2. Data Discovery

Cob Staines

2025-04-07

Table of contents

This tutorial is available as a .qmd on Github.

Motivation

- Explore what data are currently available on the database
- Identify structure of data of interest to inform access

R

Let's set up our environment to get ready to explore the database.

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 'ribbitr'... Success!
```

Load database metadata

Data structure: Schemas, tables, columns and rows

The RIBBiTR database is organized into "schemas" (think of these as folders), which can contain any number of tables. Each table consists of columns ("variables") and rows ("entries").

Metadata: Data about data

We keep track of information regarding what tables, and columns exist in the database, and what information they are designed to describe, using table and column metadata. To begin our process of data discovery, let's learn what tables are present in the data by loading the table metadata.

Table Metadata

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

Some basic database commands

Before we take a look at the metadata you just pulled, let's understand the command we just ran.

- `dplyr::tbl()` - This function is used to create a "lazy" table from a data source. To specify the source, we provide the database connection `dbcon`, as well as a pointer or "address" for the table of interest using the `Id()` function. A "lazy" table means that the data only pulled when explicitly asked for. See `collect()` below.
- `dbplyr::Id()` - This function is a pointer to pass hierarchical table identifiers (you can think of this as an address for a given table). In this case we use it to generate an pointer for the table "all_tables" in schema "public".
- `dplyr::collect()` - the `tbl()` function generates a "lazy" table, which is basically a shopping list for the data you want to pull. In order to actually pull the data from the server to your local machine (ie. "do the shopping") we need to pipe in the `collect()` function.

Also try: Run the code above without `collect()`, to see what a lazy table looks like.

Now let's take a look at the table metadata to explore what schemas and tables exist.

```
view(mdt)
```

Column metadata

Suppose our interest is in the `survey_data` schema. Let's take a closer look at the tables here by collecting metadata on table columns in this schema.

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

Notice we used the `dplyr::filter()` command on the lazy table *before* running `collect()`. This effectively revised the shopping list before going to the store, rather than bringing home the entire store and then filtering for what you want in your kitchen. Much less (computationally) expensive!

Let's check out the column metadata, and see what you can learn.

```
view(mdc)

# list the columns in our column-metadata table
colnames(mdc)
```

```
 [1] "table_schema"             "table_name"
 [3] "column_name"              "definition"
 [5] "units"                    "accuracy"
 [7] "scale"                    "format"
 [9] "reviewed"                 "natural_key"
[11] "primary_key"              "foreign_key"
[13] "unique"                   "is_nullable"
[15] "data_type"                "character_maximum_length"
[17] "numeric_precision"        "datetime_precision"
[19] "column_default"           "ordinal_position"
[21] "pg_description"           "key_type"
[23] "fkey_ref_schema"          "fkey_ref_table"
[25] "fkey_ref_column"
```

Curious about what a certain metadata column means? There's metadata for that (metametadata?)!

```
# vew metadata on metadata columns
view(mdc %>% filter(table_name == "metadata_columns"))
```

A few columns to point out:

- definition
- units
- data_type
- natural key

(more on keys later)

Our first(?) data table

Ok, let's try to apply some of what we have learned by pulling directly from a data table. We can begin by taking a look at the visual encounter surveys (VES).

```
# create lazy table for ves (visual encounter survey) table
db_ves <- tbl(dbcon, Id("survey_data", "ves"))
```

Do these functions look familiar? Turns out, we were pulling data all along! Of course, this is a lazy table (ie. shopping list) so it doesn't look like data yet. Let's see what we can learn from it before going to the store to collect the data.

What columns the table contains:

```
# return columns of lazy table
colnames(db_ves)
```

```
 [1] "taxon_ves"             "count_ves"             "ves_transect_m"
 [4] "microhabitat_type"     "life_stage"            "sex"
 [7] "comments_ves"          "microhabitat_detailed" "observer_ves"
[10] "visual_animal_state"   "ves_id"                "survey_id"
```

How many total rows a table contains:

