4. Table Joins

Cob Staines

2024-12-17

Table of contents

This tutorial is available as a .qmd on Github.

Motivation

- Understand how different data tables relate to one another
- Practice joining tables across foreign key columns

R

Let's set up our environment to get ready to join tables.

Setup

These setup steps will all be familiar to you by now.

```
# 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")) %>%
  filter(table_schema == "survey_data") %>%
  collect()

# load column metadata
mdc <- tbl(dbcon, Id("survey_data", "metadata_columns")) %>%
  filter(table_schema == "survey_data") %>%
  collect()
```

The need for joins

Let's begin with the query we constructed in our last tutorial:

```
# 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()
```

"This is all good and well, but I only want data from Brazil... and there is no country information in this table! How do I connect with and filter by country? or what if I want data from 2022 only?"

So far we have practiced pulling data from individual tables, but often we find that the data we need is stored accross multiple tables? We have to "join" these tables to bring the data we need together.

Table join basics

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
- **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.
- **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. Maintained in metadata as important columns, not always tracked in database relationships.

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

```
ves_metadata <- mdc %>%
  select(table_name,
         column_name,
         primary_key,
         foreign_key,
         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 another way?

```
## ribbitrrr key functions

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

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

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

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

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

```
[1] "ves_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 key lookups.

Joining tables by keys

We can use these key columns, and what we know about the structure of our database, to join related tables. For example, let's join the `ves` and `survey` tables along the `ves` foreign key of `survey_id`:

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

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

left_ves_survey <- db_ves %>%
  left_join(db_survey, by="survey_id")

# check columns
colnames(left_ves_survey)
```

```
 [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"
[13] "start_time"          "end_time"            "detection_type"
[16] "duration_minutes"    "observers_survey"    "comments_survey"
[19] "description"         "survey_quality"      "transect"
[22] "number_observers"    "visit_id"            "start_timestamp"
[25] "end_timestamp"
```

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)`.

When we join two tables along a key column, we are asking to align rows in the two tables where this key column is equal. In this example, we are returning a wider table where rows from the VES table are aligned with rows from the survey table where `survey_id` is equal in both.

Let's take a closer look at the number of rows in our tables to understand better what is going on.

```
ves_rows <- db_ves %>%
  summarise(row_count = n()) %>%
  pull(row_count)

(ves_rows)
```

```
integer64
[1] 29507
```

```
survey_rows <- db_survey %>%
  summarise(row_count = n()) %>%
  pull(row_count)

(survey_rows)
```

```
integer64
[1] 39477
```

```
left_ves_survey_rows <- left_ves_survey %>%
  summarise(row_count = n()) %>%
  pull(row_count)

(left_ves_survey_rows)
```

```
integer64
[1] 29507
```

Notice that in this case, the number of rows in the joined `db_ves_survey` table is equal to the number ofrows in the `db_ves` table. This has to do with the type of join we used to join these tables, in this case a "left join".

Types of joins

Joins unite rows from two tables along shared key column values, allowing unshared columns between the two tables to be brought into a single table (as we see above in `db_ves_survey`). However,here are different joins to handle key columns vales that are not shared between tables (ie. are unique to one table). The following graphic helps illustrate the different join types:

