

Unir tablas

Jorge Loría

Unir tablas

Pocas veces se tiene la suerte de tener todos los datos en una misma tabla, o que únicamente se ocupan los datos de una sola tabla.

Unir tablas

Pocas veces se tiene la suerte de tener todos los datos en una misma tabla, o que únicamente se ocupan los datos de una sola tabla.

Es por esto, que muchas veces se tienen que **unir** dataframes.

Unir tablas

Pocas veces se tiene la suerte de tener todos los datos en una misma tabla, o que únicamente se ocupan los datos de una sola tabla.

Es por esto, que muchas veces se tienen que **unir** dataframes. El método de unión va a depender tanto de los datos de los que se dispone, como la organización que se les de.

Unir tablas

Pocas veces se tiene la suerte de tener todos los datos en una misma tabla, o que únicamente se ocupan los datos de una sola tabla.

Es por esto, que muchas veces se tienen que **unir** dataframes. El método de unión va a depender tanto de los datos de los que se dispone, como la organización que se les de. Por el momento vamos a asumir que vienen acomodados de una forma ***linda***, que luego definiremos.

Unir tablas

Pocas veces se tiene la suerte de tener todos los datos en una misma tabla, o que únicamente se ocupan los datos de una sola tabla.

Es por esto, que muchas veces se tienen que **unir** dataframes. El método de unión va a depender tanto de los datos de los que se dispone, como la organización que se les de. Por el momento vamos a asumir que vienen acomodados de una forma ***linda***, que luego definiremos.

Vamos a ver varios métodos para unión de tablas.

Datos de ejemplo

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

```
library(dplyr)
x1 <- tibble(nombre = c('a','b','c','a'),
              valor_x = c('x1','x2','x3','x4'))
# es como un dataframe, pero no hace los strings en factor
y1 <- tibble(nombre = c('a','b','d','b'),
              valor_y = c('y1','y2','y3','y5'))
```

Más datos

```
y2 <- y1 %>%  
  rename('Otro_nombre' = 'nombre')  
  
x2 <- tibble(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))  
  
y3 <- tibble(nombre = c('a', 'b','d','a'),  
             valor_y = c('3y1','3y2','3y3','3y4'))
```


*_join

Hay 6 tipos de join, los cuales se parecen mucho entre sí. La gran diferencia es a *cual* de las tablas se le da 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 da 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 da 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 columna `'nombre'` y agrega las columnas del segundo argumento usando como referencia las filas donde hubo coincidencias en la primera tabla.

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 columna 'nombre' y agrega las columnas del segundo argumento usando como referencia las filas donde hubo coincidencias en la primera tabla.

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

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


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.

Defina `v1 <- seq(-120,30,by = 15) / (exp(-8:2)-1)` y calcule su media usando la función `mean` , 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')
```

```
## # A tibble: 5 x 3
##   nombre valor_x valor_y
##   <chr>   <chr>   <chr>
## 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')
```

```
## # A tibble: 6 x 3
##   nombre valor_x valor_y
##   <chr>   <chr>   <chr>
## 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 tablas:

inner_join

Conserva todas las observaciones que están en ambas tablas:

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

```
## # A tibble: 4 x 3
##   nombre valor_x valor_y
##   <chr>   <chr>   <chr>
## 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 tablas:

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

```
## # A tibble: 4 x 3
##   nombre valor_x valor_y
##   <chr>   <chr>   <chr>
## 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 tablas:

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

```
## # A tibble: 4 x 3
##   nombre valor_x valor_y
##   <chr>   <chr>   <chr>
## 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 tablas:

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

```
## # A tibble: 4 x 3
##   nombre valor_x valor_y
##   <chr>   <chr>   <chr>
## 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 tablas:

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

```
## # A tibble: 4 x 3
##   nombre valor_x valor_y
##   <chr>   <chr>   <chr>
## 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 y no agrega las columnas de la primera tabla

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

