Unir tablas

Jorge Loría

Sept 25, 2017

Unir tablas

Hay varias formas de hacer uniones de tablas. Este tipo de uniones se pueden dar cuando se tienen datos que mantienen ciertas tipos de relaciones en distintas tablas.

Datos

Vamos a ocupar las siguientes tablas para hacer los ejemplos de esta clase:

```
library(dplyr)
x1 <- data.frame(nombre = c('a','b','c','a'),
                 valor_x = c('x1','x2','x3','x4'),
                 stringsAsFactors = FALSE)
y1 <- data.frame(nombre = c('a','b','d','b'),
                 valor_y = c('y1','y2','y3','y5'),
                 stringsAsFactors = FALSE)
```

Más datos

```
y2 <- y1 %>%
  rename('Otro_nombre' = 'nombre')

x2 <- data.frame(Otro_nombre = c('a','b','c','a'),
                 Otro_valor_x = c('x1','x2','x3','x2'),
                 nuevo_valor = c(15.7,7.46,6.19,8.80),
                 stringsAsFactors = FALSE)

y3 <- data.frame(nombre = c('a', 'b','d','a'),
                 valor_y = c('3y1','3y2','3y3','3y4'),
                 stringsAsFactors = FALSE)
```

`*_join`

Hay 6 tipos de `join`, los cuales se parecen mucho entre sí. La gran diferencia es a *cual* de las tablas se le va a dar prioridad para conservar observaciones. Lo que se busca hacer con los `join` es obtener información que se tiene en otra tabla para poder realizar operaciones con variables de ambas tablas.

*_join

Hay 6 tipos de `join`, los cuales se parecen mucho entre sí. La gran diferencia es a *cual* de las tablas se le va a dar prioridad para conservar observaciones. Lo que se busca hacer con los `join` es obtener información que se tiene en otra tabla para poder realizar operaciones con variables de ambas tablas.

*_join	Prioridad
left	primer argumento
right	segundo argumento
full	ambos argumentos
inner	ambas con coincidencias
semi	primer argumento con coincidencias
anti	primer argumento sin coincidencias

`*_join`

Hay 6 tipos de `join`, los cuales se parecen mucho entre sí. La gran diferencia es a *cual* de las tablas se le va a dar prioridad para conservar observaciones. Lo que se busca hacer con los `join` es obtener información que se tiene en otra tabla para poder realizar operaciones con variables de ambas tablas.

`*_join`	Prioridad
left	primer argumento
right	segundo argumento
full	ambos argumentos
inner	ambas con coincidencias
semi	primer argumento con coincidencias
anti	primer argumento sin coincidencias

Los primeros cuatro son para aumentar la cantidad de variables (como `mutate`). Los otros dos, son para eliminar observaciones (como `filter`).

left_join

Esta función busca en las columnas del primer argumento (`x1`) que se indiquen (con el parámetro `by = 'nombre'`) cuales observaciones tienen coincidencias en la segunda tabla (`y1`) en la misma columna.

left_join

Esta función busca en las columnas del primer argumento (`x1`) que se indiquen (con el parámetro `by = 'nombre'`) cuales observaciones tienen coincidencias en la segunda tabla (`y1`) en la misma columna.

```
left_join(x1,y1,by = 'nombre')
```

```
##   nombre valor_x valor_y
## 1      a      x1      y1
## 2      b      x2      y2
## 3      b      x2      y5
## 4      c      x3    <NA>
## 5      a      x4      y1
```

NA

NA

En general, muchas veces **no** hay datos para todas las observaciones, por lo que se puede poner `NA` para estos casos. Que representa “not available”, o sea: “no disponible”.

NA

En general, muchas veces **no** hay datos para todas las observaciones, por lo que se puede poner `NA` para estos casos. Que representa "not available", o sea: "no disponible". Dependiendo de las circunstancias y la naturaleza de los datos estos se pueden tratar de varias formas, muchas de las funciones base tiene el parámetro `na.rm` que viene pre definido como `FALSE`, si se le llama con `na.rm = FALSE` se omite ese valor a la hora de realizar el llamado.