```
# count rows
db_ves %>%
  count() %>%
  pull()
```

```
integer64
[1] 30124
```

The pull() function executes a query to return a single column or variable, synonymous with the collect() function which returns a collection of variables as a table.

We can also count rows using the `summarise()` and `n()` functions. While slightly more complicated, this will carry over into more colicated queries later.

So, how many rows after filtering for unknown species:

```
# count rows with known taxa
db_ves %>%
  filter(!is.na(taxon_ves)) %>%
  summarise(row_count = n()) %>%
  pull(row_count)
```

```
integer64
[1] 29949
```

How many rows corresponding to a each life stage:

```
# count rows by life stage
db_ves %>%
  select(life_stage) %>%
  group_by(life_stage) %>%
  summarise(row_count = n()) %>%
  arrange(desc(row_count)) %>%
  collect()
```

```
# A tibble: 8 x 2
  life_stage    row_count
  <chr>           <int64>
1 tadpole           10568
2 adult              9831
3 subadult           7380
4 <NA>               1591
5 eggmass             655
6 juvenile             76
7 egg                  16
8 metamorphosed         7
```

Disconnect

Reinforcing best practice by disconnecting from the server.

```
dbDisconnect(dbcon)
```

Python

Let's set up our environment to get ready to explore the database.

Load packages

```
# minimal packages for Python 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 database metadata

Data structure: Schemas, tables, columns and rows

The RIBBiTR database is organized into "schemas" (think of these as folders), which can contain any number of tables. Each table consists of columns ("variables") and rows ("entries").

Metadata: Data about data

We keep track of information regarding what tables, and columns exist in the database, and what information they are designed to describe, using table and column metadata. To begin our process of data discovery, let's learn what tables are present in the data by loading the table metadata.

Table Metadata

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

Some basic database commands

Before we take a look at the metadata you just pulled, let's understand the command we just ran.

- `ibis.table()` - This function is used to create a "lazy" table from a data source. To specify the source, we modify the database connection `dbcon`. We specify the schema for the table as `public` (note ibis calls this "database"), as well as the table name `all_tables`. A "lazy" table means that the data only pulled when explicitly asked for. See `execute()` below.
- `ibis.to_pandas()` - the `table()` function generates a "lazy" table, which is basically a shopping list for the data you want to pull. In order to actually pull the data from the server to your local machine (ie. "do the shopping") we need to collect the lazy table by chaining the `to_pandas()` function.

Also try: Run the code above without `to_pandas()`, to see what an uncollected lazy table looks like.

Now let's take a look at the table metadata to explore what schemas and tables exist.

```
print(mdt)
```

```
        table_schema  ...                                table_description
0   microclimate_data  ...  data logger lookup table (one logger contains ...
1   microclimate_data  ...  Column-specific definitions and metadata ("dat...
2   microclimate_data  ...  Table-specific definitions and metadata ("data...
3   microclimate_data  ...                    microclimate sensor lookup table
4   microclimate_data  ...                    Dew point time series data table
5   microclimate_data  ...                  Illuminance time series data table
6   microclimate_data  ...            Relative humidity time series data table
7   microclimate_data  ...                  Temperature time series data table
8         survey_data  ...                                               None
9         survey_data  ...                                               None
10        survey_data  ...  Table-specific definitions and metadata ("data...
11        survey_data  ...         Results from mucosome-Bd-inhibition assays
12        survey_data  ...                                  Region lookup table
13        survey_data  ...                                               None
14        survey_data  ...  sample lookup table for tracking sample result...
15        survey_data  ...                               Field site lookup table
16        survey_data  ...  Tracking of human interventions, manipulations...
17        survey_data  ...                            Field survey lookup table
18        survey_data  ...                               Taxonomy lookup table
19        survey_data  ...                    Field survey visual observations
20        survey_data  ...                                               None
21        survey_data  ...                                               None
22        survey_data  ...                            Field visit lookup table
23        survey_data  ...                                               None
24        survey_data  ...                                               None
25        survey_data  ...                                               None
26        survey_data  ...                     Field survey aural observations
```

```
27        survey_data  ...          Isolate Bd strains for use wihtin RIBBiTR
28        survey_data  ...                      Results from Bd qPCR assays
29        survey_data  ...                  Field survey capture observations
30        survey_data  ...  Capture-mark-recapture codes for tracking of m...
31        survey_data  ...                                Country lookup table
32        survey_data  ...               eDNA sample collection and filtering
33        survey_data  ...            Field survey environmental observations
34        survey_data  ...  laboratory lookup table -- use to identify who...
35        survey_data  ...  Column-specific definitions and metadata ("dat...

[36 rows x 4 columns]
```

