Relational Data

Kylie Ariel Bemis

10/10/2017

Housekeeping

- Miniposter is due today on Piazza
- Homework 2 is due Friday (9/13) on Piazza
- The Midterm Exam is next Tuesday (9/17)
  - We will review Friday
  - Material covers everything through today
- Project groups (2-4 people) are due on the Midterm Exam
- Project guidelines will be posted on Piazza in next few days

Introduction to Relational Data

What is **relational data**?

Often data analysis involve multiple tables of related data.

The data analysis steps we need to do typically depend on the *relationships* between the different tables of data.

Each relationship is defined on a *pair* of tables.

We need data manipulation verbs that work on pairs of tables.

Introduction to Relational Data

`dplyr` provides three families of verbs for working with relational data:

- **Mutating joins** add new variables from one data frame to matching observations in another
- **Filtering joins** filter observations from one data frame based on matching observations in another
- **Set operations** treat observations (rows) as elements of a set

Most relational data live in *relational database management systems* (RDBMS) such as MySQL, SQLite, PostgreSQL, etc. and are accessed via Structured Query Language (SQL).

We will see that the names of `dplyr` verbs for relational data are heavily influenced by their SQL equivalents.

Relational Data in R

```
library(nycflights13)
```

The `nycflights13` package we have been using for examples and homework actually includes five related tables:

- `airlines` gives airline names based on their abbreviated code
- `airports` gives information about airports, identified by `faa` code
- `flights` gives the data on individual flights out of NYC airports
- `planes` gives information about each plane, identified by `tailnum`
- `weather` gives weather information for each NYC airport by hour

nycflights13 - airlines

```
airlines
```

```
## # A tibble: 16 x 2
##    carrier                          name
##      <chr>                         <chr>
##  1      9E           Endeavor Air Inc.
##  2      AA      American Airlines Inc.
##  3      AS        Alaska Airlines Inc.
##  4      B6             JetBlue Airways
##  5      DL        Delta Air Lines Inc.
##  6      EV    ExpressJet Airlines Inc.
##  7      F9      Frontier Airlines Inc.
##  8      FL AirTran Airways Corporation
##  9      HA      Hawaiian Airlines Inc.
## 10      MQ                   Envoy Air
## 11      OO       SkyWest Airlines Inc.
## 12      UA       United Air Lines Inc.
## 13      US             US Airways Inc.
## 14      VX              Virgin America
## 15      WN      Southwest Airlines Co.
## 16      YV          Mesa Airlines Inc.
```

nycflights13 - airports

```
airports
```

```
## # A tibble: 1,458 x 8
##      faa                           name      lat         lon <
##    <chr>                          <chr>    <dbl>       <dbl>
##  1   04G             Lansdowne Airport 41.13047  -80.61958
##  2   06A  Moton Field Municipal Airport 32.46057  -85.68003
##  3   06C        Schaumburg Regional 41.98934  -88.10124
##  4   06N            Randall Airport 41.43191  -74.39156
##  5   09J      Jekyll Island Airport 31.07447  -81.42778
##  6   0A9 Elizabethton Municipal Airport 36.37122  -82.17342
##  7   0G6        Williams County Airport 41.46731  -84.50678
##  8   0G7  Finger Lakes Regional Airport 42.88356  -76.78123
##  9   0P2   Shoestring Aviation Airfield 39.79482  -76.64719
## 10   0S9           Jefferson County Intl 48.05381 -122.81064
## # ... with 1,448 more rows, and 2 more variables: dst <chr>,
```

nycflights13 - flights

```
flights
```

```
## # A tibble: 336,776 x 19
##      year month   day dep_time sched_dep_time dep_delay arr_ti
##     <int> <int> <int>    <int>          <int>     <dbl>    <in
##  1  2013     1     1      517            515         2      8
##  2  2013     1     1      533            529         4      8
##  3  2013     1     1      542            540         2      9
##  4  2013     1     1      544            545        -1     10
##  5  2013     1     1      554            600        -6      8
##  6  2013     1     1      554            558        -4      7
##  7  2013     1     1      555            600        -5      9
##  8  2013     1     1      557            600        -3      7
##  9  2013     1     1      557            600        -3      8
## 10  2013     1     1      558            600        -2      7
## # ... with 336,766 more rows, and 12 more variables: sched_ar
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <ch
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

nycflights13 - planes

