Relational Data with dplyr

Dr. Austin Brown

Kennesaw State University

Table of Contents

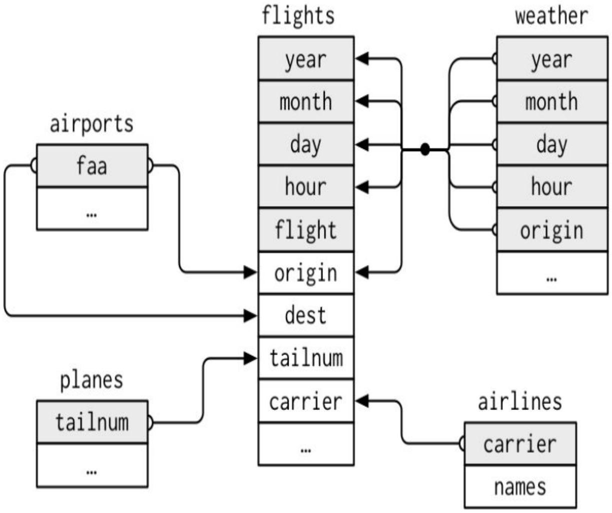
Introduction

- In most organizations, it is very rare that you will find all of the data you'll need to perform your job in a single data table.
- Most of the time, the data that we need in order to perform whatever analysis will be located in disparate, but related tables.
- Because of this, one of the key skills we need to know is how to merge or join tables together in order to have the correct data to answer the relevant research questions at hand.

Introduction

- In mostly all modern organizations, data is stored in what is referred to as a "Relational Database Management System (RDMS)."
- At a very basic level, an RDMS is a series of interrelated data tables which are connected by one or more columns called *keys*.
- A key is a unique identifier of a particular observation (think customer ID or Social Security Number).

Introduction



Introduction

- As we can maybe see in the previous figure, we have two different types of keys:

1. A primary key, which is a variable or set of variables which uniquely identify an observation *in its own table*.
    - Primary keys which require more than one column/variable are called **compound keys**.

- For example, in the RDBMS described before, `airlines$carrier` is the primary key in its table.

2. A foreign key is a variable (or set of variables) that corresponds to a primary key in a different table.

- Here, the `flights$tailnum` is a foreign key because it matches each flight to a unique airplane (i.e., `planes$tailnum` is the primary key). `flights$origin` and `flights$dest` are also examples of foreign keys.

Introduction

- How can we be confident our primary key really is uniquely identifying each observation in our dataframe?
- For example, `airports$faa` is a primary key.

```r
airports |>
  count(faa) |>
  filter(n > 1)
```

```
## # A tibble: 0 x 2
## # ... with 2 variables: faa <chr>, n <int>
## # i Use `colnames()` to see all variable names
```

Introduction

- On occasion, we may know that each row of a table is indeed a unique observation, but there may not exist a single column which contains a unique value for that row (i.e., no obvious primary key).
- In situations like this, we can create a pseudo-primary key or a *surrogate* key by simply creating a new column that is simply the row number. Another trick could be concatenating two columns which together create a unique value.

Introduction

```r
## Check: Are all the rows in Weather Unique? ##
weather |>
  distinct() |>
  count()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1 26115
```

```r
## Yep! Create Surrogate Key ##
weather <- weather |>
  mutate(ID = row_number(),
         .before=1)
```

Mutating Joins

- For those of you familiar with SQL, a mutating join is like a horizontal join or a type of `merge` for those familiar with the Base R function.
- We learned in last class that `mutate` creates new columns in our dataframe. Similarly, a mutating join will create a new dataframe from two existing dataframes, joining them row-wise by matching the keys.
- There are a few different mutating or horizontal joins we should be aware of that can help in various situations.

Mutating Joins

- Joins can be complicated, so to understand what each does, it can be helpful to use simple examples and visual aids to assist in our understanding. Let's looks at two simple dataframes:

```
## # A tibble: 3 x 2
##     key val_x
##   <dbl> <chr>
## 1     1 x1
## 2     2 x2
## 3     3 x3
```

```
## # A tibble: 3 x 2
##     key val_y
##   <dbl> <chr>
## 1     1 y1
## 2     2 y2
## 3     4 y3
```

Mutating Joins

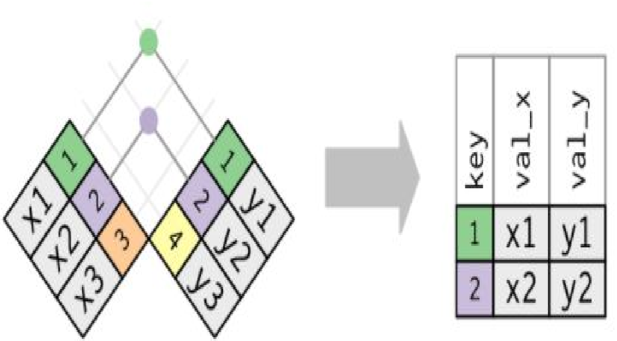
- One type of mutating join is called an “inner join.” When we perform an inner join, the resulting dataframe will only contain those rows whose keys match:

```r
x |>
  inner_join(y,by="key")
```

```
## # A tibble: 2 x 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
```

Mutating Joins

- Visually:



Mutating Joins

- As shown, an inner join only retains those rows whose keys match across both dataframes. In other words, there's a strong likelihood rows will be omitted from the outputted dataframe.