Column metadata

Suppose our interest is in the `survey_data` schema. Let's take a closer look at the tables here by collecting metadata on table columns in this schema.

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

Notice we used the `ibis.filter()` command on the lazy table *before* calling `to_pandas()`. This effectively revised the shopping list before going to the store, rather than bringing home the entire store and then filtering for what you want in your kitchen. Much less (computationally) expensive!

Let's check out the column metadata, and see what you can learn.

```
# view dataframe
print(mdc)
```

```
    table_schema        table_name  ... fkey_ref_table fkey_ref_column
0    survey_data   bd_qpcr_results  ...           None            None
1    survey_data           capture  ...           None            None
2    survey_data  metadata_columns  ...           None            None
3    survey_data  metadata_columns  ...           None            None
4    survey_data   bd_qpcr_results  ...           None            None
..           ...               ...  ...            ...             ...
441  survey_data        bd_isolate  ...           None            None
442  survey_data        bd_isolate  ...           None            None
443  survey_data        bd_isolate  ...            lab          lab_id
```

```
444  survey_data  mucosome_results  ...            lab          lab_id
445  survey_data  mucosome_results  ...     bd_isolate   bd_isolate_id

[446 rows x 25 columns]
```

```
# list the columns in our column-metadata table
mdc.columns
```

```
Index(['table_schema', 'table_name', 'column_name', 'definition', 'units',
       'accuracy', 'scale', 'format', 'reviewed', 'natural_key', 'primary_key',
       'foreign_key', 'unique', 'is_nullable', 'data_type',
       'character_maximum_length', 'numeric_precision', 'datetime_precision',
       'column_default', 'ordinal_position', 'pg_description', 'key_type',
       'fkey_ref_schema', 'fkey_ref_table', 'fkey_ref_column'],
      dtype='object')
```

Curious about what a certain metadata column means? There's metadata for that (metametadata?)!

```
# view metadata on metadata columns
metameta = mdc[mdc['table_name'] == 'metadata_columns']
print(metameta)
```

```
    table_schema        table_name  ... fkey_ref_table fkey_ref_column
2    survey_data  metadata_columns  ...           None            None
3    survey_data  metadata_columns  ...           None            None
9    survey_data  metadata_columns  ...           None            None
34   survey_data  metadata_columns  ...           None            None
57   survey_data  metadata_columns  ...           None            None
91   survey_data  metadata_columns  ...           None            None
246  survey_data  metadata_columns  ...           None            None
247  survey_data  metadata_columns  ...           None            None
248  survey_data  metadata_columns  ...           None            None
304  survey_data  metadata_columns  ...           None            None
305  survey_data  metadata_columns  ...           None            None
306  survey_data  metadata_columns  ...           None            None
307  survey_data  metadata_columns  ...           None            None
308  survey_data  metadata_columns  ...           None            None
309  survey_data  metadata_columns  ...           None            None
310  survey_data  metadata_columns  ...           None            None
311  survey_data  metadata_columns  ...           None            None
312  survey_data  metadata_columns  ...           None            None
313  survey_data  metadata_columns  ...           None            None
```

```
314  survey_data  metadata_columns  ...          None           None
342  survey_data  metadata_columns  ...          None           None
348  survey_data  metadata_columns  ...          None           None
373  survey_data  metadata_columns  ...          None           None
388  survey_data  metadata_columns  ...          None           None
389  survey_data  metadata_columns  ...          None           None

[25 rows x 25 columns]
```

