Revisão: Aulas 01 - 04

ME315

Benilton S Carvalho

Departamento de Estatística / UNICAMP

Dados Tabulares

Arquivos Tabulares

- Arquivos tabulares têm forma retangular;
- Exemplo clássico de arquivo tabular: planilha Excel;
- Podem ser apresentados ao usuário/analista em diferentes versões;
- Sugestão para realização de análises:
  - Importação dos dados feita cautelosamente;
  - Assim, minimiza-se trabalhos posteriores na formatação dos dados;
- Colunas costumam representar variáveis e linhas, observações;

Formato Tidy

- Anteriormente, i.e. no SAS, conhecido como formato longo;
- É o melhor formato para análises estatísticas;
- Pode não ser o formato mais compacto, mas é o mais versátil;
- Métodos comumente implementados para ciência de dados costumam utilizar como entrada dados no formato tidy;

Formato Tidy

- Cada linha é uma única observação;
- Cada coluna é o nome de uma variável;
- Cada célula é um valor;

Produto	Dia	Valor
Gasolina	Segunda	4.19
Gasolina	Terça	4.19
Gasolina	Quarta	4.09
Etanol	Segunda	3.39
Etanol	Terça	3.39
Etanol	Quarta	3.09

Formato não-tidy

- Nomes de colunas possuem o valor de uma variável;

Produto	Segunda	Terça	Quarta
Gasolina	4.19	4.19	4.09
Etanol	3.39	3.39	3.09

Formato não-tidy

- Valores em uma coluna correspondem a duas variáveis;
- Uma célula pode corresponder a mais de um valor;

Produto-dia	Valor
Gasolina-Segunda	4.19
Gasolina-Terça	4.19
Gasolina-Quarta	4.09
Etanol-Segunda	3.39
Etanol-Terça	3.39
Etanol-Quarta	3.09

Arquivos Delimitados (CSV)

- Arquivo no formato texto;
- Cabeçalho opcional;
- Separador (*) é vírgula;
- Separador decimal deve ser diferente de vírgula (por exemplo, ponto)
- Será problemático em países que utilizam a vírgula como separador decimal;

```
Produto,Dia,Valor
Gasolina,Segunda,4.19
Gasolina,Terça,4.19
Gasolina,Quarta,4.09
Etanol,Segunda,3.39
Etanol,Terça,3.39
Etanol,Quarta,3.09
```

Importação por Partes e de Colunas Específicas

Registros de Vôos nos EUA

- Possui mais de 5 milhões de observações e 31 variáveis;
- Ocupa 1GB de RAM;
- Entre as variáveis:
    - Dia, mês, ano, dia da semana;
    - Cia aérea, número do vôo, registro do avião;
    - Aeroportos de partida e de chegada;
    - Horários de partida e chegada (reais e programados);
    - Tempo de vôo e distância voada;
    - Atraso na chegada.

Importação do Arquivo de Vôos

```
library(tidyverse)
in1 = read_csv('../dados/flights.csv.zip')
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   AIRLINE = col_character(),
##   TAIL_NUMBER = col_character(),
##   ORIGIN_AIRPORT = col_character(),
##   DESTINATION_AIRPORT = col_character(),
##   SCHEDULED_DEPARTURE = col_character(),
##   DEPARTURE_TIME = col_character(),
##   WHEELS_OFF = col_character(),
##   WHEELS_ON = col_character(),
##   SCHEDULED_ARRIVAL = col_character(),
##   ARRIVAL_TIME = col_character(),
##   CANCELLATION_REASON = col_character()
## )

## See spec(...) for full column specifications.
```

Manipulação de Tempo e Distância

```
in1 %>%
  filter(!is.na(AIR_TIME), !is.na(DISTANCE)) %>%
  mutate(AIR_TIME=AIR_TIME/60, DISTANCE=DISTANCE*1.6) %>%
  select(AIR_TIME, DISTANCE)
```

```
## # A tibble: 5,714,008 x 2
##    AIR_TIME DISTANCE
##       <dbl>    <dbl>
##  1     2.82    2317.
##  2     4.38    3728
##  3     4.43    3674.
##  4     4.3     3747.
##  5     3.32    2317.
##  6     3.43    2542.
##  7     2.57    2078.
##  8     3.8     3400
##  9     2.88    2342.
## 10     3.1     2795.
## # … with 5,713,998 more rows
```

Reta de Regressão

$$Distancia_i = b_0 + b_1 \times Tempo_i + \epsilon_i$$

Estimadores via Mínimos Quadrados

$$
\begin{aligned}
y_i &= b_0 + b_1 x_i + \epsilon_i \\
\hat{b}_0 &= \bar{y} - \hat{b}_1 \bar{x} \\
\hat{b}_1 &= \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_i (x_i - \bar{x})^2} \\
&= \frac{\sum x_i y_i - \frac{\sum x_i \sum y_i}{n}}{\sum x_i^2 - \frac{(\sum x_i)^2}{n}}
\end{aligned}
$$