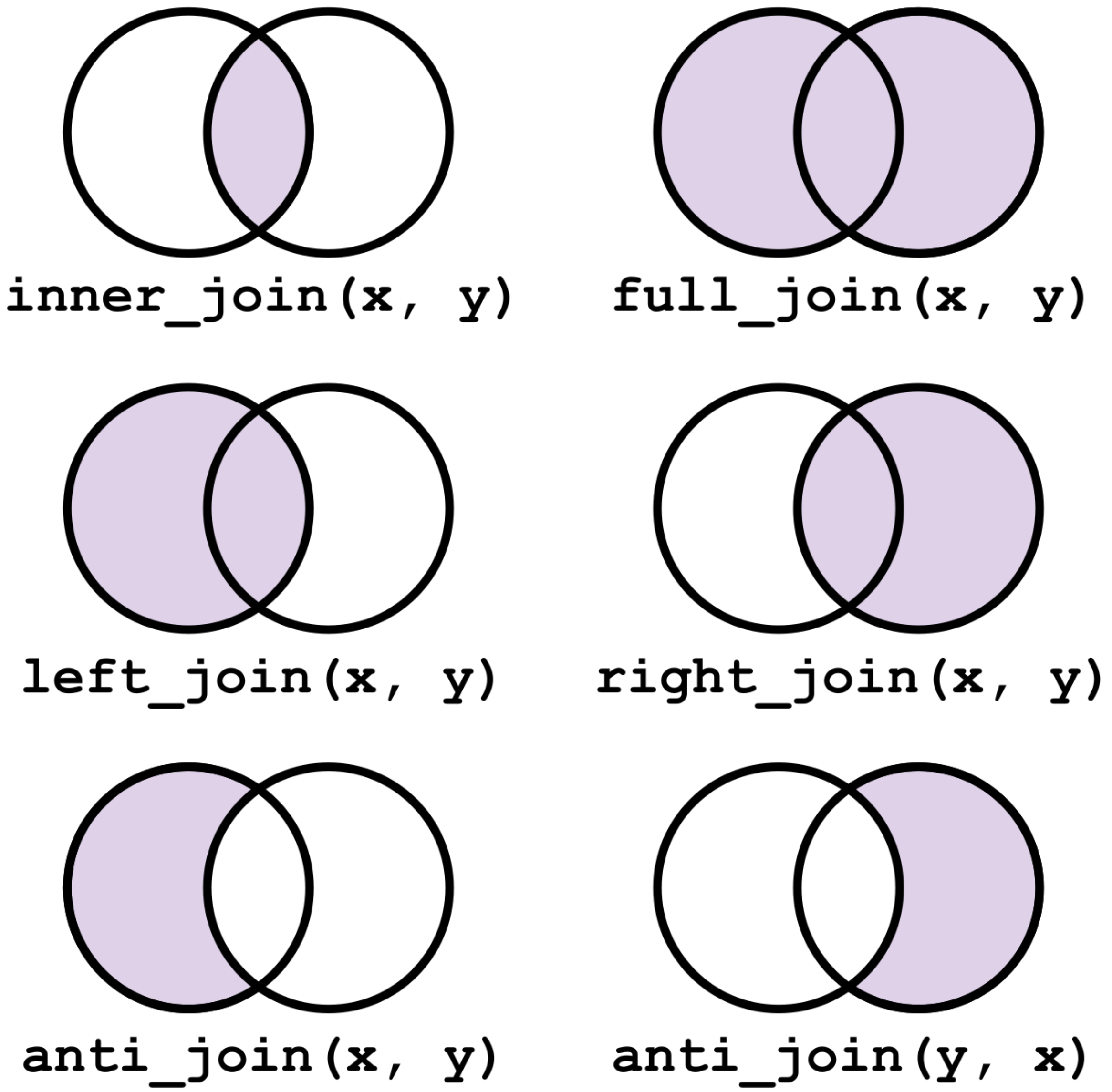

Figure 1: Types of dplyr joins. Graphic by Software Carpentry used under Creative Commons 4.0

The graphic above describes the treatment of key column values which are either unique to one table or shared in both, using Venn diagram. In our `db_ves_survey` example, our use of `left_join()` essentially says "return everything found in the 1st ("left") table (ie. `db_ves`) regardless of if it has a match in the 2nd ("right") table (ie. `db_survey`). This explains why the output row numbers are identical to those in `db_ves`.

Using this diagram, let's try a different scenario. Let's instead use a `full_join()` to return all rows from each table, regardless of whether they have a matching `survey_id` in the other.

```r
full_ves_survey <- db_ves %>%
  full_join(db_survey, by="survey_id")

full_ves_survey_rows <- full_ves_survey %>%
  summarise(row_count = n()) %>%
  pull(row_count)

(full_ves_survey_rows)
```

```
integer64
[1] 59034
```

In this case we see a greater number of rows returned than are present in either table (though less than the sum of rows in each table). This tells us that some rows did align between tables, while others did not. Specifically, `full_ves_survey_rows` returns all VES observations (which have a corresponding `survey_id` in the `survey` table by design, see note below) *as well as* all other surveys in the `survey` table, even those that do not correspond to a VES survey. We can see this quickly by comparing the distinct `detection_type`'s seen in each joined table:

```r
left_ves_survey %>%
  select(detection_type) %>%
  distinct()
```

```
# Source:   SQL [1 x 1]
# Database: postgres  [cob_reads@ribbitr.c6p56tuocn5n.us-west-1.rds.amazonaws.com:5432/ribbi
  detection_type
  <chr>
1 visual
```

```r
full_ves_survey %>%
  select(detection_type) %>%
  distinct()
```

```
# Source:   SQL [5 x 1]
# Database: postgres  [cob_reads@ribbitr.c6p56tuocn5n.us-west-1.rds.amazonaws.com:5432/ribbi
  detection_type
  <chr>
1 capture
2 environmental
3 aural
4 other
5 visual
```