NA

En general, muchas veces **no** hay datos para todas las observaciones, por lo que se puede poner `NA` para estos casos. Que representa "not available", o sea: "no disponible". Dependiendo de las circunstancias y la naturaleza de los datos estos se pueden tratar de varias formas, muchas de las funciones base tiene el parámetro `na.rm` que viene pre definido como `FALSE`, si se le llama con `na.rm = FALSE` se omite ese valor a la hora de realizar el llamado. Los valores `NA` pueden estar en cualquier clase de vectores: lógicos, enteros, numéricos y caracteres.

NA

En general, muchas veces **no** hay datos para todas las observaciones, por lo que se puede poner `NA` para estos casos. Que representa "not available", o sea: "no disponible". Dependiendo de las circunstancias y la naturaleza de los datos estos se pueden tratar de varias formas, muchas de las funciones base tiene el parámetro `na.rm` que viene pre definido como `FALSE`, si se le llama con `na.rm = FALSE` se omite ese valor a la hora de realizar el llamado. Los valores `NA` pueden estar en cualquier clase de vectores: lógicos, enteros, numéricos y caracteres. En general las funciones al recibir un `NA` van a devolver un `NA`, incluso las comparaciones: `NA == 1`, `NA == TRUE`, `NA == 'Hola'` y `NA == NA`.

NA

En general, muchas veces **no** hay datos para todas las observaciones, por lo que se puede poner `NA` para estos casos. Que representa "not available", o sea: "no disponible". Dependiendo de las circunstancias y la naturaleza de los datos estos se pueden tratar de varias formas, muchas de las funciones base tiene el parámetro `na.rm` que viene pre definido como `FALSE`, si se le llama con `na.rm = FALSE` se omite ese valor a la hora de realizar el llamado. Los valores `NA` pueden estar en cualquier clase de vectores: lógicos, enteros, numéricos y caracteres. En general las funciones al recibir un `NA` van a devolver un `NA`, incluso las comparaciones: `NA == 1`, `NA == TRUE`, `NA == 'Hola'` y `NA == NA`. Para este último caso, existe una función que se llama `is.na` que indica si el valor es `NA` o no, y se puede combiar con `ifelse` para hacer cambios sobre estos.

NA

En general, muchas veces **no** hay datos para todas las observaciones, por lo que se puede poner `NA` para estos casos. Que representa "not available", o sea: "no disponible". Dependiendo de las circunstancias y la naturaleza de los datos estos se pueden tratar de varias formas, muchas de las funciones base tiene el parámetro `na.rm` que viene pre definido como `FALSE`, si se le llama con `na.rm = FALSE` se omite ese valor a la hora de realizar el llamado. Los valores `NA` pueden estar en cualquier clase de vectores: lógicos, enteros, numéricos y caracteres. En general las funciones al recibir un `NA` van a devolver un `NA`, incluso las comparaciones: `NA == 1`, `NA == TRUE`, `NA == 'Hola'` y `NA == NA`. Para este último caso, existe una función que se llama `is.na` que indica si el valor es `NA` o no, y se puede combiar con `ifelse` para hacer cambios sobre estos.

Calcule la media del vector `seq(-120,30,by = 15)/(2*(-8:2))`, primero usando `na.rm = TRUE` y luego usando `na.rm = FALSE`.

`right_join`

Esta función es similar al `left_join` solo que conserva los valores del **segundo** argumento (cuando se lee es el de la derecha):

right_join

Esta función es similar al `left_join` solo que conserva los valores del **segundo** argumento (cuando se lee es el de la derecha):

```
right_join(x1,y1,by = 'nombre')
```

```
##   nombre valor_x valor_y
## 1      a      x1      y1
## 2      a      x4      y1
## 3      b      x2      y2
## 4      d    <NA>      y3
## 5      b      x2      y5
```

full_join

`full_join`

Conserva **todas** las filas posibles de ambos argumentos, independientemente de si tienen un valor en el otro lado:

full_join

Conserva **todas** las filas posibles de ambos argumentos, independientemente de si tienen un valor en el otro lado:

```
full_join(x1,y1,by = 'nombre')
```

```
##   nombre valor_x valor_y
## 1      a      x1      y1
## 2      b      x2      y2
## 3      b      x2      y5
## 4      c      x3    <NA>
## 5      a      x4      y1
## 6      d    <NA>      y3
```

inner_join

inner_join

Conserva todas las observaciones que están en ambas tamblas:

inner_join

Conserva todas las observaciones que están en ambas tamblas:

```
inner_join(x1,y1,by = 'nombre')
```

```
##   nombre valor_x valor_y
## 1      a      x1      y1
## 2      b      x2      y2
## 3      b      x2      y5
## 4      a      x4      y1
```

inner_join

Conserva todas las observaciones que están en ambas tamblas:

```
inner_join(x1,y1,by = 'nombre')
```

```
##   nombre valor_x valor_y
## 1      a      x1      y1
## 2      b      x2      y2
## 3      b      x2      y5
## 4      a      x4      y1
```

