3. Data Pulling

Cob Staines

2024-11-07

Table of contents

Motivation

- Download or "pull" data from the database to our local machine

- Pre- select and filter to data of interest
- Understand how different data tables relate to one another, and how to efficiently join them

R Data Pulling

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(species_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(species_ves,
         count_ves,
         life_stage,
         sex,
         survey_id) %>%
  filter(life_stage == "adult") %>%
  collect()

# preview table
head(db_ves_adult)
```

```
# A tibble: 6 x 5
  species_ves            count_ves life_stage sex     survey_id
  <chr>                      <int> <chr>      <chr>   <chr>
1 colostethus_panamensis         2 adult      <NA>    5ef5b5ea-dc62-4428-91d4-a~
2 colostethus_panamensis         1 adult      <NA>    5ef5b5ea-dc62-4428-91d4-a~
3 colostethus_panamensis         1 adult      <NA>    5ef5b5ea-dc62-4428-91d4-a~
4 colostethus_panamensis         1 adult      <NA>    5ef5b5ea-dc62-4428-91d4-a~
```

```
5 silverstoneia_flotator          1 adult      unkonwn fd70bfc9-693e-4a31-af06-5~
6 silverstoneia_flotator          1 adult      unkonwn fd70bfc9-693e-4a31-af06-5~
```

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(species_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 "species_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)
```

```
             species_ves count_ves life_stage     sex
1 colostethus_panamensis         2      adult    <NA>
2 colostethus_panamensis         1      adult    <NA>
3 colostethus_panamensis         1      adult    <NA>
4 colostethus_panamensis         1      adult    <NA>
5 silverstoneia_flotator         1      adult unkonwn
6 silverstoneia_flotator         1      adult unkonwn
                              survey_id
1 5ef5b5ea-dc62-4428-91d4-a55ba965ed47
2 5ef5b5ea-dc62-4428-91d4-a55ba965ed47
3 5ef5b5ea-dc62-4428-91d4-a55ba965ed47
4 5ef5b5ea-dc62-4428-91d4-a55ba965ed47
5 fd70bfc9-693e-4a31-af06-58b0c2b3ca1c
6 fd70bfc9-693e-4a31-af06-58b0c2b3ca1c
```

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.

Joins

"This is all good and well, but I only want data from Brazil… and there is no location information in this table! How do I connect with and filter by location?"

Recall that our database is not just a bunch of tables, it is a bunch of tables *with relationships*. For example, we can see that our VES table (`db_ves_adult`) has a column named `survey_id`. Taking a closer look at the column metadata (`mdc`), the `survey` table *also* has a column named `survey_id`. This common column is **key** to connecting our data between tables.

Understanding Keys

The concept of key columns or "keys" in database tables is used to help organize and communicate the relationships we want to establish between tables. There are several types of keys, here we will introduce 3:

- **Primary Key** *(pk or pkey)* – a column which is a unique identifier for each record (row) in a database table, ensuring that each record can be uniquely distinguished from all others in the table. Ideally a single column, often an ID
- **Natural Key** *(nk or nkey)* – A meaningful column (or set of columns) in a table which "naturally" and uniquely constrains all other columns. Often multiple columns, used to collectively define an ID column to be used as a primary key.
- **Foreign Key** *(fk or fkey)* – a column in one table which refers to the primary key in another table, establishing an asymmetric relationship between the two tables.

When we pull a data table from the database, it is often not so obvious which columns are or could be key columns, and which type of key. Luckily, we have of column metadata to help us keep track of this! Check out the `key_type` and `natural_key` columns for the survey table:

```
ves_metadata = mdc %>%
  select(table_name,
         column_name,
         key_type,
         natural_key) %>%
  filter(table_name == "ves")

view(ves_metadata)
```

We can see here that `ves_id` is the primary key (ie. unique, non-null row identifier) for the ves table (`ves_id` is also the only natural key for this table). We also see that column `survey_id` is a foreign key, meaning it points to the primary key of another table. Good investigation work, but this is tedious. Is there not a better way?

```
## ribbitrrr key functions

# primary key for ves table
tbl_pkey("ves", mdc)
```

```
[1] "ves_id"
```

```
# natural key for ves table
tbl_nkey("ves", mdc)
```

```
[1] "ves_id"
```

```
# foreign key for ves table
tbl_fkey("ves", mdc)
```

```
[1] "survey_id"
```

```
# all unique key columns for ves table
tbl_keys("ves", mdc)
```

```
[1] "ves_id"    "survey_id"
```

Notice that we passed the column metadata to these functions, to help us automate this otherwise tedious task.