- Precisamos pensar em meios de calcular estas estatísticas usando apenas partes do conjunto de dados;
- Estas estatísticas "parciais" devem poder ser combinadas;
- Estatísticas suficientes:
  - $\sum x_i$;
  - $\sum y_i$;
  - $\sum x_i y_i$;
  - $\sum x_i^2$;
  - n;

Particionando operações

$$\sum x_i y_i = \sum_{i=1}^{k_1} x_i y_i + \sum_{i=k_1+1}^{k_2} x_i y_i + \cdots$$

$$\sum x_i = \sum_{i=1}^{k_1} x_i + \sum_{i=k_1+1}^{k_2} x_i + \cdots$$

$$\sum x_i^2 = \sum_{i=1}^{k_1} x_i^2 + \sum_{i=k_1+1}^{k_2} x_i^2 + \cdots$$

Para um bloco dos dados

$$\hat{b}_0 = \bar{y} - \hat{b}_1 \bar{x}$$

$$\hat{b}_1 = \frac{\sum x_i y_i - \frac{\sum x_i \sum y_i}{n}}{\sum x_i^2 - \frac{(\sum x_i)^2}{n}}$$

```
getStats = function(input, pos){
  input %>% filter(!is.na(AIR_TIME), !is.na(DISTANCE)) %>%
    mutate(AIR_TIME=AIR_TIME/60, DISTANCE=DISTANCE*1.6) %>%
    summarise(Sxy=sum(AIR_TIME*DISTANCE), Sx=sum(AIR_TIME),
              Sy=sum(DISTANCE), Sx2=sum(AIR_TIME^2), n=n())
}
computeStats = function(stats){
  stats %>%
    summarise(num = sum(Sxy)-(sum(Sx)*sum(Sy))/sum(n),
              den = sum(Sx2)-(sum(Sx)^2)/sum(n),
              b1 = num/den,
              b0 = sum(Sy)/sum(n)-b1*sum(Sx)/sum(n)) %>%
    select(b0, b1) %>% gather(key='coef', value='valor') %>%
    knitr::kable('html')
}
```

Para um bloco dos dados

```
in1 %>% getStats() %>% computeStats()
```

coef	valor
b0	-189.3290
b1	797.3421

Processando Dados em Lote

- O pacote `readr` possui funções de importação aprimoradas;
- São funções mais rápidas e inteligentes;
- Uma classe de funções é a de operação em porções de arquivos:
  - `read_csv_chunked`;
  - `read_csv2_chunked`;
  - `read_delim_chunked`;
  - `read_tsv_chunked`;
- As funções `read_***_chunked` aceitam argumentos especiais:
  - `chunk_size`: número de linhas a serem importadas por iteração;
  - `callback`: função que é executada em cada porção dos dados;
- O argumento `callback` deve instanciar:
  - `DataFrameCallback`: se se deseja combinar resultados tabulares;
  - `ListCallback`: se se deseja combinar resultados 'flexíveis';
  - `SideEffectChunkCallback`: se se deseja visualizar efeitos colaterais.

Importação de Dados com Leitura em Lotes

```
in2 = read_csv_chunked('../dados/flights.csv.zip',
                       callback=DataFrameCallback$new(getStats),
                       chunk_size = 1e6)
```

```
## Parsed with column specification:
## cols(
##   .default = col_double(),
##   AIRLINE = col_character(),
##   TAIL_NUMBER = col_character(),
##   ORIGIN_AIRPORT = col_character(),
##   DESTINATION_AIRPORT = col_character(),
##   SCHEDULED_DEPARTURE = col_character(),
##   DEPARTURE_TIME = col_character(),
##   WHEELS_OFF = col_character(),
##   WHEELS_ON = col_character(),
##   SCHEDULED_ARRIVAL = col_character(),
##   ARRIVAL_TIME = col_character(),
##   CANCELLATION_REASON = col_character()
## )

## See spec(...) for full column specifications.
```

Importação de Dados com Leitura em Lotes

```
in2
```

```
## # A tibble: 6 x 5
##           Sxy       Sx          Sy      Sx2      n
##         <dbl>    <dbl>       <dbl>    <dbl>  <int>
## 1 3387192042. 1797787. 1234502685. 4748642. 957394
## 2 3565359207. 1872273. 1296212440  4951091. 989242
## 3 3670232911. 1873591  1311278674. 5025248. 981161
## 4 3689414436. 1871883. 1324030437. 4996242. 987786
## 5 3552818377. 1849173  1297736736  4850165. 993158
## 6 2997728461. 1545399. 1073764374. 4160835. 805267
```

```
in2 %>% computeStats()
```

coef	valor
b0	-189.3290
b1	797.3421

Importação de Dados - Colunas Específicas

- As funções de importação `read_***` possuem um argumento `col_types`;
- Opções válidas para `col_types`:
  - Especificação criada por `cols()`: todas as colunas;
  - Especificação criada por `cols_only()`: apenas um subconjunto;
- `cols()`: `cols(NOME=col_TIPO())`
  - `cols(a=col_integer());`
  - `cols(a='i')`
- `cols_only()`: `cols_only(NOME=col_TIPO())`
  - `cols_only(a=col_integer())`
  - `cols_only(a='i')`

```
mycols = cols_only(AIR_TIME='i', DISTANCE='i')
in3 = read_csv_chunked('../dados/flights.csv.zip',
                       callback=DataFrameCallback$new(getStats),
                       chunk_size = 1e6, col_types=mycols)
```

