

2. Data Discovery

Cob Staines

2024-12-17

Table of contents

Motivation	2
R	2
Load packages	2
Establish database connection	2
Load database metadata	2
Data structure: Schemas, tables, columns and rows	2
Metadata: Data about data	3
Table Metadata	3
Column metadata	3
Our first(?) data table	5
Disconnect	6
Python	6
Load packages	6
Establish database connection	7
Load database metadata	7
Data structure: Schemas, tables, columns and rows	7
Metadata: Data about data	7
Table Metadata	7
Column metadata	8
Our first(?) data table	10
Disconnect	12
DBeaver	12
Load database metadata	12
Data structure: Schemas, tables, columns and rows	12
Metadata: Data about data	13
Table Metadata	14
Column metadata	14
Schema sructure	16

Our first data table	16
--------------------------------	----

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 database... 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?)!

```
# view 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] "species_ves"      "count_ves"      "detection_location"
[4] "microhab"         "life_stage"     "sex"
[7] "comments_ves"     "microhab_moredetail" "observer_ves"
[10] "visual_animal_state" "ves_id"         "survey_id"
```

How many total rows a table contains:

```
# count rows
db_ves %>%
  summarise(row_count = n()) %>%
  pull(row_count)
```

```
integer64
[1] 29507
```

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.

How many rows after filtering for unknown species:

```
# count rows with known species
db_ves %>%
  filter(!is.na(species_ves),
         species_ves != "unknown_species") %>%
  summarise(row_count = n()) %>%
  pull(row_count)
```

```
integer64
[1] 29344
```

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: 9 x 2
  life_stage    row_count
  <chr>         <int64>
1 tadpole         10276
2 adult            9617
3 subadult        7162
4 <NA>            1725
5 eggmass          625
6 juvenile         78
7 egg              16
8 metamorphosed     7
9 metamorph         1
```

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_name	column_count	table_description
0	bay_area	amphib_dissect	41	None
1	bay_area	amphib_parasite	11	None
2	bay_area	water_quality_info	27	None
3	bay_area	site	25	None
4	bay_area	wetland_info	25	None
..
59	microclimate_data	time_series	3	None
60	microclimate_data	logger	5	None
61	microclimate_data	sensor	4	None
62	microclimate_data	metadata_columns	25	None
63	microclimate_data	metadata_tables	4	None

[64 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	site	...	None	None
1	survey_data	site	...	None	None
2	survey_data	capture	...	None	None
3	survey_data	metadata_columns	...	None	None
4	survey_data	metadata_columns	...	None	None
..
355	survey_data	bd_qpcr_results	...	None	None
356	survey_data	bd_qpcr_results	...	sample	sample_id
357	survey_data	bd_qpcr_results	...	None	None
358	survey_data	bd_qpcr_results	...	None	None
359	survey_data	site	...	None	None

[360 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
3	survey_data	metadata_columns	...	None	None
4	survey_data	metadata_columns	...	None	None
5	survey_data	metadata_columns	...	None	None
6	survey_data	metadata_columns	...	None	None
62	survey_data	metadata_columns	...	None	None
183	survey_data	metadata_columns	...	None	None
184	survey_data	metadata_columns	...	None	None
185	survey_data	metadata_columns	...	None	None
249	survey_data	metadata_columns	...	None	None
250	survey_data	metadata_columns	...	None	None
251	survey_data	metadata_columns	...	None	None
252	survey_data	metadata_columns	...	None	None
253	survey_data	metadata_columns	...	None	None
254	survey_data	metadata_columns	...	None	None
255	survey_data	metadata_columns	...	None	None
256	survey_data	metadata_columns	...	None	None
257	survey_data	metadata_columns	...	None	None
258	survey_data	metadata_columns	...	None	None
259	survey_data	metadata_columns	...	None	None
269	survey_data	metadata_columns	...	None	None
270	survey_data	metadata_columns	...	None	None
271	survey_data	metadata_columns	...	None	None
287	survey_data	metadata_columns	...	None	None
288	survey_data	metadata_columns	...	None	None
289	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
```

```
['species_ves', 'count_ves', 'detection_location', 'microhab', 'life_stage', 'sex', 'comment']
```

How many total rows a table contains:

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

29507

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 species
filtered_row_count = (
    db_ves
    .filter(_.species_ves.notnull() & (_.species_ves != 'unknown_species'))
    .count()
    .execute())

print(filtered_row_count)
```

29344

How many rows corresponding to a each life stage:

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

print(life_stage_counts)
```

	life_stage	row_count
0	tadpole	10276
1	adult	9617
2	subadult	7162
3	None	1725
4	eggmass	625
5	juvenile	78
6	egg	16
7	metamorphosed	7
8	metamorph	1

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 navigate 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.