Note: In this case, the `survey` table is intentionally designed such that all `survey_id`'s in the `ves` table. To confirm this we can try:

```r
anti_ves_survey <- db_ves %>%
  anti_join(db_survey, by="survey_id")

anti_ves_survey_rows <- anti_ves_survey %>%
  summarise(row_count = n()) %>%
  pull(row_count)

(anti_ves_survey_rows)
```

```
integer64
[1] 0
```

There are 0 rows in this anti_join. This means that in our Venn diagram, the left lobe is essentially empty or nonexistent. The outcome is that in *this case*, `left_join()` and `inner_join()` are equivalent, as are `full_join()` and `right_join()`. This is not true in general, however.

Relationship cardinality

Foreign keys which point to primary keys are an example of a many:1 ("many-to-one") relationship This is to say that a foreign key column can hold any number of duplicate values, while a primary key value can show up a maximum of once. Many:1 relationships are most common in our database. Occasionally you may see a 1:1 relationship or a many:many relationship.

Deciding how to join tables

You have the liberty to decide which join you want, so how do you make that choice? Do you want - all the data from one table (`left_join()` or `right_join()`) - all the data or from both tables (`full_join`) - only data that is shared between the two (`inner_join()`)

To join tables properly along shared and key columns, it is important to understand the structure of the database. This is where consulting the column metadata (`mdc`) or schema diagram comes in handy.

Looking at the schema diagram, we see that to join the VES data with country, we will have to recursively join the survey, visit, site, region, and country tables.

Recursive joins

With that understanding, let's connect the data we have been wanting all along:

```r
# pointers for all tables
db_ves <- tbl(dbcon, Id("survey_data", "ves"))
db_survey <- tbl(dbcon, Id("survey_data", "survey"))
db_visit <- tbl(dbcon, Id("survey_data", "visit"))
db_site <- tbl(dbcon, Id("survey_data", "site"))
db_region <- tbl(dbcon, Id("survey_data", "region"))
db_country <- tbl(dbcon, Id("survey_data", "country"))

db_ves_sup <- db_ves %>%
  left_join(db_survey, by = "survey_id") %>%
  left_join(db_visit, by = "visit_id") %>%
  left_join(db_site, by = "site_id") %>%
  left_join(db_region, by = "region_id") %>%
  left_join(db_country, by = "country_id")
```

Finally we can select for and filter the variables of interest:

```r
data_ves_filtered <- db_ves_sup %>%
  select(species_ves,
         count_ves,
         life_stage,
         sex,
         survey_id,
         site,
         date,
         country) %>%
  filter(life_stage == "adult",
         year(date) == "2022",
         country == "brazil") %>%
  collect()
```

Links, Chains, Recursive Joins

We developed some additional functions in the `ribbitrrr` package 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_left_join(dbcon, link_ves)
```

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

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

```
 [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"
[13] "start_time"          "end_time"            "detection_type"
[16] "duration_minutes"    "observers_survey"    "comments_survey"
[19] "description"         "survey_quality"      "transect"
[22] "number_observers"    "visit_id"            "start_timestamp"
[25] "end_timestamp"
```

Great, similar results to our previous manual join, but do I need to do this recursively to get to the country table? Is there another 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_left_join(dbcon, chain_ves)
```

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

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

```
 [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"
[13] "start_time"          "end_time"            "detection_type"
[16] "duration_minutes"    "observers_survey"    "comments_survey"
[19] "description"         "survey_quality"      "transect"
[22] "number_observers"    "visit_id"            "start_timestamp"
[25] "end_timestamp"       "date"                "time_of_day"
[28] "campaign"            "visit_status"        "comments_visit"
[31] "site_id"             "site"                "utm_zone"
[34] "utme"                "utmn"                "area_sqr_m"
[37] "site_code"           "elevation_m"         "depth_m"
[40] "topo"                "wilderness"          "site_comments"
[43] "region_id"           "site_name_alt"       "region"
[46] "country_id"          "location_id"         "time_zone"
[49] "country"             "iso_country_code"
```

