query, building from the previous tutorial.

```
# lazy table and collect
db_ves <- tbl(dbcon, Id("survey_data", "ves")) %>%
  collect()
```

Great, that was easy! But what if we don't need all that data? Suppose we are only interested in certain columns? We can `dplyr::select()` for specific columns to avoid pulling unnecessary data:

```
# lazy table, select, and collect
db_ves <- tbl(dbcon, Id("survey_data", "ves")) %>%
  select(taxon_ves,
         count_ves,
         life_stage,
         sex,
         survey_id) %>%
  collect()
```

And perhaps we are only interested in adults, in which case we can also `dplyr::filter()` our table to desired rows before collecting:

```
# lazy table select, filter, and collect all in one
db_ves_adult <- tbl(dbcon, Id("survey_data", "ves")) %>%
  select(taxon_ves,
         count_ves,
         life_stage,
         sex,
         survey_id) %>%
  filter(life_stage == "adult") %>%
  collect()

# preview table
head(db_ves_adult)
```

```
# A tibble: 6 x 5
  taxon_ves       count_ves life_stage sex   survey_id
  <chr>               <int> <chr>      <chr> <chr>
1 rana_muscosa            1 adult      <NA>  bb18f750-d1ab-47b1-84ef-67c090d73d~
2 rana_muscosa            3 adult      <NA>  cefe838f-c113-42af-9d8f-dddac25c3b~
3 rana_muscosa            1 adult      <NA>  bf1255c8-75b0-47d8-bd92-58df7564c4~
4 rana_muscosa            7 adult      <NA>  cfbff4da-7b21-40cb-b3cb-de3a7469ab~
5 hyliola_regilla         1 adult      <NA>  6c6e5383-1ca1-47b1-a683-9cf83668f0~
6 rana_muscosa            2 adult      <NA>  f1fdd662-7820-44c0-97cd-22f6b289c3~
```

Great! The above script is an example of how we can efficiently pull the data of interest without having to pull excess data.

SQL aside

"Wait a minute... I thought these data were encoded in SQL? Where is the SQL?" Turns out, the package `dbplyr` does all the heavy lifting for us, to convert our lazy table shopping lists into SQL code which is then run on the back end without us ever having to touch it.

But if we want to see the SQL, we can! Let's take a closer look at the lazy table for our last query (dropping the `collect()` statement):

```
# lazy table only (not collected)
ves_adult <- tbl(dbcon, Id("survey_data", "ves")) %>%
  select(taxon_ves,
         count_ves,
         life_stage,
         sex,
         survey_id) %>%
  filter(life_stage == "adult")

# render sql from lazy table
(ves_adult_q = sql_render(ves_adult))
```

```
<SQL> SELECT "taxon_ves", "count_ves", "life_stage", "sex", "survey_id"
FROM "survey_data"."ves"
WHERE ("life_stage" = 'adult')
```

The `dbplyr::sql_render()` function converts our lazy table "shopping list" into an SQL script. If we want we can interact with this script, and even send it to the database manually using `DBI::dbGetQuery()`

```
# execute SQL statement and return results
db_ves_adult_sql <- dbGetQuery(dbcon, ves_adult_q)

# preview table
head(db_ves_adult_sql)
```

```
        taxon_ves count_ves life_stage  sex
1    rana_muscosa         1      adult <NA>
2    rana_muscosa         3      adult <NA>
3    rana_muscosa         1      adult <NA>
4    rana_muscosa         7      adult <NA>
5 hyliola_regilla         1      adult <NA>
6    rana_muscosa         2      adult <NA>
                             survey_id
1 bb18f750-d1ab-47b1-84ef-67c090d73da8
```

```
2 cefe838f-c113-42af-9d8f-dddac25c3b20
3 bf1255c8-75b0-47d8-bd92-58df7564c4d9
4 cfbff4da-7b21-40cb-b3cb-de3a7469ab76
5 6c6e5383-1ca1-47b1-a683-9cf83668f0b8
6 f1fdd662-7820-44c0-97cd-22f6b289c3e2
```

On close inspection we see that this is identical to the db_ves_adult_sql above. Now you have two ways to access the same data!

We will stick with the dplyr/dbplyr methods for the rest of this tutorial, but feel free to integrate this with your curiousity and/or knowledge of SQL as we go forward.