Joining manually by keys

We can use these key columns, and what we know about the structure of our database, to join related tables. For example:

```
# lazy table of ves
db_ves = tbl(dbcon, Id("survey_data", "ves"))

# lazy table of survey
db_survey = tbl(dbcon, Id("survey_data", "survey"))

db_ves_survey = db_ves %>%
  left_join(db_survey, by="survey_id")

# check columns
colnames(db_ves_survey)
```

```
 [1] "species_ves"                    "count_ves"
 [3] "detection_location"             "microhab"
 [5] "life_stage"                     "sex"
 [7] "comments_ves"                   "microhab_moredetail"
 [9] "observer_ves"                   "visual_animal_state"
[11] "ves_id"                         "survey_id"
[13] "start_time"                     "end_time"
[15] "detection_type"                 "duration_minutes"
[17] "observers_survey"               "wind_speed_m_s"
[19] "air_temp_c"                     "water_temp_c"
[21] "p_h"                            "tds_ppm"
[23] "comments_survey"                "wind"
[25] "sky"                            "air_time"
[27] "water_time"                     "fish"
[29] "description"                    "survey_quality"
[31] "transect"                       "number_observers"
[33] "samp_loc"                       "pressure_psi"
[35] "relative_humidty_percent"       "dissolved_o2_percent"
[37] "salinity_ppt"                   "cloud_cover_percent"
[39] "precip"                         "soil_humidity_m3m3"
[41] "wind_speed_scale"               "precipitation_during_visit"
[43] "precipitation_last_48_h"        "temperature_last_48_h"
[45] "percent_cloud_cover"            "weather_condition_notes"
[47] "pressure_psi_drop"              "relative_humidity_percent"
[49] "relative_humidity_drop_percent" "wind_speed_min_m_s"
[51] "wind_speed_max_m_s"             "air_temp_c_drop"
[53] "densiometer_d1_num_covered"     "d1_n"
```

```
[55] "d1_s"                          "d1_e"
[57] "d1_w"                          "d1_percent_cover"
[59] "densiometer_d2_num_covered"    "d2_n"
[61] "d2_s"                          "d2_e"
[63] "d2_w"                          "d2_percent_cover"
[65] "depth_of_water_from_d2_cm"     "percent_vegetation_cover"
[67] "vegetation_notes"              "secchi_depth_cm"
[69] "visit_id"
```

We see that the columns of `db_ves_survey` correspond to the union of those from both the ves and survey tables. More importantly, the rows are lined up using `survey_id` as the key "join by" column. You could also substitute this with `by = tbl_fkey("ves", mdc)`.

In order to join the VES data with location, we will have to do several, recursive joins to connect the tables of... *(consults schema diagram)*... survey, visit, site, region, and location! Is there not a better way?

Links, Chains, Automated Joins

We developed some functions to help us avoid the tedium of consulting the database schema diagram or column metadata. The workflow for linking tables one at a time works like this:

```
# create a link object for table ves: "which tables are 1 step away"
link_ves = tbl_link("ves", mdc)

# join tables in link object
db_ves_survey = tbl_join(dbcon, link_ves, columns = "all")
```

```
Pulling ves ... done.
Joining with survey ... done.
```

```
# check columns
colnames(db_ves_survey)
```

```
 [1] "species_ves"                "count_ves"
 [3] "detection_location"         "microhab"
 [5] "life_stage"                 "sex"
 [7] "comments_ves"               "microhab_moredetail"
 [9] "observer_ves"               "visual_animal_state"
[11] "ves_id"                     "survey_id"
[13] "start_time"                 "end_time"
[15] "detection_type"             "duration_minutes"
```

```
[17] "observers_survey"                "wind_speed_m_s"
[19] "air_temp_c"                      "water_temp_c"
[21] "p_h"                             "tds_ppm"
[23] "comments_survey"                 "wind"
[25] "sky"                             "air_time"
[27] "water_time"                      "fish"
[29] "description"                     "survey_quality"
[31] "transect"                        "number_observers"
[33] "samp_loc"                        "pressure_psi"
[35] "relative_humidty_percent"        "dissolved_o2_percent"
[37] "salinity_ppt"                    "cloud_cover_percent"
[39] "precip"                          "soil_humidity_m3m3"
[41] "wind_speed_scale"                "precipitation_during_visit"
[43] "precipitation_last_48_h"         "temperature_last_48_h"
[45] "percent_cloud_cover"             "weather_condition_notes"
[47] "pressure_psi_drop"               "relative_humidity_percent"
[49] "relative_humidity_drop_percent" "wind_speed_min_m_s"
[51] "wind_speed_max_m_s"              "air_temp_c_drop"
[53] "densiometer_d1_num_covered"      "d1_n"
[55] "d1_s"                            "d1_e"
[57] "d1_w"                            "d1_percent_cover"
[59] "densiometer_d2_num_covered"      "d2_n"
[61] "d2_s"                            "d2_e"
[63] "d2_w"                            "d2_percent_cover"
[65] "depth_of_water_from_d2_cm"       "percent_vegetation_cover"
[67] "vegetation_notes"                "secchi_depth_cm"
[69] "visit_id"
```