Importação de Dados - Colunas Específicas

```
in3
```

```
## # A tibble: 6 x 5
##           Sxy       Sx          Sy      Sx2      n
##         <dbl>    <dbl>       <dbl>    <dbl>  <int>
## 1 3387192042. 1797787. 1234502685. 4748642. 957394
## 2 3565359207. 1872273. 1296212440  4951091. 989242
## 3 3670232911. 1873591  1311278674. 5025248. 981161
## 4 3689414436. 1871883. 1324030437. 4996242. 987786
## 5 3552818377. 1849173  1297736736  4850165. 993158
## 6 2997728461. 1545399. 1073764374. 4160835. 805267
```

```
in3 %>% computeStats()
```

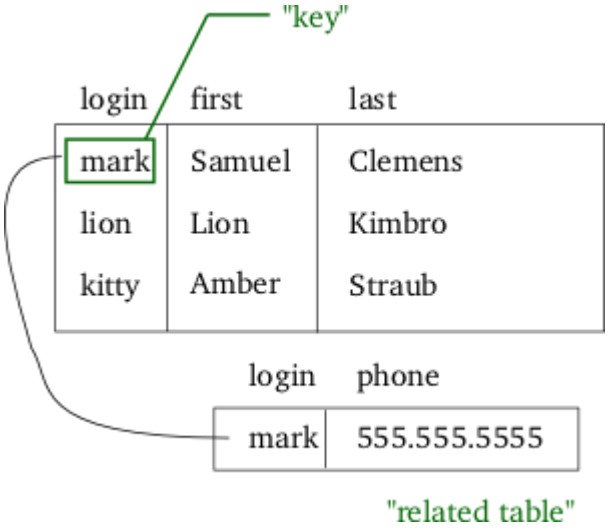
coef	valor
b0	-189.3290
b1	797.3421

Dados Relacionais

Dados em múltiplas tabelas

- É comum que dados estejam guardados em múltiplas tabelas. Esse modelo de banco de dados é conhecido como *Modelo Relacional* (https://en.wikipedia.org/wiki/Relational_model), em que os dados são acessados através de um *nome de tabela*, uma *chave* (*key*) e uma *coluna* (*features*).
- Se espera que, em no mínimo uma tabela, a chave identifique unicamente cada observação.

Exemplo



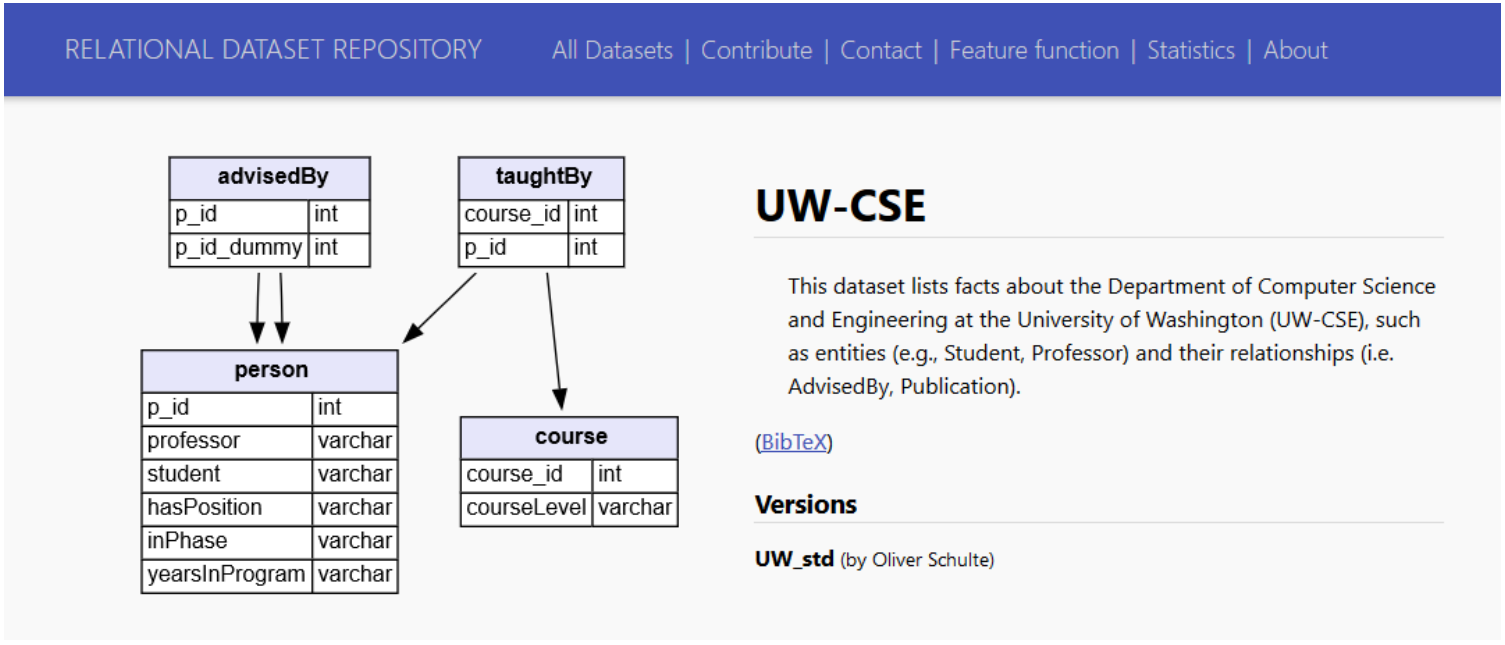
Exemplo de base relacional:

Figura de https://en.wikipedia.org/wiki/Relational_model

Consultas

- Cada tabela, separadamente, funciona como os bancos de dados com que trabalhamos até agora.
- Uma coluna em comum entre as tabelas será usada como chave, ligando a informação de cada linha. Porém, não há garantias que o valor seja único, nem sempre qual coluna servirá de chave é óbvio.
- Uma *consulta* (ou *query*) é um pedido do usuário ao *relational database management system* (RDBMS) que une informações de um grupo de indivíduos (baseados na chave) ao longo de várias tabelas.
- Nós vamos, primeiramente, examinar a operação *join*, do pacote `dplyr`, para realizar consultas em pares de tabelas.

Exemplo



Professores e alunos da University of Washington, ciência da computação.

Dados: https://relational.fit.cvut.cz/dataset/UW-CSE

Explicação: http://aiweb.cs.washington.edu/ai/mln/database.html

Tabelas não são 1-1

```
# Same course, different faculty
taughtBy %>% filter(course_id == 11)
```

```
##   course_id p_id
## 1        11   52
## 2        11   57
## 3        11  298
## 4        11  324
## 5        11  331
```

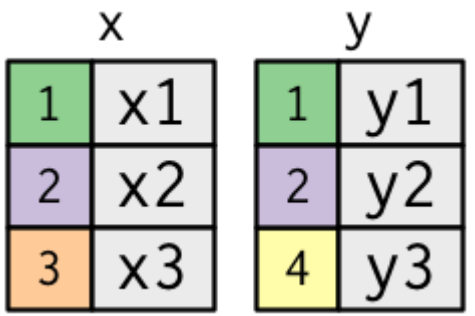
```
# Same faculty, different course
taughtBy %>% filter(p_id == 40)
```

```
##   course_id p_id
## 1         0   40
## 2         1   40
```

- `p_id` identifica indivíduos unicamente em `person`;
- `course_id` identifica cursos unicamente em `courses`.

Operações com Duas Tabelas

Usando os diagramas de Wickham and Grolemund (2017), considere dados de duas tabelas:



A coluna colorida é a chave, `x` e `y` são colunas, tomando valores `x1`, `x2`, etc.

Tipos de JOIN: setup 2

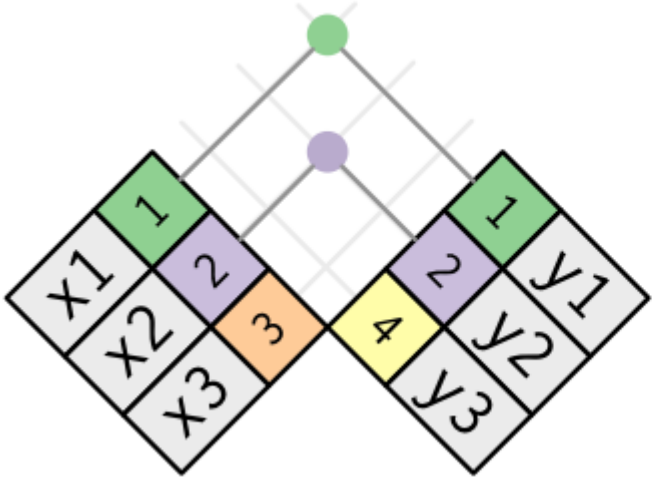
```
(x <- data.frame(key = c(1,2,3), val_x = c("x1","x2","x3")))
```

```
##   key val_x
## 1   1    x1
## 2   2    x2
## 3   3    x3
```

```
(y <- data.frame(key = c(1,2,4), val_y = c("y1","y2","y4")))
```

```
##   key val_y
## 1   1    y1
## 2   2    y2
## 3   4    y4
```