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

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

```r
# create chain object
chain_ves <- tbl_chain("ves", mdc)

# join recursively, specifying desired columns, filter, collect
db_ves_adult_final <- tbl_left_join(dbcon, chain_ves) %>%
    select(species_ves,
         count_ves,
         life_stage,
         sex,
         survey_id,
         site,
         date,
         country) %>%
  filter(life_stage == "adult",
         year(date) == "2022",
         country == "brazil")
```

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

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

Note that there are corresponding recursive join functions in `ribbitrrr` for:

- `tbl(_left_join()`
- `tbl(_inner_join()`
- `tbl(_full_join()`
- `tbl(_right_join()`

Disconnect

```
dbDisconnect(dbcon)
```

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

Let's set up our environment to get ready to join tables.

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 recursively joining tables

Setup

These setup steps will all be familiar to you by now.

```
# minimal packages for RIBBiTR DB Workflow
import pandas as pd
import ibis
from ibis import _
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")
  .filter(_.table_schema == 'survey_data')
  .to_pandas()
  )
```

The need for joins

Let's begin with the query we constructed in our last tutorial:

```
# lazy table select, filter, and collect all in one
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()
  )
```

"This is all good and well, but I only want data from Brazil… and there is no country information in this table! How do I connect with and filter by country? or what if I want data from 2022 only?"

So far we have practiced pulling data from individual tables, but often we find that the data we need is stored accross multiple tables? We have to "join" these tables to bring the data we need together.

Table join basics

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
- **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.
- **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. Maintained in metadata as important columns, not always tracked in database relationships.

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

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

print(ves_metadata)
```

```
    table_name          column_name  primary_key  foreign_key natural_key
61         ves               ves_id         True        False        True
78         ves  visual_animal_state        False        False       False
239        ves         comments_ves        False        False       False
241        ves            count_ves        False        False       False
242        ves   detection_location        False        False       False
243        ves           life_stage        False        False       False
244        ves             microhab        False        False       False
245        ves  microhab_moredetail        False        False       False
246        ves         observer_ves        False        False       False
247        ves                  sex        False        False       False
248        ves          species_ves        False        False       False
285        ves            survey_id        False         True       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 another way?

```
## data_access key functions

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

```
['ves_id']
```

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

```
['survey_id']
```

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

```
['ves_id']
```

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

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

Notice that we passed the column metadata to these functions, to help us automate key lookups.

Joining tables by keys

We can use these key columns, and what we know about the structure of our database, to join related tables. For example, let's join the `ves` and `survey` tables along the `ves` foreign key of `survey_id`:

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

# leff_join
left_ves_survey = db_ves.left_join(db_survey, "survey_id")

# check columns
left_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 `by = tbl_fkey("ves", mdc)`.

When we join two tables along a key column, we are asking to align rows in the two tables where this key column is equal. In this example, we are returning a wider table where rows from the VES table are aligned with rows from the survey table where `survey_id` is equal in both.

Let's take a closer look at the number of rows in our tables to understand better what is going on.

