Unidad: Introducción a R

Tidyverse: Parte 2 - Joins, Pivots y Funciones Avanzadas

Nicolás Sidicaro
Abril 2025

Recordando la Clase Anterior

- Introducción al ecosistema **Tidyverse**
- Uso del operador pipe `%>%`
- Tibbles como versión moderna de dataframes
- Verbos básicos de **dplyr**:
  - `filter()`: filtrar filas
  - `select()`: seleccionar columnas
  - `mutate()`: crear/transformar variables
  - `arrange()`: ordenar filas
  - `summarise()`: resumir datos
  - `group_by()`: agrupar observaciones

Agenda para Hoy

1. **Joins**: combinación de tablas
    - Tipos de joins y su funcionamiento
    - Cuándo usar cada tipo
2. **Pivots**: reestructuración de datos
    - `pivot_longer()`: convertir columnas en filas
    - `pivot_wider()`: convertir filas en columnas
    - Casos de uso típicos
3. **Funciones adicionales de Tidyverse**
    - Funciones útiles y menos conocidas
    - Casos de aplicación

Joins: ¿Qué son y para qué sirven?

Los **joins** son operaciones que nos permiten **combinar tablas relacionadas** basándose en columnas comunes.

- Provienen del concepto de álgebra relacional
- Esenciales en bases de datos relacionales
- En R, implementados de forma eficiente en **dplyr**

Permiten responder preguntas como:

- ¿Cuáles son las ventas por categoría de producto?
- ¿Qué empleados están asignados a qué proyectos?
- ¿Cómo se comparan los presupuestos con los gastos reales?

Datos de Ejemplo para Joins

```
suppressMessages(library(tidyverse))

# Tabla de clientes
clientes ← tibble(
  id_cliente = c(1, 2, 3, 4, 5),
  nombre = c("Ana García", "Juan López", "María Rodríguez", "Carlos Martínez", "Laura
  ciudad = c("Buenos Aires", "Córdoba", "Rosario", "Mendoza", "La Plata")
)

# Tabla de pedidos
pedidos ← tibble(
  id_pedido = c(101, 102, 103, 104, 105, 106),
  id_cliente = c(1, 3, 3, 2, 6, 7),
  fecha = c("2025-02-10", "2025-02-15", "2025-02-20", "2025-02-25", "2025-03-01", "202
  monto = c(1500, 800, 1200, 950, 2000, 1750)
)
```

Visualización de las Tablas

Tabla de clientes:

```
clientes
```

```
## # A tibble: 5 × 3
##   id_cliente nombre          ciudad
##        <dbl> <chr>           <chr>
## 1          1 Ana García      Buenos Aires
## 2          2 Juan López      Córdoba
## 3          3 María Rodríguez Rosario
## 4          4 Carlos Martínez Mendoza
## 5          5 Laura Sánchez   La Plata
```

Tabla de pedidos:

```
pedidos
```

```
## # A tibble: 6 × 4
##   id_pedido id_cliente fecha      monto
##       <dbl>      <dbl> <chr>      <dbl>
## 1       101          1 2025-02-10  1500
## 2       102          3 2025-02-15   800
## 3       103          3 2025-02-20  1200
## 4       104          2 2025-02-25   950
## 5       105          6 2025-03-01  2000
## 6       106          7 2025-03-05  1750
```

Tipos de Joins en dplyr

dplyr ofrece 6 tipos principales de joins:

1. **inner_join()**: solo filas con coincidencias en ambas tablas
2. **left_join()**: todas las filas de la tabla izquierda
3. **right_join()**: todas las filas de la tabla derecha
4. **full_join()**: todas las filas de ambas tablas
5. **semi_join()**: filas de la izquierda con coincidencias en la derecha
6. **anti_join()**: filas de la izquierda sin coincidencias en la derecha