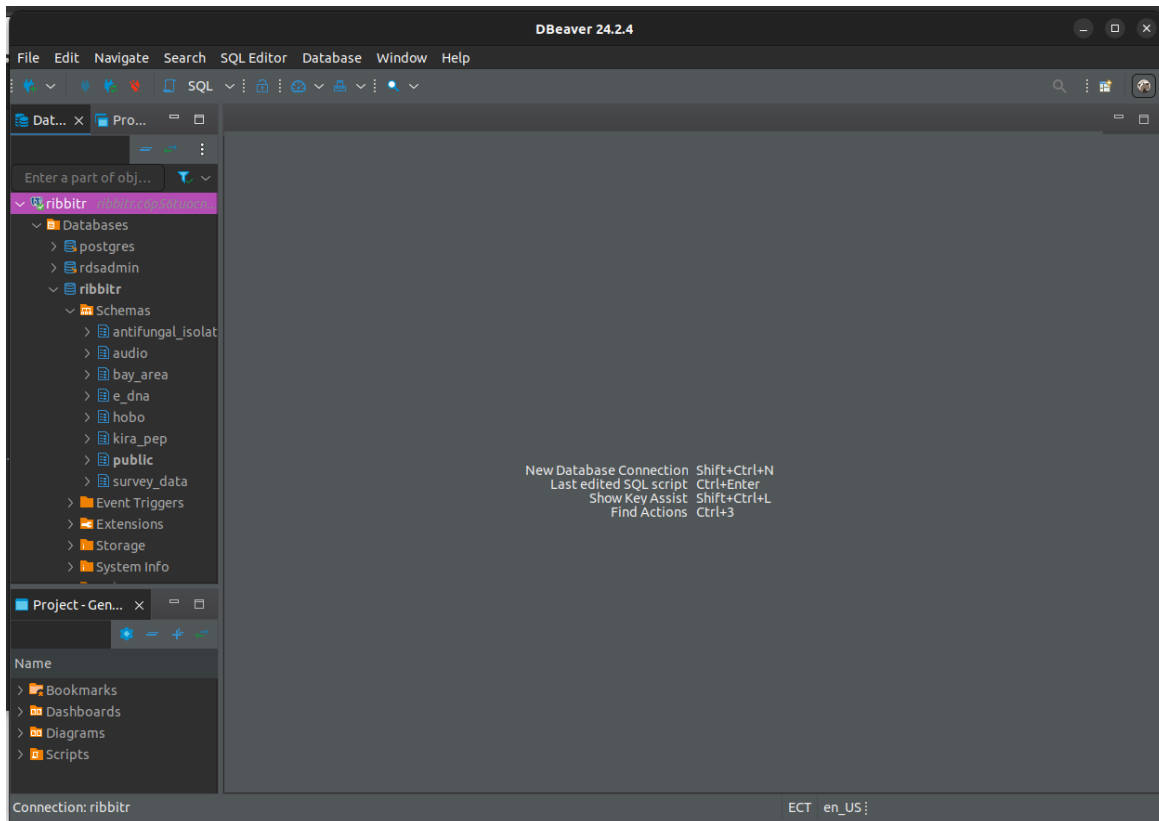
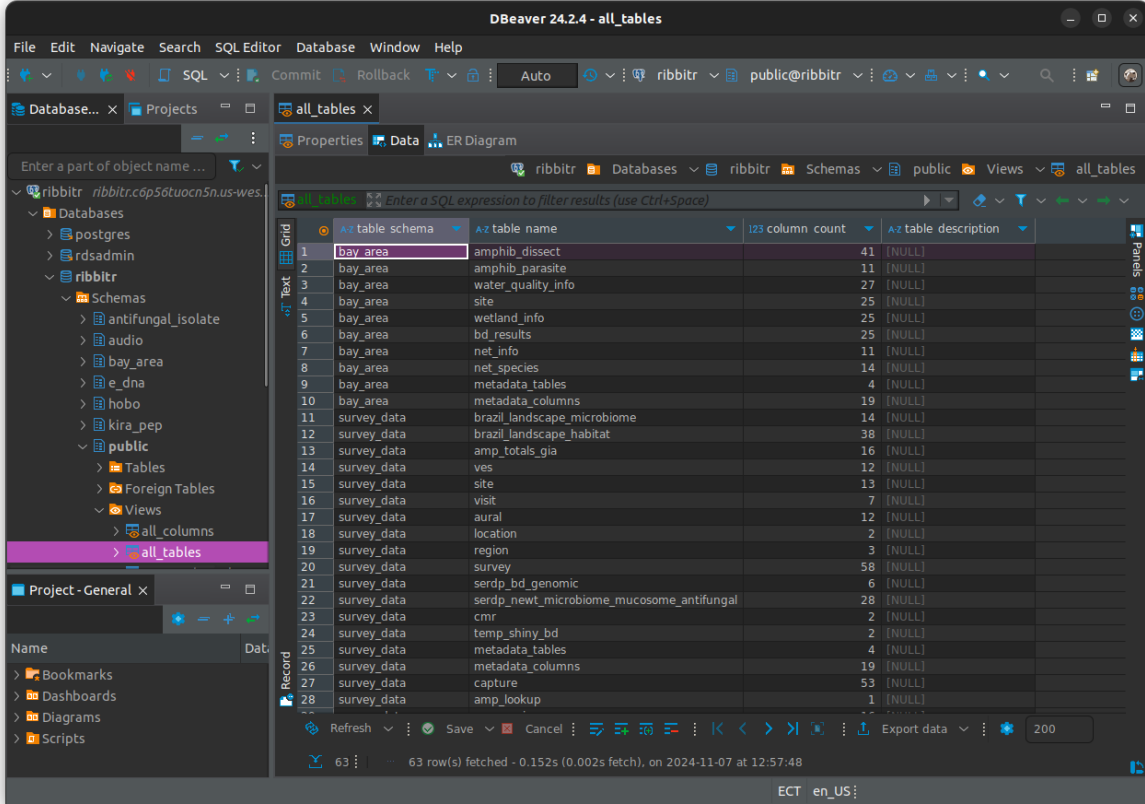