Great, similar results to our previous manual join, but do I need to do this recursively to get to the location table? Is there not a better way?

The workflow for linking tables recursively works like this:

```
# create a chain (or recusive link) object for table ves: "which tables are any number of st
chain_ves = tbl_chain("ves", mdc)

# join tables in link object
db_ves_survey = tbl_join(dbcon, chain_ves, columns = "all")
```

```
Pulling ves ... done.
Joining with survey ... done.
Joining with visit ... done.
Joining with site ... done.
Joining with region ... done.
Joining with location ... done.
```

```r
# check columns
colnames(db_ves_survey)
```

```
 [1] "species_ves"                    "count_ves"
 [3] "detection_location"             "microhab"
 [5] "life_stage"                     "sex"
 [7] "comments_ves"                   "microhab_moredetail"
 [9] "observer_ves"                   "visual_animal_state"
[11] "ves_id"                         "survey_id"
[13] "start_time"                     "end_time"
[15] "detection_type"                 "duration_minutes"
[17] "observers_survey"               "wind_speed_m_s"
[19] "air_temp_c"                     "water_temp_c"
[21] "p_h"                            "tds_ppm"
[23] "comments_survey"                "wind"
[25] "sky"                            "air_time"
[27] "water_time"                     "fish"
[29] "description"                    "survey_quality"
[31] "transect"                       "number_observers"
[33] "samp_loc"                       "pressure_psi"
[35] "relative_humidty_percent"       "dissolved_o2_percent"
[37] "salinity_ppt"                   "cloud_cover_percent"
[39] "precip"                         "soil_humidity_m3m3"
[41] "wind_speed_scale"               "precipitation_during_visit"
[43] "precipitation_last_48_h"        "temperature_last_48_h"
[45] "percent_cloud_cover"            "weather_condition_notes"
[47] "pressure_psi_drop"              "relative_humidity_percent"
[49] "relative_humidity_drop_percent" "wind_speed_min_m_s"
[51] "wind_speed_max_m_s"             "air_temp_c_drop"
[53] "densiometer_d1_num_covered"     "d1_n"
[55] "d1_s"                           "d1_e"
[57] "d1_w"                           "d1_percent_cover"
[59] "densiometer_d2_num_covered"     "d2_n"
[61] "d2_s"                           "d2_e"
[63] "d2_w"                           "d2_percent_cover"
[65] "depth_of_water_from_d2_cm"      "percent_vegetation_cover"
[67] "vegetation_notes"               "secchi_depth_cm"
[69] "visit_id"                       "date"
[71] "survey_time"                    "campaign"
[73] "visit_status"                   "comments_visit"
[75] "site_id"                        "site"
[77] "utm_zone"                       "utme"
[79] "utmn"                           "area_sqr_m"
[81] "site_code"                      "elevation_m"
```

```
[83] "depth_m"                          "topo"
[85] "wilderness"                       "site_comments"
[87] "region_id"                        "region"
[89] "location_id"                      "location"
```

Hooray! Also yikes, that's a lot of columns! I am starting to see why we store all these in seperate tables!