Representación Visual de Joins

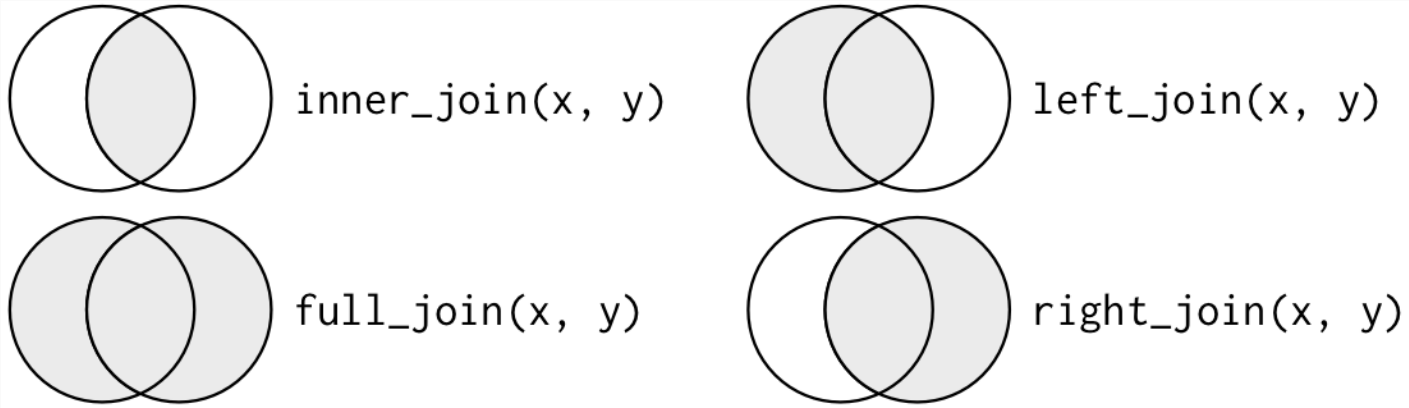


Diagrama de Venn de los diferentes tipos de joins (Fuente: R for Data Science)

Inner Join

Une filas de ambas tablas **solo** cuando hay coincidencias en la(s) columna(s) de unión:

```
# Inner join: solo clientes que tienen pedidos y viceversa
clientes %>%
  inner_join(pedidos, by = "id_cliente")
```

```
## # A tibble: 4 × 6
##   id_cliente nombre          ciudad       id_pedido fecha      monto
##        <dbl> <chr>           <chr>            <dbl> <chr>      <dbl>
## 1          1 Ana García      Buenos Aires       101 2025-02-10  1500
## 2          2 Juan López      Córdoba            104 2025-02-25   950
## 3          3 María Rodríguez Rosario            102 2025-02-15   800
## 4          3 María Rodríguez Rosario            103 2025-02-20  1200
```

Notar: Solo aparecen clientes con IDs 1, 2 y 3, y solo pedidos asociados a estos clientes

Left Join

Mantiene **todas las filas** de la tabla izquierda, incluso si no hay coincidencias:

```
# Left join: todos los clientes, tengan o no pedidos
clientes %>%
  left_join(pedidos, by = "id_cliente")
```

```
## # A tibble: 6 × 6
##   id_cliente nombre          ciudad       id_pedido fecha      monto
##        <dbl> <chr>           <chr>            <dbl> <chr>      <dbl>
## 1          1 Ana García      Buenos Aires       101 2025-02-10  1500
## 2          2 Juan López      Córdoba            104 2025-02-25   950
## 3          3 María Rodríguez Rosario            102 2025-02-15   800
## 4          3 María Rodríguez Rosario            103 2025-02-20  1200
## 5          4 Carlos Martínez Mendoza             NA <NA>          NA
## 6          5 Laura Sánchez   La Plata            NA <NA>          NA
```

Notar: Incluye a los clientes 4 y 5 aunque no tienen pedidos (valores NA)

Right Join

Mantiene **todas las filas** de la tabla derecha, incluso si no hay coincidencias:

```
# Right join: todos los pedidos, estén o no asociados a clientes conocidos
clientes %>%
  right_join(pedidos, by = "id_cliente")
```

```
## # A tibble: 6 × 6
##   id_cliente nombre          ciudad       id_pedido fecha      monto
##        <dbl> <chr>           <chr>            <dbl> <chr>      <dbl>
## 1          1 Ana García      Buenos Aires       101 2025-02-10  1500
## 2          2 Juan López      Córdoba            104 2025-02-25   950
## 3          3 María Rodríguez Rosario            102 2025-02-15   800
## 4          3 María Rodríguez Rosario            103 2025-02-20  1200
## 5          6 <NA>            <NA>               105 2025-03-01  2000
## 6          7 <NA>            <NA>               106 2025-03-05  1750
```

Notar: Incluye pedidos con ID cliente 6 y 7, aunque no están en la tabla de clientes