A few columns to point out:

- definition
- units
- data_type
- natural key

(more on keys later)

Our first(?) data table

Ok, let's try to apply some of what we have learned by pulling directly from a data table. We can begin by taking a look at the visual encounter surveys (VES).

```
# create lazy table for ves (visual encounter survey) table
db_ves = dbcon.table(database="survey_data", name="ves")
```

Do these functions look familiar? Turns out, we were pulling data all along! Of course, this is a lazy table (ie. shopping list) so it doesn't look like data yet. Let's see what we can learn from it before going to the store to collect the data.

What columns the table contains:

```
# return columns of lazy table
db_ves.columns
```

```
['taxon_ves', 'count_ves', 'ves_transect_m', 'microhabitat_type', 'life_stage', 'sex', 'comm
```

How many total rows a table contains:

```
# count rows
(db_ves
 .count()
 .execute())
```

```
30124
```

The ibis.execute() function executes a query and returns the result, regardless of the format. This is synonymous with the to_pandas() function which returns query results as a pandas dataframe where possible.

How many rows after filtering for unknown species:

```
# count rows with known taxa
filtered_row_count = (
  db_ves
  .filter(_.taxon_ves.notnull())
  .count()
  .execute())

print(filtered_row_count)
```

```
29949
```

How many rows corresponding to a each life stage:

```
# count rows by life stage
life_stage_counts = (
    db_ves.group_by('life_stage')
    .aggregate(row_count=_.count())
    .order_by(_.row_count.desc())
    .to_pandas()
)

print(life_stage_counts)
```

```
      life_stage  row_count
0        tadpole      10568
1          adult       9831
2       subadult       7380
3           None       1591
4        eggmass        655
5       juvenile         76
6            egg         16
7  metamorphosed          7
```

Disconnect

Reinforcing best practice by disconnecting from the server.

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

DBeaver

Double-click on the `ribbitr` connection in the "Database Navigator" panel to begin your connection. Once connected you should be able to navigae a dropdown menu to explore the connection.

Load database metadata

Data structure: Schemas, tables, columns and rows

The RIBBiTR database is organized into "schemas" (think of these as folders), which can contain any number of tables. Each table consists of columns ("variables") and rows ("entries"). You can explore this structure through the dropdown menu in the "Database Navigator" panel on the left.