Con esta se tiene certeza de que no se van a tener `NA` nuevos por el `join`.

inner_join

Conserva todas las observaciones que están en ambas tamblas:

```
inner_join(x1,y1,by = 'nombre')
```

```
##   nombre valor_x valor_y
## 1      a      x1      y1
## 2      b      x2      y2
## 3      b      x2      y5
## 4      a      x4      y1
```

Con esta se tiene certeza de que no se van a tener `NA` nuevos por el `join`. ¿Porqué?

inner_join

Conserva todas las observaciones que están en ambas tamblas:

```
inner_join(x1,y1,by = 'nombre')
```

```
##   nombre valor_x valor_y
## 1      a      x1      y1
## 2      b      x2      y2
## 3      b      x2      y5
## 4      a      x4      y1
```

Con esta se tiene certeza de que no se van a tener `NA` nuevos por el `join`. ¿Porqué? Si quisiera replicar un `inner_join` usando un `full_join` y varios filter, ¿cómo lo haría?

inner_join

Conserva todas las observaciones que están en ambas tamblas:

```
inner_join(x1,y1,by = 'nombre')
```

```
##   nombre valor_x valor_y
## 1      a      x1      y1
## 2      b      x2      y2
## 3      b      x2      y5
## 4      a      x4      y1
```

Con esta se tiene certeza de que no se van a tener `NA` nuevos por el `join`. ¿Porqué? Si quisiera replicar un `inner_join` usando un `full_join` y varios filter, ¿cómo lo haría? ¿Podría hacer lo mismo con un `right_join` o un `left_join`?

semi_join

semi_join

Conserva todas las observaciones del primer argumento que tienen al menos una coincidencia en el segundo argumento.

```
semi_join(x1,y1,by = 'nombre')
```

```
##   nombre valor_x
## 1      a      x1
## 2      a      x4
## 3      b      x2
```

semi_join

Conserva todas las observaciones del primer argumento que tienen al menos una coincidencia en el segundo argumento.

```
semi_join(x1,y1,by = 'nombre')
```

```
##   nombre valor_x
## 1      a      x1
## 2      a      x4
## 3      b      x2
```

Con esta se tiene la misma garantía que la que se tiene con el `inner_join`.

semi_join

Conserva todas las observaciones del primer argumento que tienen al menos una coincidencia en el segundo argumento.

```
semi_join(x1,y1,by = 'nombre')
```

```
##   nombre valor_x
## 1      a      x1
## 2      a      x4
## 3      b      x2
```

Con esta se tiene la misma garantía que la que se tiene con el `inner_join`. ¿Porqué?

semi_join

Conserva todas las observaciones del primer argumento que tienen al menos una coincidencia en el segundo argumento.

```
semi_join(x1,y1,by = 'nombre')
```

```
##   nombre valor_x
## 1      a      x1
## 2      a      x4
## 3      b      x2
```

Con esta se tiene la misma garantía que la que se tiene con el `inner_join`. ¿Porqué? ¿Como podría hacer lo mismo pero usando otro tipo de `join`?

anti_join

anti_join

Esta quita todas las observaciones del primer argumento que tienen un valor correspondiente en el segundo argumento:

```
anti_join(x1,y1,by = 'nombre')
```

```
##   nombre valor_x
## 1      c      x3
```

anti_join

Esta quita todas las observaciones del primer argumento que tienen un valor correspondiente en el segundo argumento:

```
anti_join(x1,y1,by = 'nombre')
```

```
##   nombre valor_x
## 1      c      x3
```

¿Se obtiene un resultado distinto si se llama como primer argumento `y1` y de segundo argumento `x1`?

anti_join

Esta quita todas las observaciones del primer argumento que tienen un valor correspondiente en el segundo argumento:

```
anti_join(x1,y1,by = 'nombre')
```

```
##   nombre valor_x
## 1      c      x3
```

¿Se obtiene un resultado distinto si se llama como primer argumento `y1` y de segundo argumento `x1`?