```
planes
```

```
## # A tibble: 3,322 x 9
##    tailnum  year                type     manufacturer
##      <chr> <int>               <chr>            <chr>
##  1  N10156  2004 Fixed wing multi engine          EMBRAER EMB
##  2  N102UW  1998 Fixed wing multi engine AIRBUS INDUSTRIE  A3
##  3  N103US  1999 Fixed wing multi engine AIRBUS INDUSTRIE  A3
##  4  N104UW  1999 Fixed wing multi engine AIRBUS INDUSTRIE  A3
##  5  N10575  2002 Fixed wing multi engine          EMBRAER EMB
##  6  N105UW  1999 Fixed wing multi engine AIRBUS INDUSTRIE  A3
##  7  N107US  1999 Fixed wing multi engine AIRBUS INDUSTRIE  A3
##  8  N108UW  1999 Fixed wing multi engine AIRBUS INDUSTRIE  A3
##  9  N109UW  1999 Fixed wing multi engine AIRBUS INDUSTRIE  A3
## 10  N110UW  1999 Fixed wing multi engine AIRBUS INDUSTRIE  A3
## # ... with 3,312 more rows, and 4 more variables: engines <in
## #   seats <int>, speed <int>, engine <chr>
```

nycflights13 - weather

```
weather
```

```
## # A tibble: 26,130 x 15
##    origin  year month   day  hour  temp  dewp humid wind_dir
##     <chr> <dbl> <dbl> <int> <int> <dbl> <dbl> <dbl>    <dbl>
##  1    EWR  2013     1     1     0 37.04 21.92 53.97      230
##  2    EWR  2013     1     1     1 37.04 21.92 53.97      230
##  3    EWR  2013     1     1     2 37.94 21.92 52.09      230