INNER JOIN: inner_join



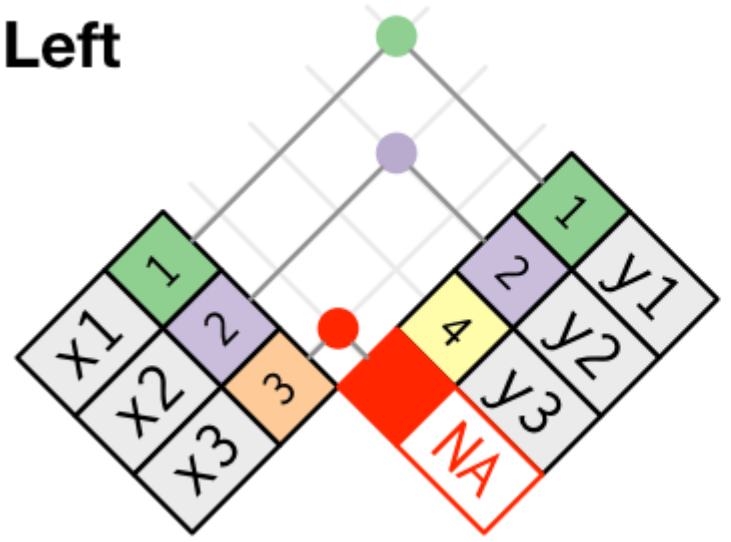
key	val_x	val_y
1	x1	y1
2	x2	y2

INNER JOIN: inner_join

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

```
##   key val_x val_y
## 1   1    x1    y1
## 2   2    x2    y2
```

OUTER JOIN: left_join



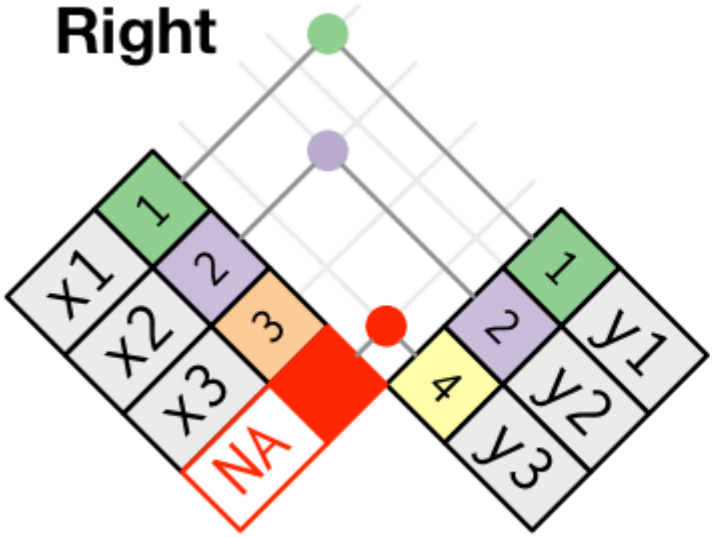
key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA

OUTER JOIN: left_join

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

```
##   key val_x val_y
## 1   1    x1    y1
## 2   2    x2    y2
## 3   3    x3  <NA>
```

OUTER JOIN: right_join



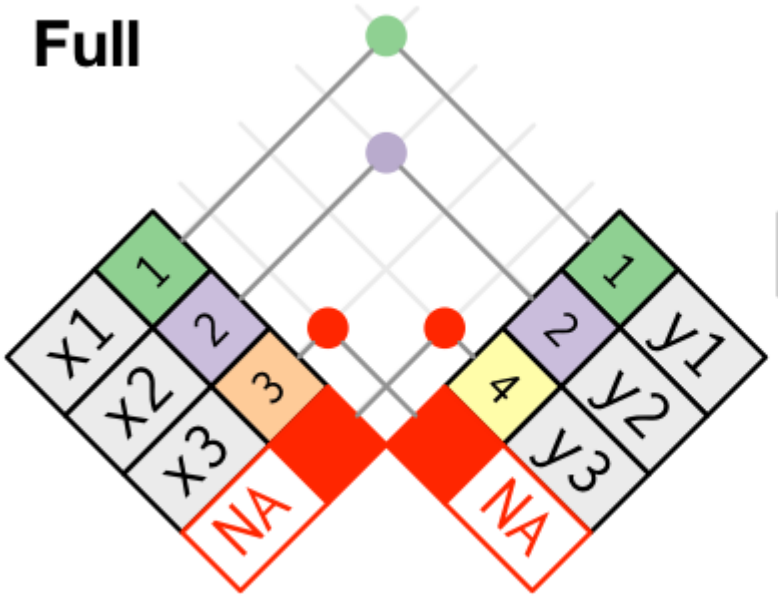
key	val_x	val_y
1	x1	y1
2	x2	y2
4	NA	y3

OUTER JOIN: right_join

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

```
##   key val_x val_y
## 1   1    x1    y1
## 2   2    x2    y2
## 3   4  <NA>    y4
```