¿Qué se obtiene al llamar con el mismo dataframe en ambos argumentos?

anti_join

Esta quita todas las observaciones del primer argumento que tienen un valor correspondiente en el segundo argumento:

```
anti_join(x1,y1,by = 'nombre')
```

```
##   nombre valor_x
## 1      c      x3
```

¿Se obtiene un resultado distinto si se llama como primer argumento `y1` y de segundo argumento `x1`?

¿Qué se obtiene al llamar con el mismo dataframe en ambos argumentos?

¿Se puede garantizar lo mismo que con los `join` anteriores?

anti_join

Esta quita todas las observaciones del primer argumento que tienen un valor correspondiente en el segundo argumento:

```
anti_join(x1,y1,by = 'nombre')
```

```
##   nombre valor_x
## 1      c      x3
```

¿Se obtiene un resultado distinto si se llama como primer argumento `y1` y de segundo argumento `x1`?

¿Qué se obtiene al llamar con el mismo dataframe en ambos argumentos?

¿Se puede garantizar lo mismo que con los `join` anteriores?

¿Como puede replicar este `join` usando otros tipos de `*_join`?

```
by =
```

`by =`

Si las tablas que se están usando tienen nombres de columnas distintas, se puede usar un vector con nombres como parámetro para realizar el `*_join`, y se conservan los nombres del primer `data.frame`:

by =

Si las tablas que se están usando tienen nombres de columnas distintas, se puede usar un vector con nombres como parámetro para realizar el `*_join`, y se conservan los nombres del primer `data.frame`:

```
x1 %>%
  left_join(y2, by = c('nombre' = 'Otro_nombre'))
```

```
##   nombre valor_x valor_y
## 1      a      x1      y1
## 2      b      x2      y2
## 3      b      x2      y5
## 4      c      x3    <NA>
## 5      a      x4      y1
```

by =

Si las tablas que se están usando tienen nombres de columnas distintas, se puede usar un vector con nombres como parámetro para realizar el `*_join`, y se conservan los nombres del primer `data.frame`:

```
x1 %>%
  left_join(y2, by = c('nombre' = 'Otro_nombre'))
```

```
##   nombre valor_x valor_y
## 1      a      x1      y1
## 2      b      x2      y2
## 3      b      x2      y5
## 4      c      x3    <NA>
## 5      a      x4      y1
```

¿Cómo queda el llamado anterior si se quiere poner de primer argumento `y2`?

```
by =
```

`by =`

Se pueden indicar más columnas usando vectores de mayor longitud:

by =

Se pueden indicar más columnas usando vectores de mayor longitud:

```
x1 %>%
  mutate(Mi_valor = row_number()) %>%
  full_join(x2,by = c('nombre' = 'Otro_nombre',
                      'valor_x' = 'Otro_valor_x'))
```

```
##   nombre valor_x Mi_valor nuevo_valor
## 1      a      x1        1       15.70
## 2      b      x2        2        7.46
## 3      c      x3        3        6.19
## 4      a      x4        4          NA
## 5      a      x2       NA        8.80
```

by =

Se pueden indicar más columnas usando vectores de mayor longitud:

```
x1 %>%
  mutate(Mi_valor = row_number()) %>%
  full_join(x2,by = c('nombre' = 'Otro_nombre',
                      'valor_x' = 'Otro_valor_x'))
```

```
##   nombre valor_x Mi_valor nuevo_valor
## 1      a      x1        1       15.70
## 2      b      x2        2        7.46
## 3      c      x3        3        6.19
## 4      a      x4        4          NA
## 5      a      x2       NA        8.80
```

Haga lo anterior (incluyendo el `mutate`) pero usando un `left_join`.

by =

Se pueden indicar más columnas usando vectores de mayor longitud:

```
x1 %>%
  mutate(Mi_valor = row_number()) %>%
  full_join(x2,by = c('nombre' = 'Otro_nombre',
                      'valor_x' = 'Otro_valor_x'))
```

```
##   nombre valor_x Mi_valor nuevo_valor
## 1      a      x1        1       15.70
## 2      b      x2        2        7.46
## 3      c      x3        3        6.19
## 4      a      x4        4          NA
## 5      a      x2       NA        8.80
```