Figure 1: Navigate to Databases -> ribbitr -> Schemas

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



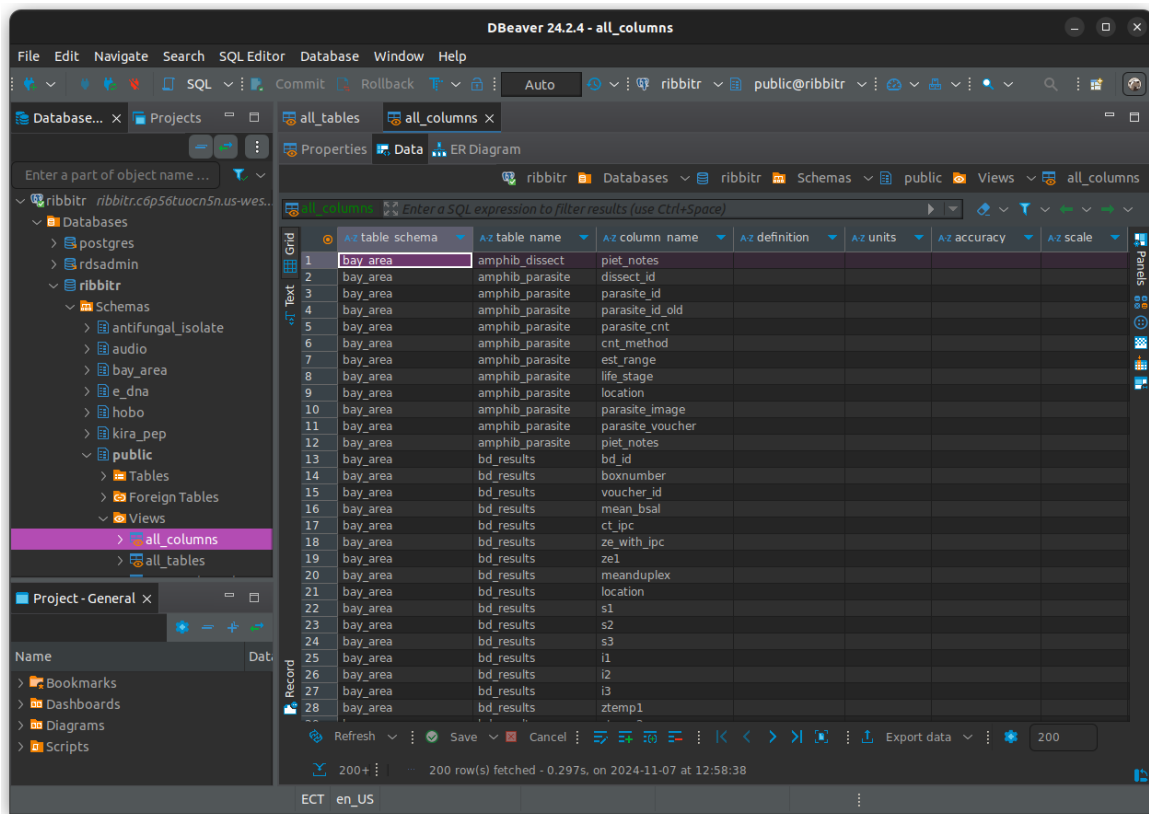
The screenshot shows the DBeaver 24.2.4 interface. The left sidebar displays a tree view of the database structure, with the 'public' schema and 'all_tables' view selected. The main panel shows a table with columns: 'table schema', 'table name', 'column count', and 'table description'. The table contains 28 rows of data, listing various tables and their metadata.