OUTER JOIN: full_join



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

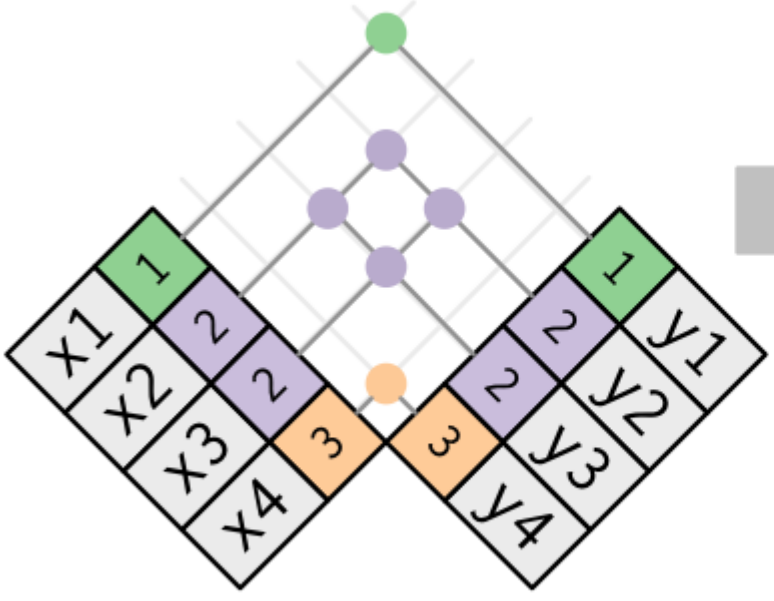
OUTER JOIN: full_join

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

```
##   key val_x val_y
## 1   1    x1    y1
## 2   2    x2    y2
## 3   3    x3  <NA>
## 4   4  <NA>    y4
```

Duplicated keys

Quando há mais de uma entrada para as duas tabelas, é executado um produto cartesiano das entradas.



key	val_x	val_y
1	x1	y1
2	x2	y2
2	x2	y3
2	x3	y2
2	x3	y3
3	x4	y4

Evite joins assim. Em tese, as bases relacionais devem ter pelo menos uma chave que unicamente determina as observações em cada tabela.

Sintaxe do parâmetro "by"

A ação padrão das funções `*_join(x, y)` no `dplyr` é `by = NULL`, que realiza o join pela combinação de *todas* as colunas com nomes idênticos em x e y. Isso pode ser perigoso!

```
x$newCol <- c(1, 1, 2)
y$newCol <- c(1, 2, 2)
full_join(x, y)
```

```
## Joining, by = c("key", "newCol")
```

```
##   key val_x newCol val_y
## 1   1    x1      1    y1
## 2   2    x2      1  <NA>
## 3   3    x3      2  <NA>
## 4   2  <NA>      2    y2
## 5   4  <NA>      2    y4
```

```
x$newCol <- NULL
y$newCol <- NULL
```

Sintaxe do parâmetro "by"

Já `by = "colName"` une as observações pelo "colName" especificado.

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

```
##   key val_x val_y
## 1   1    x1    y1
## 2   2    x2    y2
## 3   3    x3  <NA>
## 4   4  <NA>    y4
```

Caso você queira comparar diferentes colunas, a sintaxe é `by = c("colunaX" = "colunaY")`. Note que o `R` remove `key` de `y` sem avisar!

```
x$newKey <- c(1,4,2)
full_join(x, y, by = c("newKey" = "key"))
```

```
##   key val_x newKey val_y
## 1   1    x1      1    y1
## 2   2    x2      4    y4
## 3   3    x3      2    y2
```

Filtering joins

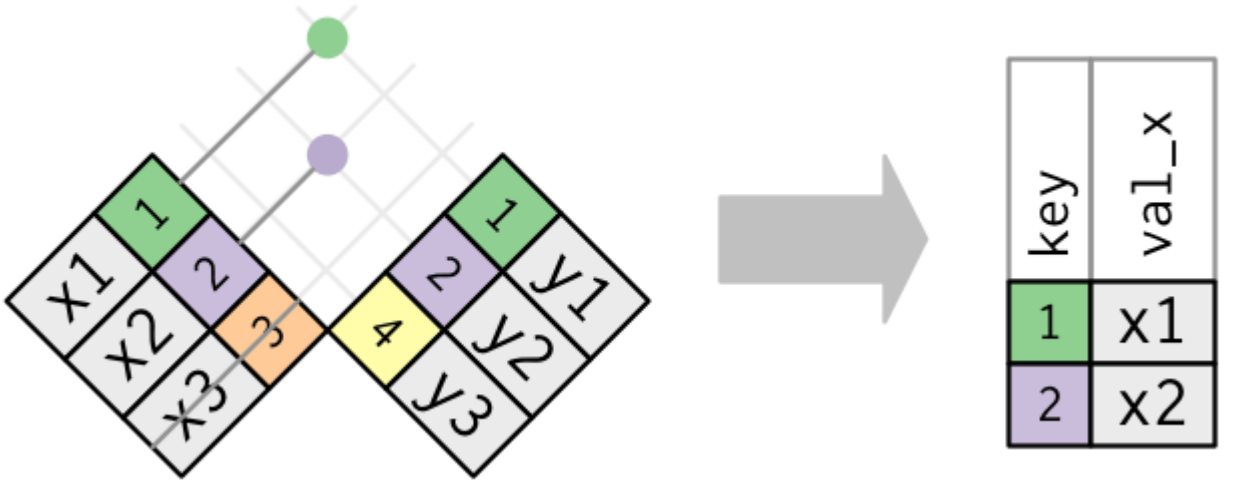
Há dois importantes filtering joins:

- `semi_join(x, y)` mantém todas as observações em `x` que estão presentes em `y`.
- `anti_join(x, y)` remove todas as observações em `x` que estão presentes em `y`.

Esses `*_join` retornam tabelas `x` filtradas, e não unem `x` e `y`.

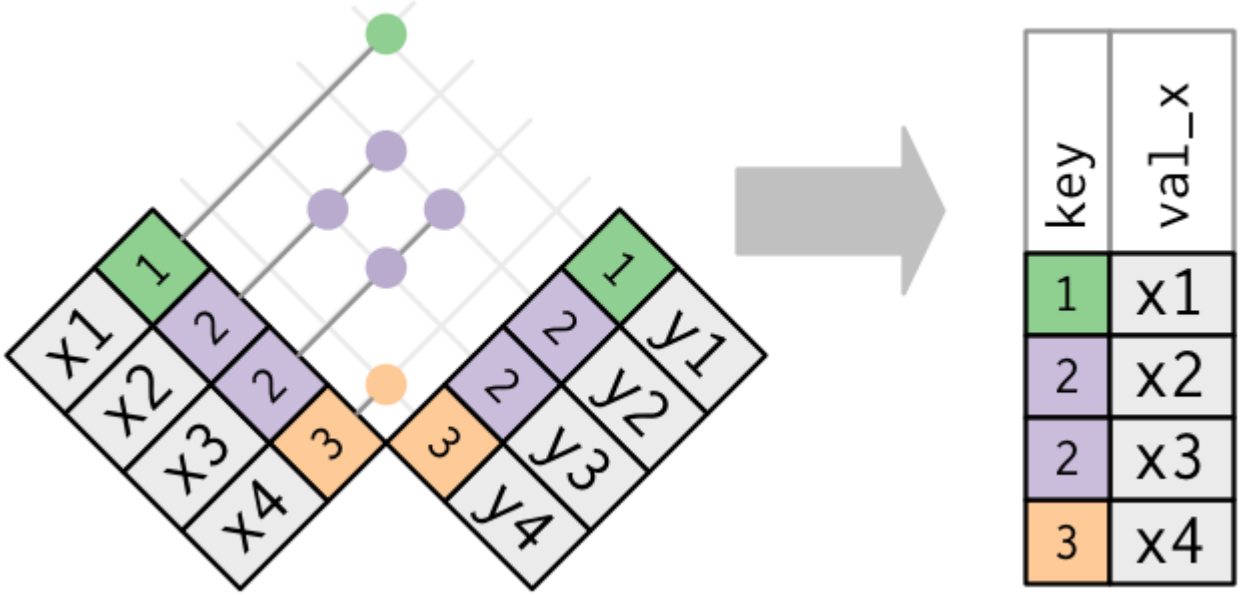
semi_join

`semi_join(x,y)` só retorna elementos de `x` que também estão em `y`



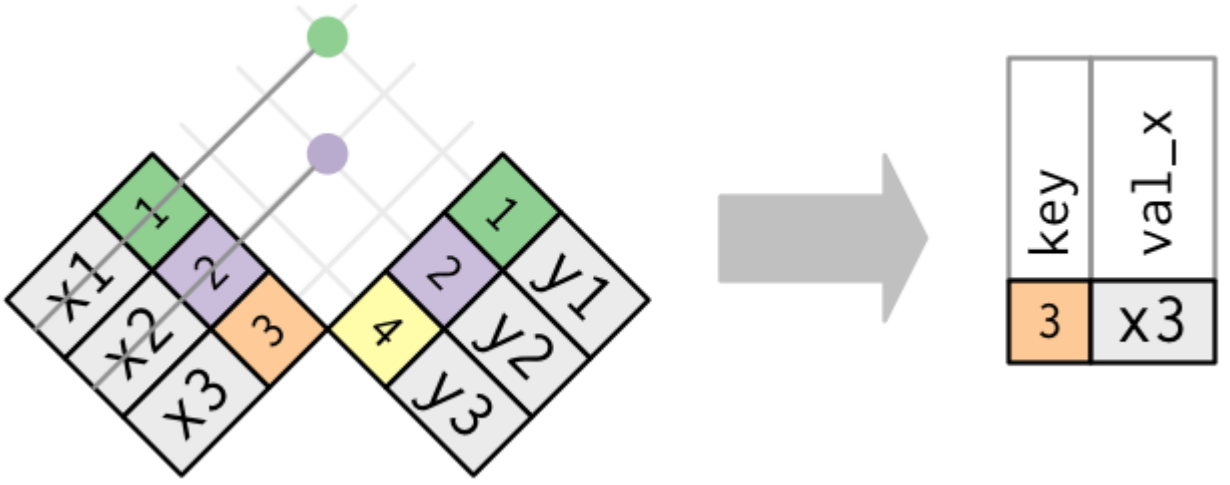
semi_join: duplicated keys

Não há problema se as chaves forem duplicadas para o `semi_join`, isto é, o `semi_join` não duplica as linhas.