Haga lo anterior (incluyendo el `mutate`) pero usando un `left_join`. Repita el ejercicio pero con un `right_join`.

```
by =
```

`by =`

Si los nombres por los que se va a unir son idénticos y son las únicas columnas que cumplen esto, se puede omitir el parámetro `by`, pero se imprime en consola una línea de aviso de cuales columnas se están usando:

by =

Si los nombres por los que se va a unir son idénticos y son las únicas columnas que cumplen esto, se puede omitir el parámetro by, pero se imprime en consola una línea de aviso de cuales columnas se están usando:

```
x1 %>% full_join(y1)
```

```
## Joining, by = "nombre"
```

```
##   nombre valor_x valor_y
## 1      a      x1      y1
## 2      b      x2      y2
## 3      b      x2      y5
## 4      c      x3    <NA>
## 5      a      x4      y1
## 6      d    <NA>      y3
```

Valores duplicados

Valores duplicados

Hasta el momento a cada una de las llaves le corresponde una única observación en al menos una de las tablas por las que se realiza la unión. En caso de que ambas tablas presenten valores duplicados, se hacen todas las posibles combinaciones (i.e. producto cartesiano) entre las coincidencias que se encuentren:

Valores duplicados

Hasta el momento a cada una de las llaves le corresponde una única observación en al menos una de las tablas por las que se realiza la unión. En caso de que ambas tablas presenten valores duplicados, se hacen todas las posibles combinaciones (i.e. producto cartesiano) entre las coincidencias que se encuentren:

```
x1 %>% left_join(y3, by = c('nombre'))
```

```
##   nombre valor_x valor_y
## 1      a      x1     3y1
## 2      a      x1     3y4
## 3      b      x2     3y2
## 4      c      x3    <NA>
## 5      a      x4     3y1
## 6      a      x4     3y4
```

Valores duplicados

Hasta el momento a cada una de las llaves le corresponde una única observación en al menos una de las tablas por las que se realiza la unión. En caso de que ambas tablas presenten valores duplicados, se hacen todas las posibles combinaciones (i.e. producto cartesiano) entre las coincidencias que se encuentren:

```
x1 %>% left_join(y3, by = c('nombre'))
```

```
##   nombre valor_x valor_y
## 1      a      x1     3y1
## 2      a      x1     3y4
## 3      b      x2     3y2
## 4      c      x3    <NA>
## 5      a      x4     3y1
## 6      a      x4     3y4
```

Note que ahora se obtienen 4 observaciones con `nombre = 'a'`.

`bind_*`

Estas funciones unen (pegan) data.frames en un sentido más literal que el que se usa para el join. Hay dos funciones de `bind_*`: `bind_cols` que pega **columnas** nuevas, y `bind_rows` que pega **filas** nuevas:

bind_*

Estas funciones unen (pegan) data.frames en un sentido más literal que el que se usa para el join. Hay dos funciones de `bind_*`: `bind_cols` que pega **columnas** nuevas, y `bind_rows` que pega **filas** nuevas:

```
bind_cols(x1,x2)
```

```
##   nombre valor_x Otro_nombre Otro_valor_x nuevo_valor
## 1      a      x1           a           x1       15.70
## 2      b      x2           b           x2        7.46
## 3      c      x3           c           x3        6.19
## 4      a      x4           a           x2        8.80
```

bind_*

Estas funciones unen (pegan) data.frames en un sentido más literal que el que se usa para el join. Hay dos funciones de `bind_*`: `bind_cols` que pega **columnas** nuevas, y `bind_rows` que pega **filas** nuevas:

```
bind_cols(x1,x2)
```

```
##   nombre valor_x Otro_nombre Otro_valor_x nuevo_valor
## 1      a      x1           a           x1       15.70
## 2      b      x2           b           x2        7.46
## 3      c      x3           c           x3        6.19
## 4      a      x4           a           x2        8.80
```

Si los argumentos no coinciden en la cantidad de columnas, se obtiene un error.