```
# Count rows in db_ves
print(db_ves.count().execute())
```

```
29507
```

```
# Count rows in db_survey
print(db_survey.count().execute())
```

```
39477
```

```
# Count rows in left_ves_survey
print(left_ves_survey.count().execute())
```

```
29507
```

Notice that in this case, the number of rows in the joined `db_ves_survey` table is equal to the number ofrows in the `db_ves` table. This has to do with the type of join we used to join these tables, in this case a "left join".

Types of joins

Joins unite rows from two tables along shared key column values, allowing unshared columns between the two tables to be brought into a single table (as we see above in `db_ves_survey`). However,here are different joins to handle key columns vales that are not shared between tables (ie. are unique to one table). The following graphic helps illustrate the different join types:

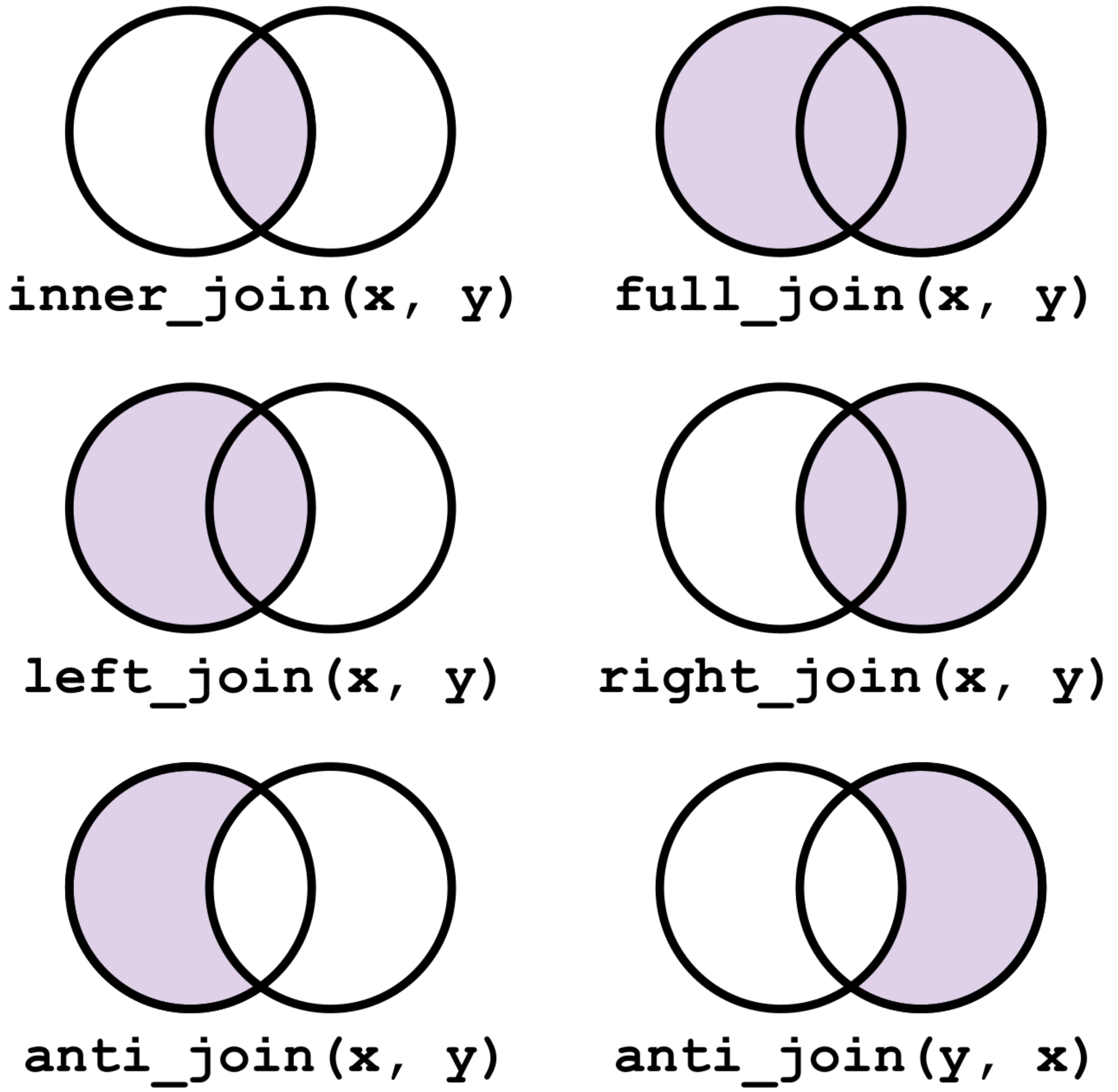

Figure 2: Types of dplyr joins. Graphic by Software Carpentry used under Creative Commons 4.0

The graphic above describes the treatment of key column values which are either unique to one table or shared in both, using Venn diagram. In our `db_ves_survey` example, our use of `.left_join()` essentially says "return everything found in the 1st ("left") table (ie. `db_ves`) regardless of if it has a match in the 2nd ("right") table (ie. `db_survey`). This explains why the output row numbers are identical to those in `db_ves`.

Using this diagram, let's try a different scenario. Let's instead use `.outer_join()` to return all rows from each table, regardless of whether they have a matching `survey_id` in the other.