Full Join

Mantiene **todas las filas** de ambas tablas, coincidan o no:

```
# Full join: todos los clientes y todos los pedidos
clientes %>%
  full_join(pedidos, by = "id_cliente")
```

```
## # A tibble: 8 × 6
##   id_cliente nombre          ciudad       id_pedido fecha      monto
##        <dbl> <chr>           <chr>            <dbl> <chr>      <dbl>
## 1          1 Ana García      Buenos Aires       101 2025-02-10  1500
## 2          2 Juan López      Córdoba            104 2025-02-25   950
## 3          3 María Rodríguez Rosario            102 2025-02-15   800
## 4          3 María Rodríguez Rosario            103 2025-02-20  1200
## 5          4 Carlos Martínez Mendoza             NA <NA>          NA
## 6          5 Laura Sánchez   La Plata            NA <NA>          NA
## 7          6 <NA>            <NA>               105 2025-03-01  2000
## 8          7 <NA>            <NA>               106 2025-03-05  1750
```

Notar: Incluye todos los clientes y todos los pedidos, con NA donde corresponda

Semi Join

Mantiene filas de la tabla izquierda que tienen **al menos una coincidencia** en la derecha:

```
# Semi join: solo clientes que han realizado algún pedido
clientes %>%
  semi_join(pedidos, by = "id_cliente")
```

```
## # A tibble: 3 × 3
##   id_cliente nombre          ciudad
##        <dbl> <chr>           <chr>
## 1          1 Ana García      Buenos Aires
## 2          2 Juan López      Córdoba
## 3          3 María Rodríguez Rosario
```

Notar: Solo incluye a los clientes 1, 2 y 3, pero **sin traer datos** de la tabla de pedidos

Anti Join

Mantiene filas de la tabla izquierda que **no tienen coincidencias** en la derecha:

```
# Anti join: solo clientes que no han realizado pedidos
clientes %>%
  anti_join(pedidos, by = "id_cliente")
```

```
## # A tibble: 2 × 3
##   id_cliente nombre          ciudad
##        <dbl> <chr>           <chr>
## 1          4 Carlos Martínez Mendoza
## 2          5 Laura Sánchez   La Plata
```

Notar: Solo incluye a los clientes 4 y 5, que no tienen pedidos asociados

Uso de Claves Compuestas

Cuando la relación entre tablas se basa en múltiples columnas:

```
# Ejemplo con clave compuesta
items_pedido ← tibble(
  id_pedido = c(101, 101, 102, 103, 103, 104),
  id_producto = c(10, 20, 10, 30, 40, 20),
  cantidad = c(2, 1, 3, 1, 2, 2)
)

productos ← tibble(
  id_producto = c(10, 20, 30, 40, 50),
  nombre_producto = c("Laptop", "Monitor", "Teclado", "Mouse", "Altavoces"),
  precio = c(800, 300, 50, 25, 100)
)
```

Uso de Claves Compuestas

```r
# Join por múltiples columnas
pedidos %>%
  inner_join(items_pedido, by = "id_pedido") %>%
  inner_join(productos, by = "id_producto") %>%
  select(id_pedido, fecha, nombre_producto, cantidad, precio) %>%
  head(4)
```

```
## # A tibble: 4 × 5
##   id_pedido fecha      nombre_producto cantidad precio
##       <dbl> <chr>      <chr>              <dbl>  <dbl>
## 1       101 2025-02-10 Laptop                 2    800
## 2       101 2025-02-10 Monitor                1    300
## 3       102 2025-02-15 Laptop                 3    800
## 4       103 2025-02-20 Teclado                1     50
```

Columnas con nombres diferentes

Cuando las columnas de unión tienen nombres distintos:

```
# Tabla con nombre de columna diferente
nuevos_productos ← tibble(
  codigo_producto = c(10, 20, 30, 40, 50),
  descripcion = c("Laptop", "Monitor", "Teclado", "Mouse", "Altavoces"),
  precio_unitario = c(800, 300, 50, 25, 100)
)

# Join especificando la correspondencia entre columnas
items_pedido %>%
  inner_join(nuevos_productos, by = c("id_producto" = "codigo_producto")) %>%
  head(4)
```

```
## # A tibble: 4 × 5
##   id_pedido id_producto cantidad descripcion precio_unitario
##       <dbl>       <dbl>    <dbl> <chr>                 <dbl>
## 1       101          10        2 Laptop                  800
## 2       101          20        1 Monitor                 300
## 3       102          10        3 Laptop                  800
## 4       103          30        1 Teclado                  50
```