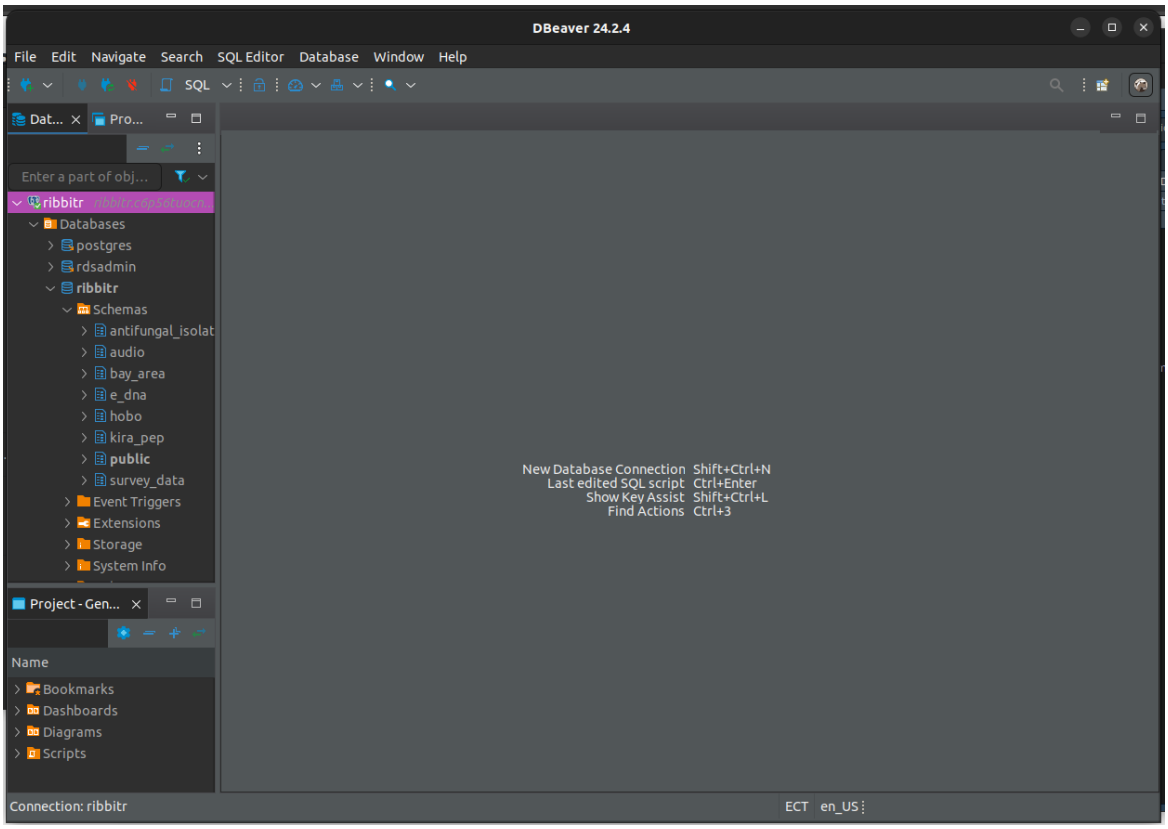
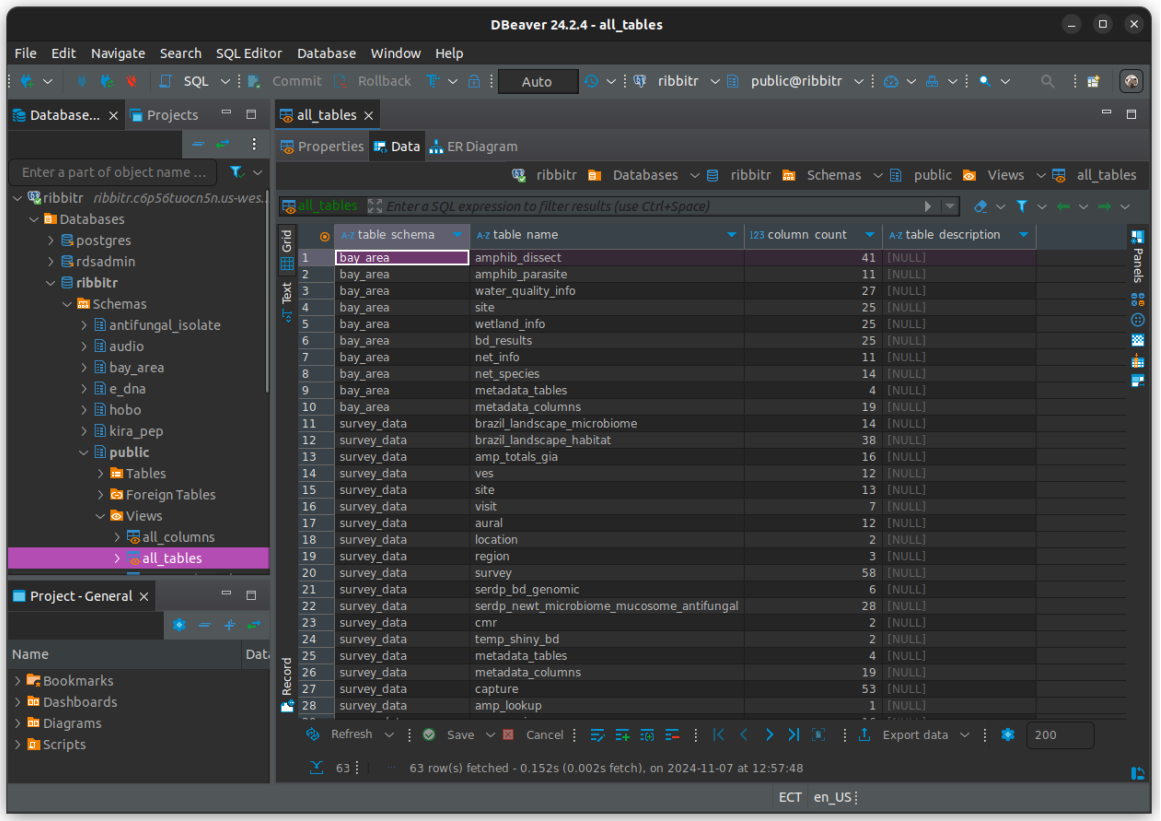