##  4    EWR  2013     1     1     3 37.94 23.00 54.51      230
##  5    EWR  2013     1     1     4 37.94 24.08 57.04      240
##  6    EWR  2013     1     1     6 39.02 26.06 59.37      270
##  7    EWR  2013     1     1     7 39.02 26.96 61.63      250
##  8    EWR  2013     1     1     8 39.02 28.04 64.43      240
##  9    EWR  2013     1     1     9 39.92 28.04 62.21      250
## 10    EWR  2013     1     1    10 39.02 28.04 64.43      260
## # ... with 26,120 more rows, and 5 more variables: wind_gust
## #   precip <dbl>, pressure <dbl>, visib <dbl>, time_hour <dtt
```

Relations in nycflights13

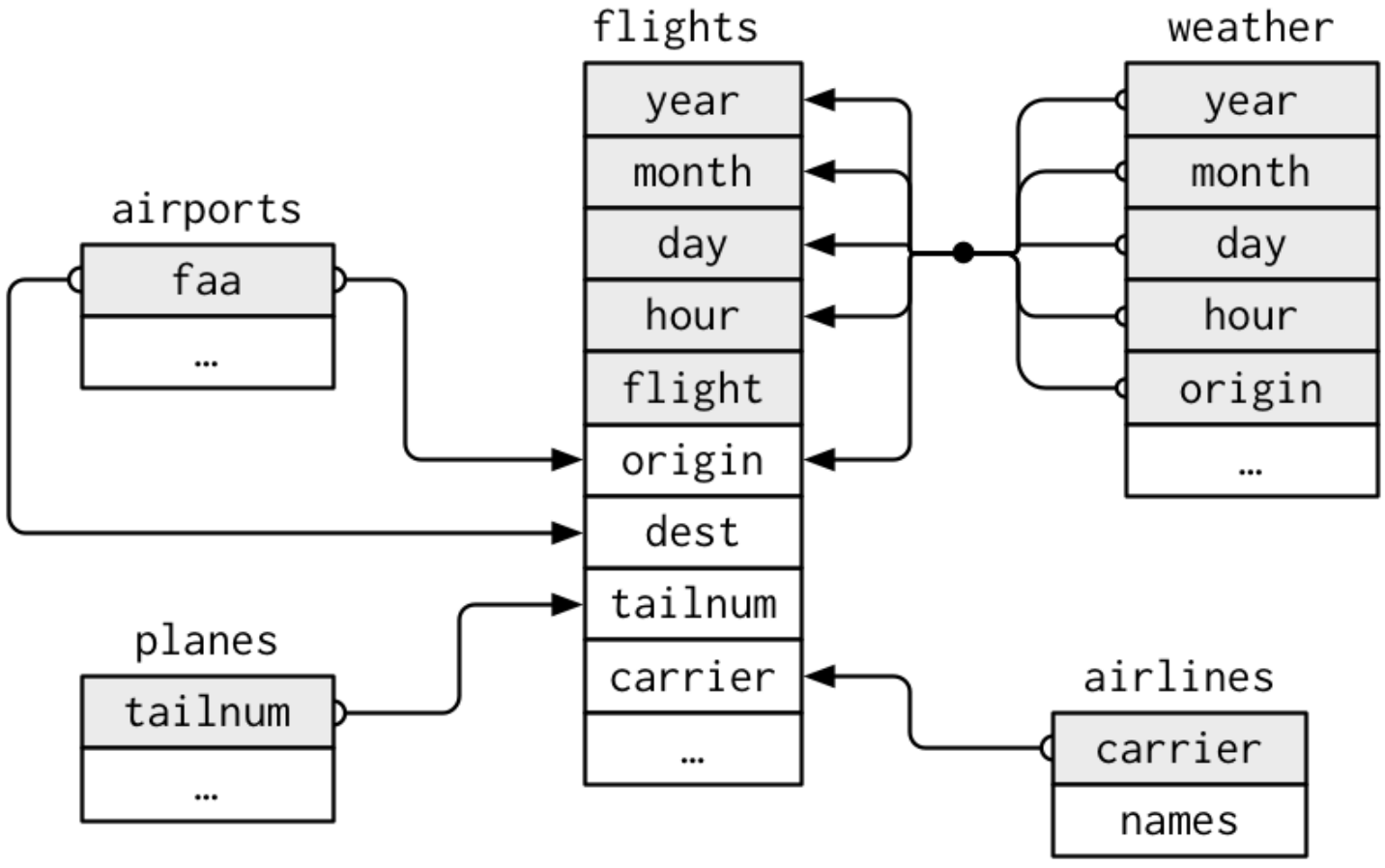


Figure 1: nycflights13

Keys

Variables used to connect tables are called **keys**.

A key may be a single variable, or multiple variables.

- A **primary key** uniquely identifies an observation in its own table
  - `planes` has a primary key of `tailnum`
  - `airports` has a primary key of `faa`
  - `weather` has a primary key of `(year, month, day, hour, origin)`
- A **foreign key** uniquely identifies an observation in another table
  - `flights` has a foreign key of `tailnum` for `planes`
  - `flights` has a foreign key of `carrier` for `airlines`
  - `flights` has a foreign key of `(year, month, day, hour, origin)` for `weather`

Checking primary keys