Consejos para Usar Joins Correctamente

- **Unicidad**: asegúrate de que las claves sean únicas en al menos una de las tablas
- **Consistencia**: verifica que los tipos de datos sean compatibles
- **Propósito**: elige el tipo de join según lo que necesites preservar:
  - **inner_join**: cuando solo necesitas coincidencias exactas
  - **left_join**: cuando necesitas preservar todos los registros de tu tabla principal
  - **right_join/full_join**: cuando necesitas asegurar que no se pierda información
  - **semi_join/anti_join**: para filtrar en base a la existencia (o no) en otra tabla

Pivots: Reestructuración de Datos

Los **pivots** nos permiten transformar la estructura de nuestros datos entre formatos:

- "Wide" (ancho): más columnas, menos filas
- "Long" (largo): menos columnas, más filas

Dos funciones principales:

- `pivot_longer()`: convierte de formato ancho a largo
- `pivot_wider()`: convierte de formato largo a ancho

Tidy Data: El Concepto Detrás de los

Según los principios de **tidy data** (Hadley Wickham):

1. Cada variable forma una columna
2. Cada observación forma una fila
3. Cada tipo de unidad observacional forma una tabla

Los pivots nos ayudan a transformar datos para que cumplan estos principios.

Ejemplo de Datos para Pivots

Datos de temperatura en formato "ancho":

```
# Temperaturas mensuales por ciudad en formato ancho
temperaturas_ancho ← tibble(
  ciudad = c("Buenos Aires", "Córdoba", "Mendoza", "Bariloche"),
  ene = c(28, 30, 31, 17),
  feb = c(27, 29, 30, 16),
  mar = c(25, 26, 27, 13)
)

temperaturas_ancho
```

```
## # A tibble: 4 × 4
##   ciudad          ene   feb   mar
##   <chr>         <dbl> <dbl> <dbl>
## 1 Buenos Aires     28    27    25
## 2 Córdoba          30    29    26
## 3 Mendoza          31    30    27
## 4 Bariloche        17    16    13
```

pivot_longer(): De Ancho a Largo

Usamos `pivot_longer()` para convertir de formato ancho a largo:

```
# Transformar a formato largo
temperaturas_largo ← temperaturas_ancho %>%
  pivot_longer(
    cols = c(ene, feb, mar),  # Columnas a transformar
    names_to = "mes",         # Nueva columna para los nombres
    values_to = "temperatura" # Nueva columna para los valores
  )

head(temperaturas_largo,6)
```

```
## # A tibble: 6 × 3
##   ciudad       mes   temperatura
##   <chr>        <chr>       <dbl>
## 1 Buenos Aires ene            28
## 2 Buenos Aires feb            27
## 3 Buenos Aires mar            25
## 4 Córdoba      ene            30
## 5 Córdoba      feb            29
## 6 Córdoba      mar            26
```

Resultado de pivot_longer()

Ahora los datos están en formato "tidy":

- Cada variable (ciudad, mes, temperatura) es una columna
- Cada observación (combinación ciudad-mes) es una fila

Ventajas de este formato:

- Facilita filtrado, agrupación y visualización
- Compatible con la mayoría de funciones de análisis
- Mejor para análisis estadístico y machine learning

pivot_wider(): De Largo a Ancho

Podemos volver al formato original con `pivot_wider()`:

```
# Volver al formato ancho
temperaturas_largo %>%
  pivot_wider(
    names_from = mes,        # Columna que contiene los nombres
    values_from = temperatura # Columna que contiene los valores
  )
```