Figure 1: Navigate to `Databases` -> `ribbitr` -> `Schemas`

Metadata: Data about data

We keep track of information regarding what tables, and columns exist in the database, and what information they are designed to describe, using table and column metadata. To begin our process of data discovery, let's learn what tables are present in the data by loading the table metadata.

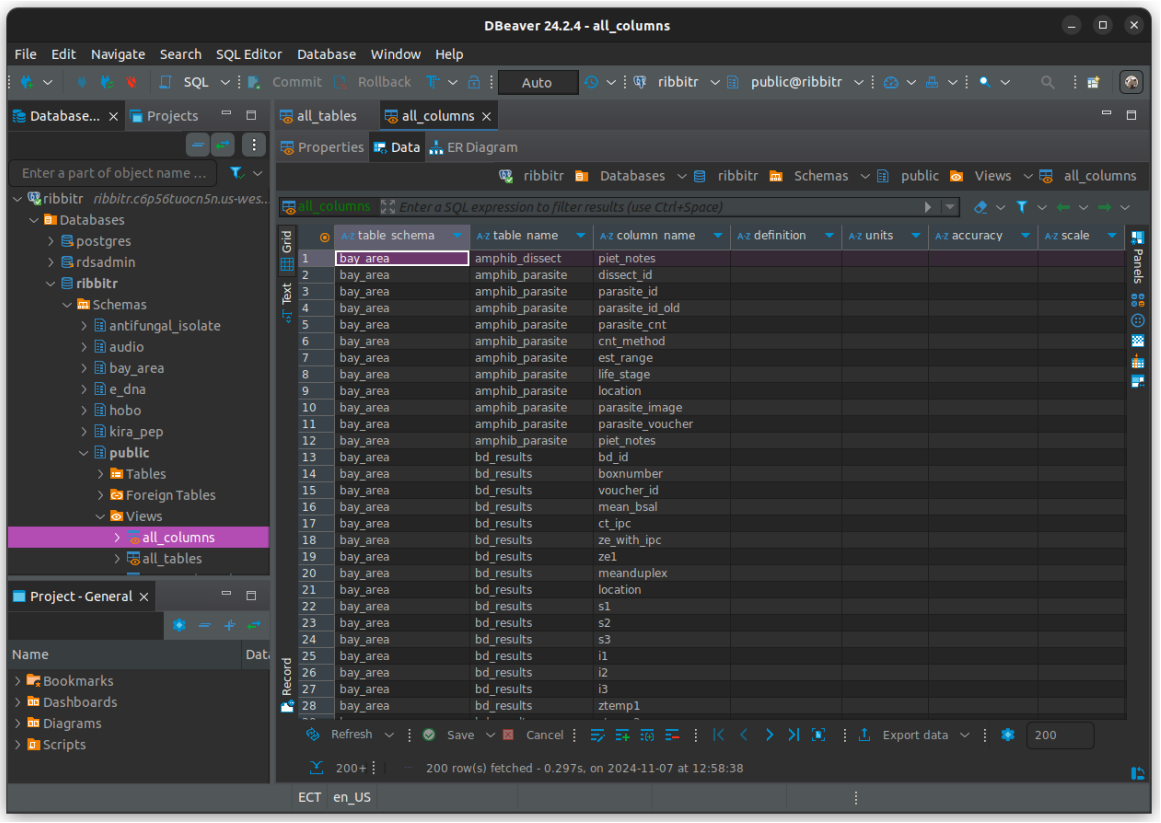
Table Metadata



See what you can learn about the tables in the database form the table metadata.