Disconnect

```
dbDisconnect(dbcon)
```

Also try:

- Check out the SQL code for your last query with:

```
sql_render(db_ves_adult_final)
```

Python

Let's set up our environment to get ready to pull data.

Load packages

```
# minimal packages for RIBBiTR DB data discovery
import ibis
from ibis import _
import pandas as pd
import dbconfig
```

Establish database connection

```
# establish database connection
dbcon = ibis.postgres.connect(**dbconfig.ribbitr)
```

Load metadata

We recommend always loading the column metadata (and perhaps the table metadata) along with any data you are pulling. Not only will this give you a quick reference to identify what the data represent, but it will also allow us to automate some data pulling processes (more on that later).

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

# load table "all_columns" from schema "public", filtering to schema "survey_data"
mdc = (
  dbcon.table(database="public", name="all_columns")
  .filter(_.table_schema == 'survey_data')
  .to_pandas()
)
```

Pulling data

Let's construct our first data query, building from the previous tutorial.

```
# lazy table and collect
db_ves = dbcon.table(database="survey_data", name="ves").to_pandas()
db_ves.columns
```

```
Index(['taxon_ves', 'count_ves', 'detection_location', 'microhab',
       'life_stage', 'sex', 'comments_ves', 'microhab_moredetail',
       'observer_ves', 'visual_animal_state', 'ves_id', 'survey_id'],
      dtype='object')
```

Great, that was easy! But what if we don't need all that data? Suppose we are only interested in certain columns? We can `ibis.select()` for specific columns to avoid pulling unnecessary data:

```
# lazy table, select, and collect
db_ves_select = (
  dbcon.table(database="survey_data", name="ves")
  .select([
```

```
    'taxon_ves',
    'count_ves',
    'life_stage',
    'sex',
    'survey_id'
  ])
  .to_pandas()
  )

db_ves_select.columns
```

```
Index(['taxon_ves', 'count_ves', 'life_stage', 'sex', 'survey_id'], dtype='object')
```

And perhaps we are only interested in adults, in which case we can also `ibis.filter()` our table to desired rows before collecting:

```
# lazy table select, filter, and collect all in one
db_ves_adult = (
  dbcon.table(database="survey_data", name="ves")
  .select([
    'taxon_ves',
    'count_ves',
    'life_stage',
    'sex',
    'survey_id'
  ])
  .filter(_.life_stage == 'adult')
  .to_pandas()
  )

# preview table
db_ves_adult.head()
```

```
         taxon_ves  count_ves  ...   sex                              survey_id
0     rana_muscosa        1.0  ...  None  bb18f750-d1ab-47b1-84ef-67c090d73da8
1     rana_muscosa        3.0  ...  None  cefe838f-c113-42af-9d8f-dddac25c3b20
2     rana_muscosa        1.0  ...  None  bf1255c8-75b0-47d8-bd92-58df7564c4d9
3     rana_muscosa        7.0  ...  None  cfbff4da-7b21-40cb-b3cb-de3a7469ab76
4  hyliola_regilla        1.0  ...  None  6c6e5383-1ca1-47b1-a683-9cf83668f0b8

[5 rows x 5 columns]
```

Great! The above script is an example of how we can efficiently pull the data of interest without having to pull excess data.

SQL aside

"Wait a minute… I thought these data were encoded in SQL? Where is the SQL?" Turns out, the package `ibis` does all the heavy lifting for us, to convert our lazy table shopping lists into SQL code which is then run on the back end without us ever having to touch it.

But if we want to see the SQL, we can! Let's take a closer look at the lazy table for our last query (dropping the `to_pandas()` statement):

```
# lazy table only (not collected)
ves_adult = (
  dbcon.table(database="survey_data", name="ves")
  .select([
    'taxon_ves',
    'count_ves',
    'life_stage',
    'sex',
    'survey_id'
  ])
  .filter(_.life_stage == 'adult')
  )

# render sql from lazy table
ves_adult.compile()
```

```
'SELECT "t0"."taxon_ves", "t0"."count_ves", "t0"."life_stage", "t0"."sex", "t0"."survey_id"
```

The `ibis.compile()` function converts our lazy table "shopping list" into an SQL script. If we want we can revise with this script, and even send it to the database manually using dedicated python - SQL packages such as psycopg2 or SQLAlchemy.

Disconnect

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

Also try:

- Check out the SQL code for your last query with:

```
db_ves_adult_final.compile()
```

<- 2. Data Discovery | 4. Table Joins ->