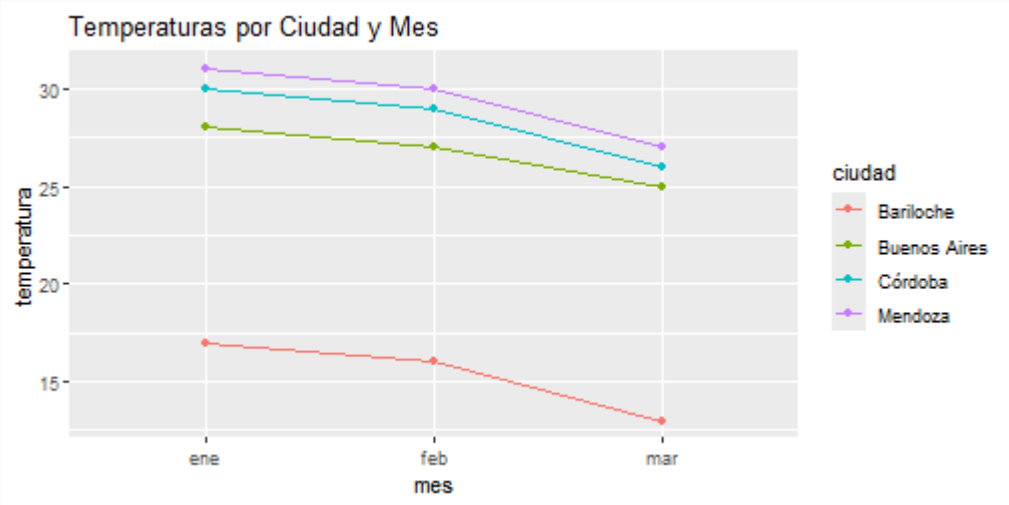
```
## # A tibble: 4 × 4
##   ciudad          ene   feb   mar
##   <chr>         <dbl> <dbl> <dbl>
## 1 Buenos Aires     28    27    25
## 2 Córdoba          30    29    26
## 3 Mendoza          31    30    27
## 4 Bariloche        17    16    13
```

Casos de Uso para pivot_longer()

1. **Análisis estadístico**:
    - Mayoría de pruebas estadísticas requieren datos en formato largo
2. **Visualización con ggplot2**:
    - Facilita gráficos con múltiples series y facetas

Casos de Uso para pivot_longer()

```
# Visualización con datos en formato largo
temperaturas_largo %>%
  ggplot(aes(x = mes, y = temperatura, group = ciudad, color = ciudad)) +
  geom_line() +
  geom_point() +
  labs(title = "Temperaturas por Ciudad y Mes")
```



Casos de Uso para pivot_wider()

1. **Presentación de resultados**:
    - Tablas más compactas y legibles para reportes
2. **Cálculos entre columnas**:
    - Cuando necesitamos comparar o realizar operaciones entre columnas
3. **Exportación a hojas de cálculo**:
    - Formato familiar para usuarios de Excel

Casos de Uso para pivot_wider()

```
# Ejemplo: calcular diferencia de temperatura entre enero y marzo
temperaturas_ancho %>%
  mutate(diferencia_ene_mar = ene - mar) %>%
  arrange(desc(diferencia_ene_mar))
```

```
## # A tibble: 4 × 5
##   ciudad         ene   feb   mar diferencia_ene_mar
##   <chr>        <dbl> <dbl> <dbl>               <dbl>
## 1 Córdoba         30    29    26                   4
## 2 Mendoza         31    30    27                   4
## 3 Bariloche       17    16    13                   4
## 4 Buenos Aires    28    27    25                   3
```

Casos de Uso para pivot_wider()

```
# Ejemplo: calcular diferencia de temperatura entre enero y marzo
temperaturas_ancho %>%
  mutate(diferencia_ene_mar = ene - mar) %>%
  arrange(desc(diferencia_ene_mar))
```

```
## # A tibble: 4 × 5
##   ciudad         ene   feb   mar diferencia_ene_mar
##   <chr>        <dbl> <dbl> <dbl>               <dbl>
## 1 Córdoba         30    29    26                   4
## 2 Mendoza         31    30    27                   4
## 3 Bariloche       17    16    13                   4
## 4 Buenos Aires    28    27    25                   3
```

pivot_longer() Avanzado

Opciones adicionales para situaciones más complejas:

```r
# Datos de ventas por trimestre y año
ventas ← tibble(
  producto = c("A", "B", "C"),
  `2023_Q1` = c(100, 200, 150),
  `2023_Q2` = c(120, 230, 140),
  `2024_Q1` = c(110, 210, 160),
  `2024_Q2` = c(130, 240, 170)
)

# Separar año y trimestre en columnas distintas
tmp ← ventas %>%
  pivot_longer(
    cols = -producto,
    names_to = c("año", "trimestre"),
    names_sep = "_",
    values_to = "ventas"
  )
```

pivot_longer() Avanzado

Opciones adicionales para situaciones más complejas:

```
# Separar año y trimestre en columnas distintas
ventas %>%
  pivot_longer(
    cols = -producto,
    names_to = c("año", "trimestre"),
    names_sep = "_",
    values_to = "ventas"
  )
```

```
## # A tibble: 12 × 4
##    producto año   trimestre ventas
##    <chr>    <chr> <chr>      <dbl>
##  1 A        2023  Q1           100
##  2 A        2023  Q2           120
##  3 A        2024  Q1           110
##  4 A        2024  Q2           130
##  5 B        2023  Q1           200
##  6 B        2023  Q2           230
##  7 B        2024  Q1           210
##  8 B        2024  Q2           240
##  9 C        2023  Q1           150
## 10 C        2023  Q2           140
```