```
# Full join of db_ves and db_survey
full_ves_survey = db_ves.outer_join(db_survey, db_ves.survey_id == db_survey.survey_id)

# count rows
print(full_ves_survey.count().execute())
```

```
59034
```

In this case we see a greater number of rows returned than are present in either table (though less than the sum of rows in each table). This tells us that some rows did align between tables, while others did not. Specifically, `full_ves_survey_rows` returns all VES observations (which have a corresponding `survey_id` in the `survey` table by design, see note below) *as well as* all other surveys in the `survey` table, even those that do not correspond to a VES survey. We can see this quickly by comparing the distinct `detection_type`'s seen in each joined table:

```
# Distinct detection_types in left_ves_survey
print(left_ves_survey.select(['detection_type']).distinct().execute())
```

```
  detection_type
0         visual
```

```
# Distinct detection_types in full_ves_survey
print(full_ves_survey.select(['detection_type']).distinct().execute())
```

```
  detection_type
0        capture
1  environmental
2          aural
3          other
4         visual
```

Note: In this case, the `survey` table is intentionally designed such that all `survey_id`'s in the `ves` table. To confirm this we can try:

```
# Anti-join of db_ves and db_survey
anti_ves_survey = db_ves.anti_join(db_survey, db_ves.survey_id == db_survey.survey_id)
anti_ves_survey_rows = anti_ves_survey.count().execute()
print(anti_ves_survey_rows)
```

```
0
```

There are 0 rows in this anti_join. This means that in our Venn diagram, the left lobe is essentially empty or nonexistent. The outcome is that in *this case*, `left_join()` and `inner_join()` are equivalent, as are `full_join()` and `right_join()`. This is not true in general, however.

Relationship cardinality

Foreign keys which point to primary keys are an example of a many:1 ("many-to-one") relationship This is to say that a foreign key column can hold any number of duplicate values, while a primary key value can show up a maximum of once. Many:1 relationships are most common in our database. Occasionally you may see a 1:1 relationship or a many:many relationship.

Deciding how to join tables

You have the liberty to decide which join you want, so how do you make that choice? Do you want - all the data from one table (`left_join()` or `right_join()`) - all the data or from both tables (`full_join`) - only data that is shared between the two (`inner_join()`)

To join tables properly along shared and key columns, it is important to understand the structure of the database. This is where consulting the column metadata (`mdc`) or schema diagram comes in handy.

Looking at the schema diagram, we see that to join the VES data with country, we will have to recursively join the survey, visit, site, region, and country tables.

Recursive joins

With that understanding, let's connect the data we have been wanting all along:

```
# Ponters for all tables
db_ves = dbcon.table('ves', database='survey_data')
db_survey = dbcon.table('survey', database='survey_data')
db_visit = dbcon.table('visit', database='survey_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')

# Recursive joins
db_ves_sup = (
    db_ves
    .left_join(db_survey, db_ves.survey_id == db_survey.survey_id)
    .left_join(db_visit, db_survey.visit_id == db_visit.visit_id)
    .left_join(db_site, db_visit.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)
)
```

Finally we can select for and filter the variables of interest:

```
data_ves_filtered = (
    db_ves_sup
    .select(
      'species_ves',
      'count_ves',
      'life_stage',
      'sex',
      'survey_id',
      'site',
      'date',
      'country')
    .filter([
        (_.life_stage == 'adult') &
        (_.date.year() == 2022) &
        (_.country == 'brazil')
    ])
    .to_pandas()
)
```

Links, Chains, Automated Joins

We developed some additional functions in the `ribbitrrr` package 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, join="left")
```

```
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 of `ves` and `survey`, but do I need to do this recursively to get to the country table? Is there another 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, join="left")
```

```
Pulling ves ... done.
Joining with survey ... done.
Joining with visit ... done.
Joining with site ... done.
Joining with region ... done.
Joining with country ... 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 separate tables!

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

```
# join recursively, specifying desired columns, filter, collect
db_ves_filtered = (
  db.tbl_join(dbcon, chain_ves, join="left")
  .select(
    'species_ves',
    'count_ves',
    'life_stage',
    'sex',
    'survey_id',
    'site',
    'date',
    'country')
    .filter([
      (_.life_stage == 'adult') &
      (_.date.year() == 2022) &
      (_.country == 'brazil')
      ])
)
```

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

```
data_ves_filtered = db_ves_filtered.to_pandas()
```

Note that you can specify the recursive join type using `tbl_join(..., join = "full")` where valid options are "left", "full", "inner", and "right"

Disconnect

```
dbcon.disconnect()
```

Also try:

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

```
db_ves_filtered.compile()
```

<- 3. Data Pulling | 5. Data Workflow ->