bind_*

Estas funciones unen (pegan) data.frames en un sentido más literal que el que se usa para el join. Hay dos funciones de `bind_*`: `bind_cols` que pega **columnas** nuevas, y `bind_rows` que pega **filas** nuevas:

```
bind_cols(x1,x2)
```

```
##   nombre valor_x Otro_nombre Otro_valor_x nuevo_valor
## 1      a      x1           a           x1       15.70
## 2      b      x2           b           x2        7.46
## 3      c      x3           c           x3        6.19
## 4      a      x4           a           x2        8.80
```

Si los argumentos no coinciden en la cantidad de columnas, se obtiene un error. Obtenga ese error usando:

```
x2 %>%
  filter(nuevo_valor < 10 ) %>%
  bind_cols(x1)
```

bind_rows

Para esto, se ocupa que los dataframes que se pasan como argumento tengan los mismos nombres de columnas:

bind_rows

Para esto, se ocupa que los dataframes que se pasan como argumento tengan los mismos nombres de columnas:

```
y1 %>% bind_rows(y3)
```

```
##   nombre valor_y
## 1      a      y1
## 2      b      y2
## 3      d      y3
## 4      b      y5
## 5      a     3y1
## 6      b     3y2
## 7      d     3y3
## 8      a     3y4
```

bind_rows

En caso de que no, toma todas las columnas de los dataframes que se le pasen, y une en las que coinciden:

bind_rows

En caso de que no, toma todas las columnas de los dataframes que se le pasen, y une en las que coinciden:

```
x1 %>% bind_rows(y3)
```

```
##   nombre valor_x valor_y
## 1      a      x1    <NA>
## 2      b      x2    <NA>
## 3      c      x3    <NA>
## 4      a      x4    <NA>
## 5      a    <NA>     3y1
## 6      b    <NA>     3y2
## 7      d    <NA>     3y3
## 8      a    <NA>     3y4
```

Nota sobre `bind_*`

Nota sobre `bind_*`

Por ahora solo hemos usado dos argumentos para realizar los `bind`, sin embargo, pueden recibir tantos argumentos como se quiera:

Nota sobre `bind_*`

Por ahora solo hemos usado dos argumentos para realizar los `bind`, sin embargo, pueden recibir tantos argumentos como se quiera:

```
x1 %>% bind_cols(x2,y1,y2,y3)
```

```
##   nombre valor_x Otro_nombre Otro_valor_x nuevo_valor no
## 1      a      x1           a           x1       15.70
## 2      b      x2           b           x2        7.46
## 3      c      x3           c           x3        6.19
## 4      a      x4           a           x2        8.80
##   Otro_nombre1 valor_y1 nombre2 valor_y2
## 1            a       y1       a      3y1
## 2            b       y2       b      3y2
## 3            d       y3       d      3y3
## 4            b       y5       a      3y4
```

`expand.grid`

Algunas veces hay que completar *a pata* los datos que se tienen, o se quieren obtener todas las posibles combinaciones entre varios vectores. Para este tipo de tareas, existe la función `expand.grid`:

expand.grid

Algunas veces hay que completar *a pata* los datos que se tienen, o se quieren obtener todas las posibles combinaciones entre varios vectores. Para este tipo de tareas, existe la función `expand.grid`:

```
expand.grid(Anno = 2017:2018,Mes = 1:3)
```

```
##   Anno Mes
## 1 2017   1
## 2 2018   1
## 3 2017   2
## 4 2018   2
## 5 2017   3
## 6 2018   3
```

En general, pueden ponerse tantos vectores como se quieran.

Ejercicio

Complete la tabla de salarios que se usó la clase anterior, puede apoyarse en las funciones `expand.grid`, `bind_*`, y `unique` para obtener todos los nombres de la tabla.

Ejercicio

Complete la tabla de salarios que se usó la clase anterior, puede apoyarse en las funciones `expand.grid`, `bind_*`, y `unique` para obtener todos los nombres de la tabla.

Termine los ejercicios que se trabajaron el lunes, si ve la oportunidad de usar lo que vimos en la clase de hoy, hágalo.

Ejercicio

Complete la tabla de salarios que se usó la clase anterior, puede apoyarse en las funciones `expand.grid`, `bind_*`, y `unique` para obtener todos los nombres de la tabla.

Termine los ejercicios que se trabajaron el lunes, si ve la oportunidad de usar lo que vimos en la clase de hoy, hágalo.

`bind_rows`

Próximamente:

tidyr :)