Más Opciones de pivot_longer()

- **names_prefix**: eliminar prefijos de los nombres de columna
- **names_pattern**: extraer con expresiones regulares
- **values_drop_na**: eliminar filas con valores NA
- **names_ptypes**: especificar tipos de las nuevas columnas

```
# Ejemplo con values_drop_na y names_prefix
ventas_con_na ← tibble(
  producto = c("A", "B", "C"),
  año_2023 = c(100, 200, NA),
  año_2024 = c(110, NA, 160)
)

ventas_con_na %>%
  pivot_longer(
    cols = starts_with("año_"),
    names_to = "año",
    names_prefix = "año_",
    values_to = "ventas",
    values_drop_na = TRUE
  )
```

```
## # A tibble: 4 × 3
##   producto año   ventas
```

Más Opciones de pivot_longer()

```
ventas_con_na %>%
  pivot_longer(
    cols = starts_with("año_"),
    names_to = "año",
    names_prefix = "año_",
    values_to = "ventas",
    values_drop_na = TRUE
  )
```

```
## # A tibble: 4 × 3
##   producto año   ventas
##   <chr>    <chr>  <dbl>
## 1 A        2023     100
## 2 A        2024     110
## 3 B        2023     200
## 4 C        2024     160
```

Funciones Adicionales del Tidyverse

Exploremos otras funciones útiles que complementan lo visto hasta ahora:

1. **Funciones de transformación de conjuntos**
2. **Funciones para trabajar con fechas**
3. **Manejo de valores ausentes**
4. **Programación funcional con purrr**

Transformación de Conjuntos

Funciones que operan sobre conjuntos de datos como si fueran conjuntos matemáticos:

```
# Creamos dos dataframes
set_a ← tibble(
  x = 1:5,
  y = letters[1:5]
)

set_b ← tibble(
  x = 3:7,
  y = letters[3:7]
)

# Unión: filas de ambos conjuntos (sin duplicados)
bind_rows(set_a, set_b) %>% distinct()
```

```
## # A tibble: 7 × 2
##       x y
##   <int> <chr>
## 1     1 a
## 2     2 b
## 3     3 c
## 4     4 d
## 5     5 e
```

Transformación de Conjuntos

```
# Intersección: filas comunes a ambos conjuntos
intersect(set_a, set_b)
```

```
## # A tibble: 3 × 2
##       x y
##   <int> <chr>
## 1     3 c
## 2     4 d
## 3     5 e
```

```
# Diferencia: filas en A pero no en B
setdiff(set_a, set_b)
```

```
## # A tibble: 2 × 2
##       x y
##   <int> <chr>
## 1     1 a
## 2     2 b
```

Operaciones de Unión de Filas y

Funciones `bind_rows()` y `bind_cols()`:

```
# Unir filas (preservando columnas)
bind_rows(
  tibble(x = 1:3, y = letters[1:3]),
  tibble(x = 4:6, y = letters[4:6])
)
```

```
## # A tibble: 6 × 2
##       x y
##   <int> <chr>
## 1     1 a
## 2     2 b
## 3     3 c
## 4     4 d
## 5     5 e
## 6     6 f
```

Operaciones de Unión de Filas y

```r
# Unir columnas (deben tener mismo número de filas)
bind_cols(
  tibble(x = 1:3),
  tibble(y = letters[1:3])
)
```

```
## # A tibble: 3 × 2
##       x y
##   <int> <chr>
## 1     1 a
## 2     2 b
## 3     3 c
```

Nota: `bind_cols()` debe usarse con precaución, ya que no verifica la correspondencia de filas.