```
## # A tibble: 3 x 2
##   nombre valor_x
##   <chr>   <chr>
## 1 a      x1
## 2 b      x2
## 3 a      x4
```

semi_join

Conserva todas las observaciones del primer argumento que tienen al menos una coincidencia en el segundo argumento y no agrega las columnas de la primera tabla

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

```
## # A tibble: 3 x 2
##   nombre valor_x
##   <chr>   <chr>
## 1 a      x1
## 2 b      x2
## 3 a      x4
```

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 y no agrega las columnas de la primera tabla

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

```
## # A tibble: 3 x 2
##   nombre valor_x
##   <chr>   <chr>
## 1 a      x1
## 2 b      x2
## 3 a      x4
```

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 y no agrega las columnas de la primera tabla

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

```
## # A tibble: 3 x 2
##   nombre valor_x
##   <chr>   <chr>
## 1 a      x1
## 2 b      x2
## 3 a      x4
```

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')
```

```
## # A tibble: 1 x 2  
##   nombre valor_x  
##   <chr>   <chr>  
## 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')
```

```
## # A tibble: 1 x 2  
##   nombre valor_x  
##   <chr>   <chr>  
## 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')
```

```
## # A tibble: 1 x 2  
##   nombre valor_x  
##   <chr>   <chr>  
## 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')
```

```
## # A tibble: 1 x 2
##   nombre valor_x
##   <chr>   <chr>
## 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')
```

```
## # A tibble: 1 x 2
##   nombre valor_x
##   <chr>   <chr>
## 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'))
```

```
## # A tibble: 5 x 3  
##   nombre valor_x valor_y  
##   <chr>   <chr>   <chr>  
## 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'))
```

```
## # A tibble: 5 x 3  
##   nombre valor_x valor_y  
##   <chr>   <chr>   <chr>  
## 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'))
```

```
## # A tibble: 5 x 4  
##   nombre valor_x Mi_valor nuevo_valor  
##   <chr>   <chr>     <int>     <dbl>  
## 1 a      x1         1       15.7  
## 2 b      x2         2        7.46  
## 3 c      x3         3        6.19  
## 4 a      x4         4        NA  
## 5 a      x2        NA        8.8
```

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'))
```

```
## # A tibble: 5 x 4  
##   nombre valor_x Mi_valor nuevo_valor  
##   <chr>   <chr>     <int>     <dbl>  
## 1 a      x1         1      15.7  
## 2 b      x2         2       7.46  
## 3 c      x3         3       6.19  
## 4 a      x4         4        NA  
## 5 a      x2        NA       8.8
```

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'))
```

```
## # A tibble: 5 x 4  
##   nombre valor_x Mi_valor nuevo_valor  
##   <chr>   <chr>     <int>     <dbl>  
## 1 a      x1         1       15.7  
## 2 b      x2         2        7.46  
## 3 c      x3         3        6.19  
## 4 a      x4         4        NA  
## 5 a      x2         NA        8.8
```

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"
```

```
## # A tibble: 6 x 3
```

```
##   nombre valor_x valor_y
```

```
##   <chr>   <chr>   <chr>
```

```
## 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'))
```

```
## # A tibble: 6 x 3
##   nombre valor_x valor_y
##   <chr>   <chr>   <chr>
## 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'))
```

```
## # A tibble: 6 x 3
##   nombre valor_x valor_y
##   <chr>   <chr>   <chr>
## 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 hemos usado 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 hemos usado 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)
```

```
## # A tibble: 4 x 5
##   nombre valor_x Otro_nombre Otro_valor_x nuevo_valor
##   <chr>   <chr>   <chr>         <chr>         <dbl>
## 1 a      x1      a            x1            15.7
## 2 b      x2      b            x2            7.46
## 3 c      x3      c            x3            6.19
## 4 a      x4      a            x2            8.8
```

bind_*

Estas funciones unen (pegan) data.frames en un sentido más literal que el que hemos usado 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)
```

```
## # A tibble: 4 x 5
##   nombre valor_x Otro_nombre Otro_valor_x nuevo_valor
##   <chr>   <chr>   <chr>         <chr>         <dbl>
## 1 a      x1      a            x1            15.7
## 2 b      x2      b            x2            7.46
## 3 c      x3      c            x3            6.19
## 4 a      x4      a            x2            8.8
```