anti_join

`anti_join(x,y)` só retorna elementos de `x` que **não** estão em `y`. É útil para detectar se há chaves faltantes em uma tabela.

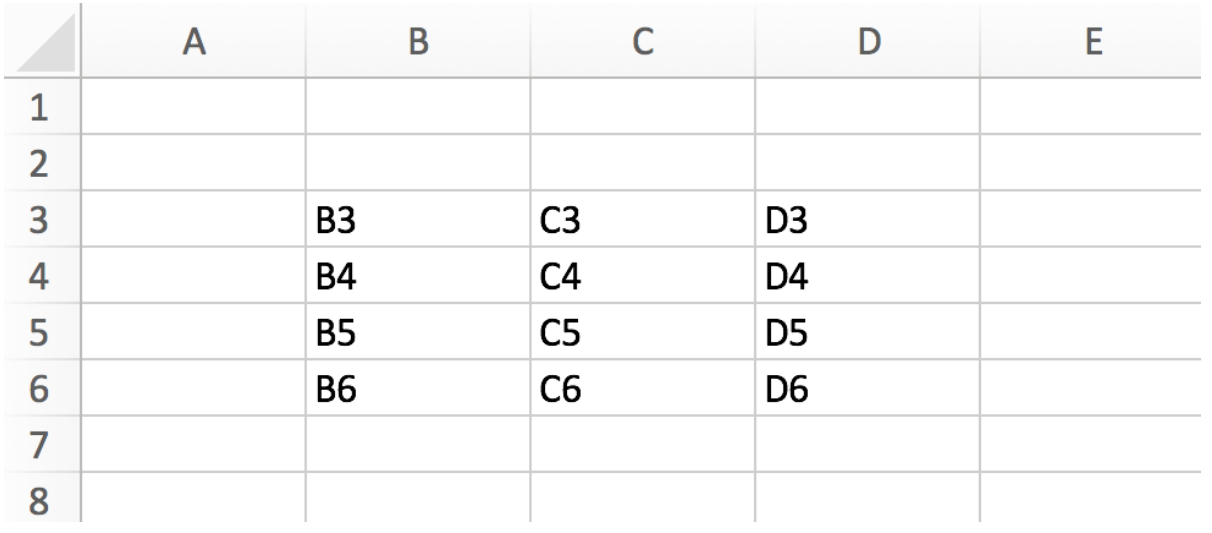


Importação de Arquivos Excel

Arquivos Excel e Pacote `readxl`

- Em um arquivo Excel:
  - Células que você vê podem não existir;
  - Células que você não vê podem existir;
- Pacote `readxl`:
  - Funciona em qualquer ambiente (Windows, Linux, Mac) sem dependências externas;
  - Permite a leitura de arquivos Excel (.xls ou .xlsx);
  - Lê apenas células com conteúdo;
  - Células vazias em colunas existentes são preenchidas com `NA`;
  - Resultados são `tibble`.

`readxl` busca identificar geometria



	A	B	C	D	E
1					
2					
3		B3	C3	D3	
4		B4	C4	D4	
5		B5	C5	D5	
6		B6	C6	D6	
7					
8					

```
library(readxl)
read_excel(readxl_example("geometry.xlsx"))
```

```
## # A tibble: 3 x 3
##   B3    C3    D3
##   <chr> <chr> <chr>
## 1 B4    C4    D4
## 2 B5    C5    D5
## 3 B6    C6    D6
```

`readr` pode ler faixas de dados

	A	B	C	D	E
1					
2					
3		B3	C3	D3	
4		B4	C4	D4	
5		B5	C5	D5	
6		B6	C6	D6	
7					
8					

`readr` e faixas de dados

```
library(readxl)
read_excel(readxl_example("geometry.xlsx"), range = "A2:C4")
```

```
## New names:
## * `` -> ...1
## * `` -> ...2
## * `` -> ...3
```

```
## # A tibble: 2 x 3
##   ...1  ...2  ...3
##   <lgl> <chr> <chr>
## 1 NA    B3    C3
## 2 NA    B4    C4
```

Leitura por partes com `readxl`

- `cell_rows()`
  - `cell_rows(1:10)`
- `cell_cols()`
  - `cell_cols(4:8)`
- `anchored()`
  - `anchored("C5", c(3, 4))`
- `cell_limits()`
  - `cell_limits(c(5, 3), c(NA, NA))`