Trabajando con Fechas: lubridate

El paquete **lubridate** facilita enormemente el manejo de fechas:

```
library(lubridate)

# Fechas en diferentes formatos
fechas ← tibble(
  fecha1 = c("2025-03-15", "2025-04-20", "2025-05-25"),
  fecha2 = c("15/03/2025", "20/04/2025", "25/05/2025"),
  fecha3 = c("Mar 15, 2025", "Apr 20, 2025", "May 25, 2025")
)
```

Trabajando con Fechas: lubridate

```
# Convertir a objetos fecha
fechas_convertidas ← fechas %>%
  mutate(
    fecha1_date = ymd(fecha1),
    fecha2_date = dmy(fecha2),
    fecha3_date = mdy(fecha3)
  )

head(fechas_convertidas)
```

```
## # A tibble: 3 × 6
##   fecha1     fecha2     fecha3       fecha1_date fecha2_date fecha3_date
##   <chr>      <chr>      <chr>        <date>      <date>      <date>
## 1 2025-03-15 15/03/2025 Mar 15, 2025 2025-03-15  2025-03-15  2025-03-15
## 2 2025-04-20 20/04/2025 Apr 20, 2025 2025-04-20  2025-04-20  2025-04-20
## 3 2025-05-25 25/05/2025 May 25, 2025 2025-05-25  2025-05-25  2025-05-25
```

Operaciones con Fechas en lubridate

```
fechas_mutadas ← fechas_convertidas %>%
  mutate(
    año = year(fecha1_date),
    mes = month(fecha1_date, label = TRUE),
    día = day(fecha1_date),
    día_semana = wday(fecha1_date, label = TRUE),
    fecha_futura = fecha1_date + months(3)
  ) %>%
  select(fecha1, año, mes, día, día_semana, fecha_futura)

head(fechas_mutadas)
```

```
## # A tibble: 3 × 6
##   fecha1       año mes     día día_semana fecha_futura
##   <chr>      <dbl> <ord> <int> <ord>      <date>
## 1 2025-03-15  2025 mar      15 sáb        2025-06-15
## 2 2025-04-20  2025 abr      20 dom        2025-07-20
## 3 2025-05-25  2025 may      25 dom        2025-08-25
```

Manejo de Valores Ausentes (NA)

Funciones útiles para trabajar con valores ausentes:

```
# Datos con valores ausentes
datos_na ← tibble(
  id = 1:5,
  valor1 = c(10, NA, 30, NA, 50),
  valor2 = c(NA, 20, 30, 40, NA)
)

# Detectar NA
datos_na %>%
  mutate(
    tiene_na_valor1 = is.na(valor1),
    tiene_na_valor2 = is.na(valor2),
    tiene_algun_na = is.na(valor1) | is.na(valor2)
  )
```

```
## # A tibble: 5 × 6
##      id valor1 valor2 tiene_na_valor1 tiene_na_valor2 tiene_algun_na
##   <int>  <dbl>  <dbl> <lgl>           <lgl>           <lgl>
## 1     1     10     NA FALSE           TRUE            TRUE
## 2     2     NA     20 TRUE            FALSE           TRUE
## 3     3     30     30 FALSE           FALSE           FALSE
## 4     4     NA     40 TRUE            FALSE           TRUE
```

Manejo de Valores Ausentes (NA)

```
# Filtrar filas con/sin NA
datos_na %>%
  filter(!is.na(valor1))  # Solo filas donde valor1 no es NA
```

```
## # A tibble: 3 × 3
##      id valor1 valor2
##   <int>  <dbl>  <dbl>
## 1     1     10     NA
## 2     3     30     30
## 3     5     50     NA
```

```
# Reemplazar NA
datos_na %>%
  mutate(
    valor1_fill = replace_na(valor1, 0),
    valor2_fill = if_else(is.na(valor2), valor1 * 2, valor2)
  )
```

```
## # A tibble: 5 × 5
##      id valor1 valor2 valor1_fill valor2_fill
##   <int>  <dbl>  <dbl>       <dbl>       <dbl>
## 1     1     10     NA          10          20
## 2     2     NA     20           0          20
```

Hasta acá hoy

El resto es para que chusmeen. Algunas de estas cosas las vamos a ver en unas clases, pero ya la pueden ir buscando en google

Programación Funcional con purrr

purrr permite aplicar funciones de manera elegante:

```
# Lista de vectores
lista_numeros ← list(
  a = 1:5,
  b = 10:15,
  c = 20:25
)

# map(): aplicar una función a cada elemento
map_dbl(lista_numeros, mean)  # Devuelve vector numérico
```

```
##    a    b    c
##  3.0 12.5 22.5
```

```
# Diferentes tipos de salida
map_chr(lista_numeros, function(x) paste(x, collapse = ", "))
```

```
##                          a                          b                          c
##          "1, 2, 3, 4, 5" "10, 11, 12, 13, 14, 15" "20, 21, 22, 23, 24, 25"
```