	table schema	table name	column count	table description
1	bay_area	amphib_dissect	41	[NULL]
2	bay_area	amphib_parasite	11	[NULL]
3	bay_area	water_quality_info	27	[NULL]
4	bay_area	site	25	[NULL]
5	bay_area	wetland_info	25	[NULL]
6	bay_area	bd_results	25	[NULL]
7	bay_area	net_info	11	[NULL]
8	bay_area	net_species	14	[NULL]
9	bay_area	metadata_tables	4	[NULL]
10	bay_area	metadata_columns	19	[NULL]
11	survey_data	brazil_landscape_microbiome	14	[NULL]
12	survey_data	brazil_landscape_habitat	38	[NULL]
13	survey_data	amp_totals_gia	16	[NULL]
14	survey_data	ves	12	[NULL]
15	survey_data	site	13	[NULL]
16	survey_data	visit	7	[NULL]
17	survey_data	aural	12	[NULL]
18	survey_data	location	2	[NULL]
19	survey_data	region	3	[NULL]
20	survey_data	survey	58	[NULL]
21	survey_data	serdp_bd_genomic	6	[NULL]
22	survey_data	serdp_newt_microbiome_mucosome_antifungal	28	[NULL]
23	survey_data	cmr	2	[NULL]
24	survey_data	temp_shiny_bd	2	[NULL]
25	survey_data	metadata_tables	4	[NULL]
26	survey_data	metadata_columns	19	[NULL]
27	survey_data	capture	53	[NULL]
28	survey_data	amp_lookup	1	[NULL]

See what you can learn about the tables in the database from 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 to 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 structure