- Because of this property, it is more common to perform an outer join, of which there are three:
  1. `left_join`
  2. `right_join`
  3. `full_join`

Mutating Joins

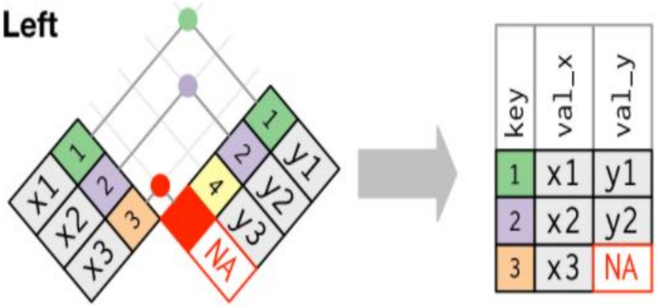
- Outer joins, unlike inner joins, will retain rows. But this depends on the type of outer join specified.

- A `left_join` will retain all of the rows in the first specified dataframe, and put `NA` values in the unmatched rows.

```
x |>
  left_join(y,by="key")
```

```
## # A tibble: 3 x 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     3 x3    <NA>
```

Mutating Joins



key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA

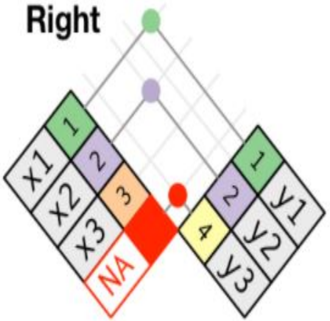
Mutating Joins

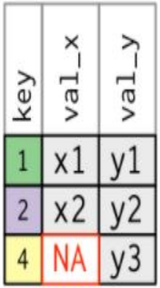
- A `right_join` does just the opposite. It will retain all the rows in the dataframe included in the `right_join` function.

```
x |>
  right_join(y,by="key")
```

```
## # A tibble: 3 x 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     4 <NA>  y3
```

Mutating Joins





key	val_x	val_y
1	x1	y1
2	x2	y2
4	NA	y3

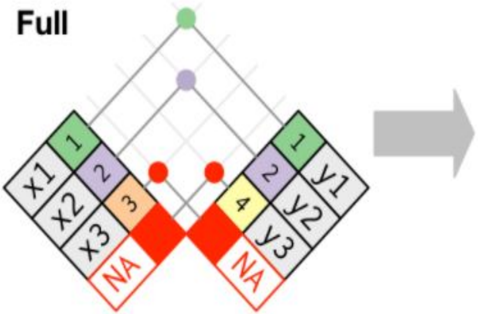
Mutating Joins

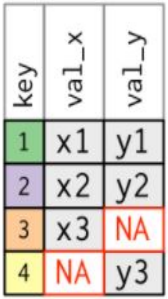
- A full_join retains the rows in both dataframes:

```
x |>
  full_join(y,by="key")
```

```
## # A tibble: 4 x 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     3 x3    <NA>
## 4     4 <NA>  y3
```

Mutating Joins





key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA
4	NA	y3

Mutating Joins

- All of the simple examples we've gone through to this point have assumed that each row in both of our dataframes have unique keys. Sometimes, this isn't the case.
- If we attempt a horizontal join where one or both of the dataframes have duplicate keys, the resulting table will contain all unique combinations of rows.
- On the following slides, one table has duplicate keys, but the other doesn't (i.e., a one-to-many relationship exists).