Programación Funcional con purrr

```
# Abreviaciones de función con ~
map_dbl(lista_numeros, ~ sum(.x) / length(.x))
```

```
##    a    b    c
##  3.0 12.5 22.5
```

```
# map_if: aplicar función solo si cumple una condición
map_if(lista_numeros,
       .p = ~ mean(.x) > 15,  # Predicado
       .f = ~ .x * 2,         # Función si verdadero
       .else = ~ .x)          # Función si falso
```

```
## $a
## [1] 1 2 3 4 5
##
## $b
## [1] 10 11 12 13 14 15
##
## $c
## [1] 40 42 44 46 48 50
```

Funciones Especializadas en dplyr

Otras funciones útiles en dplyr:

```
# Casos condicionales con case_when()
mtcars %>%
  as_tibble(rownames = "modelo") %>%
  mutate(
    tamaño_motor = case_when(
      disp < 100 ~ "Pequeño",
      disp < 300 ~ "Mediano",
      TRUE ~ "Grande"
    )
  ) %>%
  count(tamaño_motor)
```

```
## # A tibble: 3 × 2
##   tamaño_motor     n
##   <chr>        <int>
## 1 Grande          11
## 2 Mediano         16
## 3 Pequeño          5
```

Funciones Especializadas en dplyr

```r
# row_number(), min_rank(), dense_rank(), etc.
mtcars %>%
  as_tibble(rownames = "modelo") %>%
  select(modelo, mpg) %>%
  mutate(
    posición = row_number(desc(mpg)),
    empates_min = min_rank(desc(mpg)),
    empates_dense = dense_rank(desc(mpg))
  ) %>%
  arrange(posición) %>%
  head(5)
```

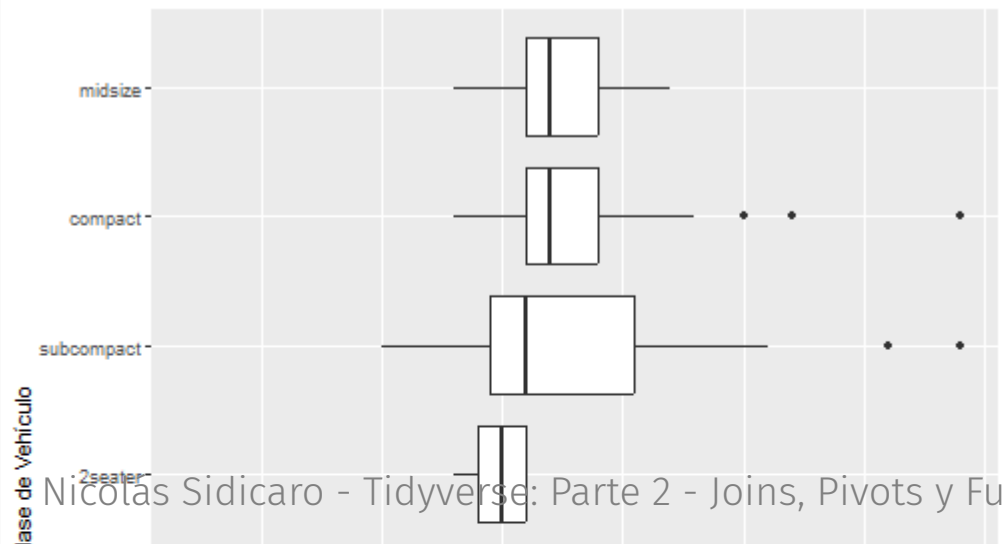
```
## # A tibble: 5 × 5
##   modelo           mpg posición empates_min empates_dense
##   <chr>          <dbl>    <int>       <int>         <int>
## 1 Toyota Corolla  33.9        1           1             1
## 2 Fiat 128        32.4        2           2             2
## 3 Honda Civic     30.4        3           3             3
## 4 Lotus Europa    30.4        4           3             3
## 5 Fiat X1-9       27.3        5           5             4
```

Otras Funciones Útiles

- **forcats** para manipular factores:
  - `fct_reorder()`: reordenar niveles de factor
  - `fct_lump()`: agrupar categorías poco frecuentes

```
library(forcats)

# Reordenar factores
ggplot(mpg, aes(x = fct_reorder(class, hwy, .fun = median), y = hwy)) +
  geom_boxplot() +
  coord_flip() +
  labs(x = "Clase de Vehículo", y = "Millas por Galón (Autopista)")
```



¡Gracias!

Próxima clase: Expresiones regulares y loops en R