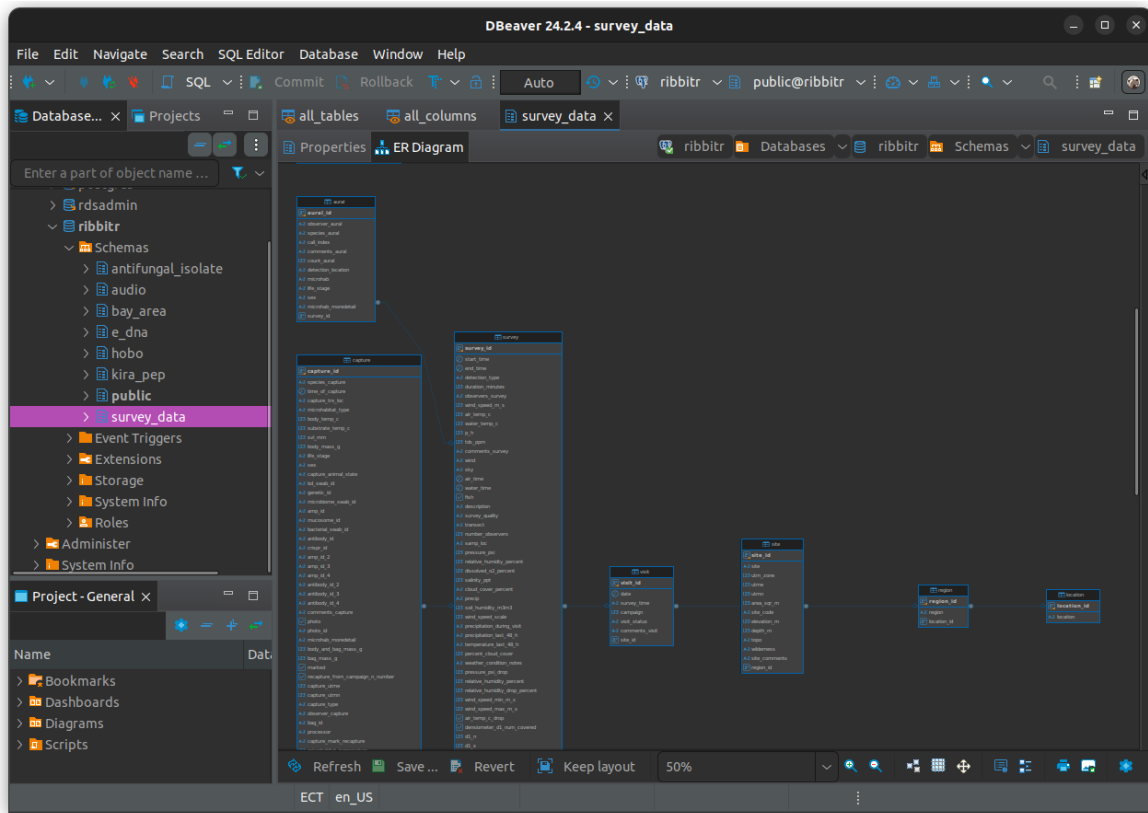
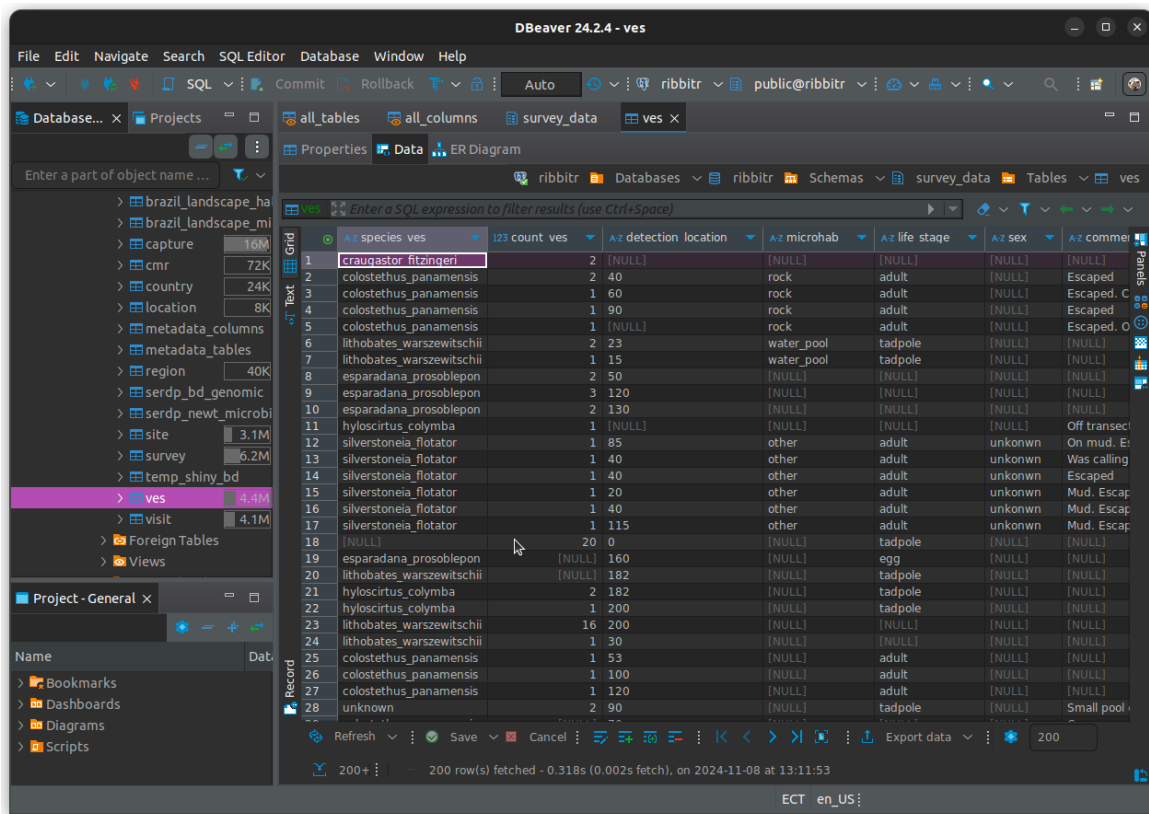


Figure 2: Navigate to Databases -> ribbitr -> Schemas -> survey_data. Right-click and select View Schema. Select the ER Diagram tab.

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 ->