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 hemos usado 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)
```

```
## # A tibble: 4 x 5
##   nombre valor_x Otro_nombre Otro_valor_x nuevo_valor
##   <chr>   <chr>   <chr>         <chr>         <dbl>
## 1 a      x1      a            x1            15.7
## 2 b      x2      b            x2            7.46
## 3 c      x3      c            x3            6.19
## 4 a      x4      a            x2            8.8
```

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

```
x2 %>%
```

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)
```

```
## # A tibble: 8 x 2
##   nombre valor_y
##   <chr>   <chr>
## 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)
```

```
## # A tibble: 8 x 3
##   nombre valor_x valor_y
##   <chr>   <chr>   <chr>
## 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
```

bind_rows

Se puede incluir el parámetro `.id` para identificar a cual tabla pertenecen las filas que se obtienen:

```
y1 %>% bind_rows(y3, .id = 'Num_Tabla')
```

```
## # A tibble: 8 x 3
##   Num_Tabla nombre valor_y
##   <chr>      <chr>  <chr>
## 1 1          a      y1
## 2 1          b      y2
## 3 1          d      y3
## 4 1          b      y5
## 5 2          a     3y1
## 6 2          b     3y2
## 7 2          d     3y3
## 8 2          a     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)
```

```
## # A tibble: 4 x 11
##   nombre valor_x Otro_nombre Otro_valor_x nuevo_valor no
##   <chr>   <chr>   <chr>         <chr>         <dbl> <chr>
## 1 a      x1      a            x1            15.7 a
## 2 b      x2      b            x2            7.46 b
## 3 c      x3      c            x3            6.19 d
## 4 a      x4      a            x2            8.8  b
## # ... with 4 more variables: Otro_nombre1 <chr>, valor_y1 <dbl>,
## #   nombre2 <chr>, valor_y2 <chr>
```

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 = 2018:2019,  
            Nombre = c('Pablo', 'Ana'))
```

```
##      Anno Nombre  
## 1 2018   Pablo  
## 2 2019   Pablo  
## 3 2018     Ana  
## 4 2019     Ana
```

En general, pueden ponerse tantos vectores como se quieran.

expand.grid

Como el nombre se obtiene como un factor, se puede usar el parámetro `stringsAsFactors`, para obtenerlos como strings.

```
expand.grid(Anno = 2017:2018,Mes = 1:3,  
            Nombre = c('Pablo','Ana'),stringsAsFactors = F)
```

##		Anno	Mes	Nombre
##	1	2017	1	Pablo
##	2	2018	1	Pablo
##	3	2017	2	Pablo
##	4	2018	2	Pablo
##	5	2017	3	Pablo
##	6	2018	3	Pablo
##	7	2017	1	Ana
##	8	2018	1	Ana
##	9	2017	2	Ana
##	10	2018	2	Ana
##	11	2017	3	Ana
##	12	2018	3	Ana

crossing

También se puede usar la función `crossing` que viene en el paquete `tidyr`:

```
library(tidyr)
crossing(Anno = 2018:2019,
         Nombre = c('Pablo', 'Ana'))
```

```
## # A tibble: 4 x 2
##   Anno Nombre
##   <int> <chr>
## 1  2018 Ana
## 2  2018 Pablo
## 3  2019 Ana
## 4  2019 Pablo
```

Próximamente:

tidyr :)

Ejercicio

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

Termine los ejercicios que se trabajaron la clase anterior, 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` o `crossing`, `bind_*`, y `unique` para obtener todos los nombres de la tabla.

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

`bind_rows`