Mutating Joins

```
x
```

```
## # A tibble: 4 x 2
##     key val_x
##   <dbl> <chr>
## 1     1 x1
## 2     2 x2
## 3     2 x3
## 4     1 x4
```

```
y
```

```
## # A tibble: 2 x 2
##     key val_y
##   <dbl> <chr>
## 1     1 y1
## 2     2 y2
```

Mutating Joins

```r
left_join(x,y,by="key")
```

```
## # A tibble: 4 x 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     2 x3    y2
## 4     1 x4    y1
```

Mutating Joins

- Now, when both tables have duplicate keys, we will again get all unique combinations of rows (a Cartesian product).

```
x <- tribble(~key,~val_x,1,"x1",2,"x2",2,"x3",3,"x4")
y <- tribble(~key,~val_y,1,"y1",2,"y2",2,"y3",3,"y4")
```

Mutating Joins

```r
left_join(x,y,by="key")
```

```
## # A tibble: 6 x 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     2 x2    y3
## 4     2 x3    y2
## 5     2 x3    y3
## 6     3 x4    y4
```

Mutating Joins

- Let's look at a couple of more complicated examples. Let's say we want to append the full name of the destination airport to the full flights dataframe.
- Obviously, we want keep the full number of rows in the flights dataframe, but just want the full airport name to be an additional column.

Mutating Joins

- One cool feature about these mutating joins that are similar to PROC SQL/SQL is the ability to perform "natural joins."
- A natural join is one where the join is based on all columns with the same name in the two dataframes being joined.
- Let's say we want to join the weather data to the `flights2` dataframe we just created.

Mutating Joins

- One word of caution about natural joins:
- While natural joins can be handy when joining on several conditions, we have to make sure that thoses columns with the same names contain the same data.
- For example, if we have two columns both named “name,” this might not mean the same thing in both dataframes.
- Additionally, this is a good reminder to have clear naming conventions for variables.

Filtering Joins

- As we learned in last class, filtering is the action we perform upon the rows of a dataframe.
- Similarly, sometimes we need to join two tables vertically (i.e., adds rows/observations) instead of horizontally (i.e., adding columns).
- `dplyr` allows for these types of joins through the use of `semi_join` and `anti_join`.

Filtering Joins

- A `semi_join` will keep all the rows in the first specified dataframe that have a match in the second specified dataframe.
- So let's say we are interested in seeing what the top 10 destinations are for NYC flights and we want to see what flights went to those specific destinations.

Filtering Joins

- An anti_join is somewhat the opposite of a semi_join. Instead of including only those matching rows in both dataframes, it will omit those rows in the first specified dataframe which match in the second specified dataframe.

- A really handy application of this is in sentiment analysis (which is actually pretty easy to do using the tidyverse functions). If we go through, say consumer reviews, we only want to include those words which aren't filler words like "a, an, the, and, of" and so on, we can use an anti_join to remove those words from our list of other words.

- With the flights data, let's say we want to see how many flights don't have the airplane tailnumbers which match in the planes dataframe.

Set Operations

- The last type of joins are called “set operations” which you may be familiar with if you work with SQL.

- These operations are most useful in R when you have two dataframes with identical columns which need to be “stacked” (or “row binded” for you base R lovers) on top of each other, but perhaps with some conditions.

Set Operations

- There are three main set operations in the `dplyr` package:
- `intersect` only returns the rows which appear in both dataframes
- `union` returns <u>unique</u> rows in both dataframes
- `setdiff` returns only those observations appearing in the first specified dataframe, but not the second

Set Operations

- An example of an `intersect` operation:

```r
df1 <- tribble(~x,~y,1,1,2,1)
df2 <- tribble(~x,~y,1,1,1,2)
intersect(df1,df2)
```

```
## # A tibble: 1 x 2
##       x     y
##   <dbl> <dbl>
## 1     1     1
```

Set Operations

- An example of an `union` operation:

```r
union(df1,df2)
```

```
## # A tibble: 3 x 2
##         x     y
##     <dbl> <dbl>
## 1       1     1
## 2       2     1
## 3       1     2
```

Set Operations

- An example of a `setdiff` operation:

```r
setdiff(df1,df2)
```

```
## # A tibble: 1 x 2
##        x     y
##    <dbl> <dbl>
## 1      2     1
```

Final Notes

- Now, all of these operations we've talked about are useful in the various circumstances where they can be applied.
- However, there may be some instances when you need to just stack two dataframes on top of each other regardless if the rows match or now.
- You have a few options available to you for such an operation.

Final Notes

- union_all

```
union_all(df1,df2)
```

```
## # A tibble: 4 x 2
##       x     y
##   <dbl> <dbl>
## 1     1     1
## 2     2     1
## 3     1     1
## 4     1     2
```

Final Notes

- `bind_rows`

```
bind_rows(df1,df2)
```

```
## # A tibble: 4 x 2
##       x     y
##   <dbl> <dbl>
## 1     1     1
## 2     2     1
## 3     1     1
## 4     1     2
```

Final Notes

- rbind.data.frame

```
rbind.data.frame(df1,df2)
```

```
## # A tibble: 4 x 2
##       x     y
##   <dbl> <dbl>
## 1     1     1
## 2     2     1
## 3     1     1
## 4     1     2
```