Let's use the chain workflow to join only the data we want, to filter to Brazil data.

```
# lazy table, select, filter
db_ves_adult = tbl(dbcon, Id("survey_data", "ves")) %>%
  select(species_ves,
         count_ves,
         life_stage,
         sex,
         survey_id) %>%
  filter(life_stage == "adult")

# create chain object
chain_ves = tbl_chain("ves", mdc)

# join recursively, specifying desired columns, filter, collect
db_ves_adult_final = tbl_join(dbcon, chain_ves, tbl = db_ves_adult, columns = c("location")) %>%
  filter(location == "brazil")
```

```
Joining with survey ... done.
Joining with visit ... done.
Joining with site ... done.
Joining with region ... done.
Joining with location ... done.
```

```
# pull selected, filtered, joined data
data_ves_adult_final = db_ves_adult_final %>%
  collect()
```

A few differences here:

- We provided our pre-selected and filtered dv_ves_adult table to the join function with `tbl = db_ves_adult`, rather than having it pull all the data.
- We specified any additional columns to include with `columns = c("location")`, in addition to any key columns (included by default). The result is much less columns, only those we want.

Also try:

- Run `?tbl_chain` and `?tbl_join` to Learn more about other possible parameters to pass to these functions
- Check out the SQL code for your last query with:

```
sql_render(db_ves_adult_final)
```

Python Data Pulling

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

Load packages

For the second half of this tutorial, you will need to download the db_access.py script and save it alongside your `dbconfig.py` file. This script will provide us with some useful functions for easily pulling data.

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

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(['species_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([
    'species_ves',
    'count_ves',
    'life_stage',
    'sex',
    'survey_id'
  ])
  .to_pandas()
  )

db_ves_select.columns
```

```
Index(['species_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([
    'species_ves',
    'count_ves',
    'life_stage',
    'sex',
    'survey_id'
  ])
  .filter(_.life_stage == 'adult')
  .to_pandas()
  )

# preview table
db_ves_adult.head()
```

```
              species_ves  ...                              survey_id
0  colostethus_panamensis  ...  5ef5b5ea-dc62-4428-91d4-a55ba965ed47
1  colostethus_panamensis  ...  5ef5b5ea-dc62-4428-91d4-a55ba965ed47
2  colostethus_panamensis  ...  5ef5b5ea-dc62-4428-91d4-a55ba965ed47
3  colostethus_panamensis  ...  5ef5b5ea-dc62-4428-91d4-a55ba965ed47
4  silverstoneia_flotator  ...  fd70bfc9-693e-4a31-af06-58b0c2b3ca1c

[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([
    'species_ves',
    'count_ves',
    'life_stage',
    'sex',
    'survey_id'
  ])
  .filter(_.life_stage == 'adult')
  )

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

```
'SELECT "t0"."species_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.

Joins

"This is all good and well, but I only want data from Brazil… and there is no location information in this table! How do I connect with and filter by location?"

Recall that our database is not just a bunch of tables, it is a bunch of tables *with relationships*. For example, we can see that our VES table (`db_ves_adult`) has a column named `survey_id`. Taking a closer look at the column metadata (`mdc`), the `survey` table *also* has a column named `survey_id`. This common column is **key** to connecting our data between tables.

Understanding Keys

The concept of key columns or "keys" in database tables is used to help organize and communicate the relationships we want to establish between tables. There are several types of keys, here we will introduce 3:

- **Primary Key** *(pk or pkey)* – a column which is a unique identifier for each record (row) in a database table, ensuring that each record can be uniquely distinguished from all others in the table. Ideally a single column, often an ID

- **Natural Key** *(nk or nkey)* – A meaningful column (or set of columns) in a table which "naturally" and uniquely constrains all other columns. Often multiple columns, used to collectively define an ID column to be used as a primary key.
- **Foreign Key** *(fk or fkey)* – a column in one table which refers to the primary key in another table, establishing an asymmetric relationship between the two tables.

When we pull a data table from the database, it is often not so obvious which columns are or could be key columns, and which type of key. Luckily, we have of column metadata to help us keep track of this! Check out the `key_type` and `natural_key` columns for the survey table:

```
ves_metadata = mdc[(mdc['table_name'] == 'ves')][['table_name', 'column_name', 'key_type', '

print(ves_metadata)
```

```
    table_name          column_name key_type natural_key
98         ves               ves_id       PK        True
124        ves  visual_animal_state     None       False
326        ves         comments_ves     None       False
331        ves            count_ves     None       False
332        ves   detection_location     None       False
333        ves           life_stage     None       False
334        ves             microhab     None       False
335        ves  microhab_moredetail     None       False
336        ves         observer_ves     None       False
337        ves                  sex     None       False
338        ves          species_ves     None       False
339        ves            survey_id       FK       False
```

We can see here that `ves_id` is the primary key (ie. unique, non-null row identifier) for the ves table (`ves_id` is also the only natural key for this table). We also see that column `survey_id` is a foreign key, meaning it points to the primary key of another table. Good investigation work, but this is tedious. Is there not a better way?