```
planes %>%
  count(tailnum) %>%
  filter(n > 1)
```

```
## # A tibble: 0 x 2
## # ... with 2 variables: tailnum <chr>, n <int>
```

Checking primary keys

```
weather %>%
  count(year, month, day, hour, origin) %>%
  filter(n > 1)
```

```
## # A tibble: 0 x 6
## # ... with 6 variables: year <dbl>, month <dbl>, day <int>, h
## #   origin <chr>, n <int>
```

What is the primary key of `flights`?

```
flights %>%
  count(year, month, day, flight) %>%
  filter(n > 1)
```

```
## # A tibble: 29,768 x 5
##     year month   day flight     n
##    <int> <int> <int>  <int> <int>
##  1  2013     1     1      1     2
##  2  2013     1     1      3     2
##  3  2013     1     1      4     2
##  4  2013     1     1     11     3
##  5  2013     1     1     15     2
##  6  2013     1     1     21     2
##  7  2013     1     1     27     4
##  8  2013     1     1     31     2
##  9  2013     1     1     32     2
## 10  2013     1     1     35     2
## # ... with 29,758 more rows
```

Add a primary key to `flights`

Called a **surrogate key**.

```
flights %>%
  mutate(row_id=row_number()) %>%
  select(row_id, year:dep_delay)
```

```
## # A tibble: 336,776 x 7
##    row_id  year month   day dep_time sched_dep_time dep_delay
##     <int> <int> <int> <int>    <int>          <int>     <dbl>
##  1      1  2013     1     1      517            515         2
##  2      2  2013     1     1      533            529         4
##  3      3  2013     1     1      542            540         2
##  4      4  2013     1     1      544            545        -1
##  5      5  2013     1     1      554            600        -6
##  6      6  2013     1     1      554            558        -4
##  7      7  2013     1     1      555            600        -5
##  8      8  2013     1     1      557            600        -3
##  9      9  2013     1     1      557            600        -3
## 10     10  2013     1     1      558            600        -2
## # ... with 336,766 more rows
```

Why are keys important?

- A primary key and a foreign key in another table form a **relation**
- These relations are typically "one-to-many"
  - `tailnum` in `planes` connecting to many flights in `flights`
  - `carrier` in `airlines` connecting to many flights in `flights`
- We typically use these relations to describe how we **join** tables
- But as we will see soon, we can explicitly specify which variables are used to join tables, so why else might keys be important?