Column metadata

Suppose our interest is in the `survey_data` schema. Let's take a closer look at the tables here by collecting metadata on table columns in this schema.



Click on the dropdown arrow next to `table_schema`, click on `Order by table_schema ASC`. Repeat for the `table_name` and `column_name` columns.

Scroll down until you see rows with `table_schema` = `survey_data`. Explore a table of interest t see what you can learn.

Curious about what a certain metadata column means? There's metadata for that (metametadata?)! Scroll down to `table_name` = `metadata_columns` to learn what the different columns in the current table mean.

A few columns to point out:

- definition
- units
- data_type
- natural key

(more on keys later)

Schema sructure

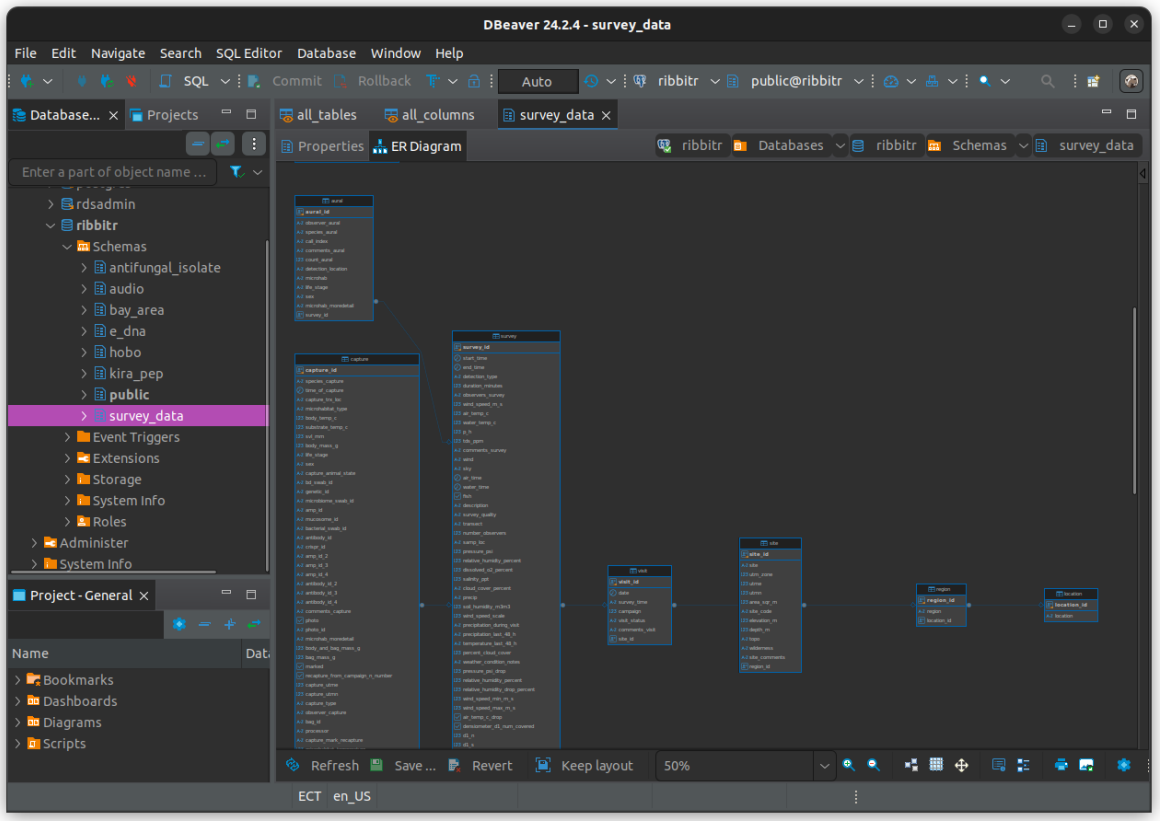
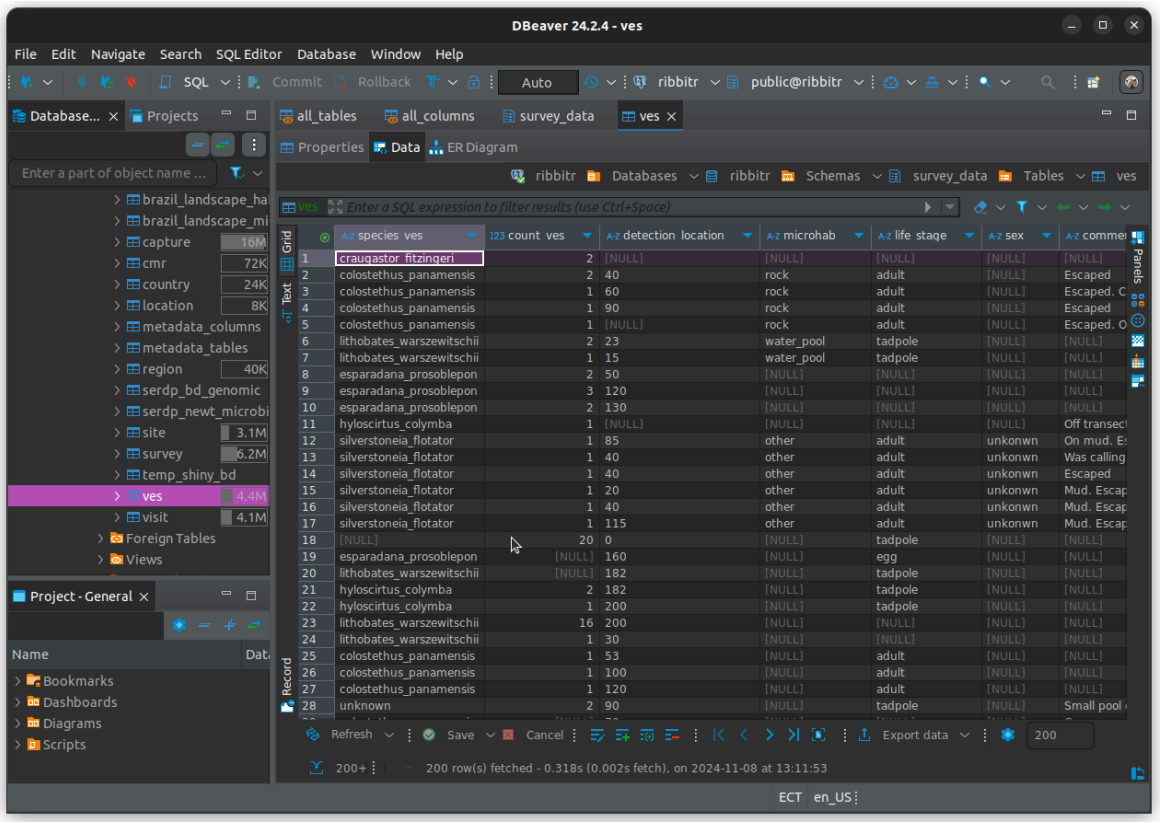


Figure 2: Navigate to `Databases` -> `ribbitr` -> `Schemas` -> `survey_data`. Right-click and select `View Schema`. Select the `ER Diagram` tab.

This shows a diagram of the different tables within the `surevy_data` schema, as well as their columns and any relationships between tables. This is a useful visual reference for later, when we begin joining tables.

Our first data table

To begin looking at data, let's navigate to the visual encounter surveys (VES).



This is your first look at field data within the database! From here you can explore organizing the data by columns, as well as exporting the table to a .csv.

<- 1. Connection Setup | 3. Data Pulling ->