```
## data_access key functions

# primary key for ves table
db.tbl_pkey("ves", mdc)
```

```
['ves_id']
```

```
# natural key for ves table
db.tbl_nkey("ves", mdc)
```

```
['ves_id']
```

```
# foreign key for ves table
db.tbl_fkey("ves", mdc)
```

```
['survey_id']
```

```
# all unique key columns for ves table
db.tbl_keys("ves", mdc)
```

```
['ves_id', 'survey_id']
```

Notice that we passed the column metadata to these functions, to help us automate this otherwise tedious task.

Joining manually by keys

We can use these key columns, and what we know about the structure of our database, to join related tables. For example:

```
# ves lazy table
db_ves = dbcon.table(database="survey_data", name="ves")
# survey lazy table
db_survey = dbcon.table(database="survey_data", name="survey")
# joined lazy table
db_ves_survey = db_ves.left_join(db_survey, "survey_id")

# check columns
db_ves_survey.columns
```

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

We see that the columns of `db_ves_survey` correspond to the union of those from both the ves and survey tables. More importantly, the rows are lined up using `survey_id` as the key "join by" column. You could also substitute this with `tbl_fkey("ves", mdc)`.

In order to join the VES data with location, we will have to do several, recursive joins to connect the tables of… *(consults schema diagram)*… survey, visit, site, region, and location! Is there not a better way?

Links, Chains, Automated Joins

We developed some functions to help us avoid the tedium of consulting the database schema diagram or column metadata. The workflow for linking tables one at a time works like this:

```
# create a link object for table ves: "which tables are 1 step away"
link_ves = db.tbl_link("ves", mdc)

# join tables in link object
db_ves_survey = db.tbl_join(dbcon, link_ves, columns="all")
```

```
Pulling ves ... done.
Joining with survey ... done.
```

```
# check columns
db_ves_survey.columns
```

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

Great, similar results to our previous manual join, but do I need to do this recursively to get to the location table? Is there not a better way?

The workflow for linking tables recursively works like this:

```
# create a chain (or recusive link) object for table ves: "which tables are any number of st
chain_ves = db.tbl_chain("ves", mdc)

# join tables in link object
db_ves_survey = db.tbl_join(dbcon, chain_ves, columns = "all")
```

```
Pulling ves ... done.
Joining with survey ... done.
Joining with visit ... done.
Joining with site ... done.
Joining with region ... done.
Joining with location ... done.
```

```
# check columns
db_ves_survey.columns
```

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

Hooray! Also yikes, that's a lot of columns! I am starting to see why we store all these in seperate tables!

Let's use the chain workflow to join only the data we want, to filter to Brazil data.

```
# lazy table, select, filter
db_ves_adult = (
  dbcon.table(database="survey_data", name="ves")
  .select([
    'species_ves',
    'count_ves',
    'life_stage',
    'sex',
    'survey_id'
    ])
  .filter(_.life_stage == 'adult')
  )

# create chain object
chain_ves = db.tbl_chain("ves", mdc)

# join recursively, providing selected and filtered table
db_ves_adult_final = (
  db.tbl_join(dbcon, chain_ves, tbl=db_ves_adult)
  .filter(_.location == 'brazil')
  )
```

```
Joining with survey ... done.
Joining with visit ... done.
Joining with site ... done.
Joining with region ... done.
Joining with location ... done.
```

```
# pull selected, filtered, joined data
data_ves_adult_final = db_ves_adult_final.to_pandas()
```

A few differences here:

- We provided our pre-selected and filtered `dv_ves_adult` table to the join function with `tbl = db_ves_adult`, rather than having it pull all the data.
- We specified any additional columns to include with `columns = c("location")`, in addition to any key columns (included by default). The result is much less columns, only those we want.

Also try:

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

```
db_ves_adult_final.compile()
```

Previous Tutorial: Data Discovery | Next Tutorial: Data Workflow