How is relational data stored?

- Why don't we always keep all tables in memory like `nycflights13`?
- How do RDBMS's store the data in their tables?
- Are RDBMS's only good for accessing large data on disk?
- Why does data in a RDBMS sometimes occupy more storage space than the actual data?
- Why (as seen in the homework) are keys explicitly specified when creating tables in SQL?

How does a RDBMS actually manage relational data?

- RBDMS's use **indexing** to look up data faster
- Indexes store where (in storage) to find particular rows in a table
- Indexes require additional storage space but allow faster queries
- Indexes are often a different type of data structure
  - B-trees (sorted data)
  - Hash tables (unsorted data)
- Primary and foreign keys are often indexed
- When working with data in memory, this often isn't important, but indexing can be very important when accessing data stored on disk

Visualizing joins

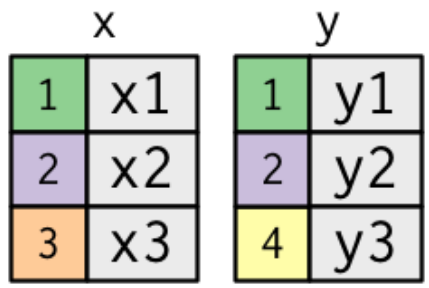


Figure 2: Two tables

Visualizing joins

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

Mutating joins

Similar to `mutate()`, **mutating joins** *add new variables* from one table (`x`) to matching observations in another (`y`)

- **Inner joins** keep only observations the appear in both `x` and `y`
- **Outer joins** keep observations that appear in at least one of `x` or `y`
  - A **left join** keeps all observations in `x`
  - A **right join** keeps all observations in `y`
  - A **full join** keeps all observations from both `x` and `y`

Mutating joins

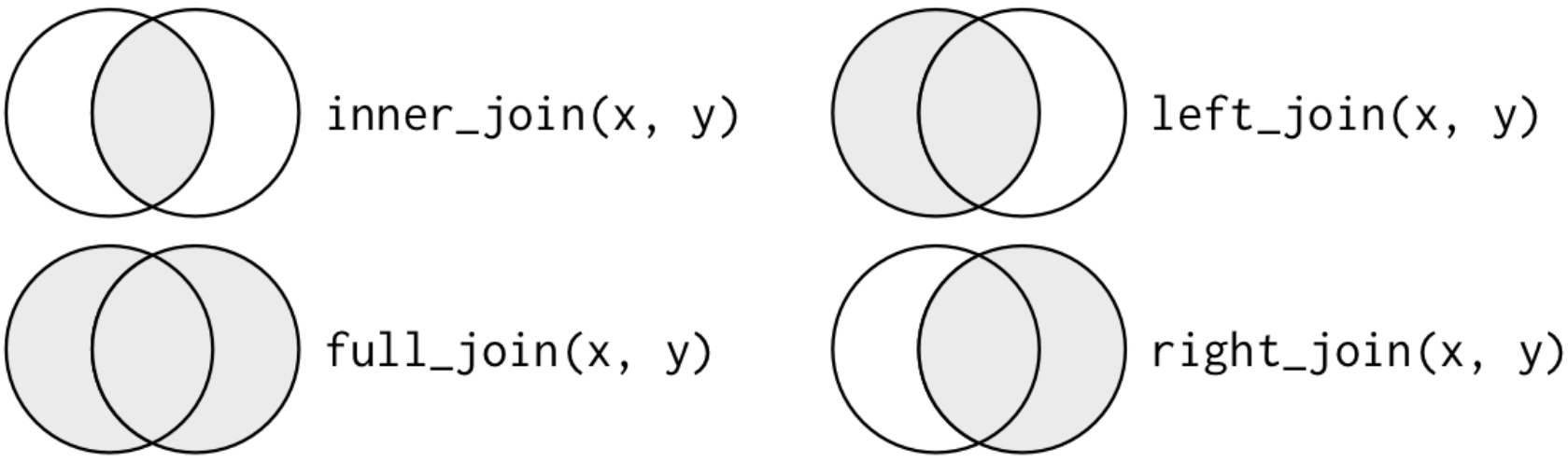


Figure 3: Types of joins as Venn diagrams

Visualizing joins

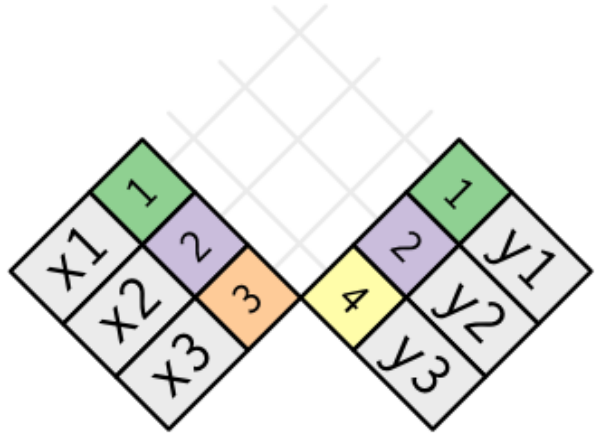


Figure 4: Two tables with possible matches

Inner joins

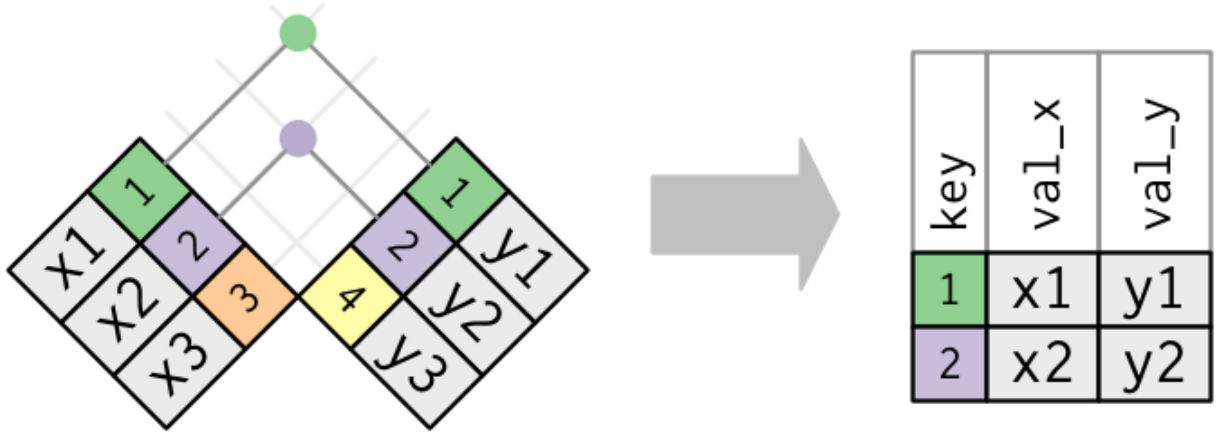


Figure 5: Keep observations that appear in both

Inner joins

```
inner_join(x, y)
```

```
## Joining, by = "key"
```

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

Inner joins

We can specify the variables used to join `x` and `y` explicitly.

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

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

Joining by all variables that appear in both (default) is a **natural join**.

Inner joins

`dplyr`:

```
inner_join(x, y, by = "z")
```

`base` R:

```
merge(x, y, by = "z")
```

SQL:

```
SELECT * FROM x INNER JOIN y USING (z)
```

Inner joins can silently drop observations, so are rarely used for data analysis.

Outer joins

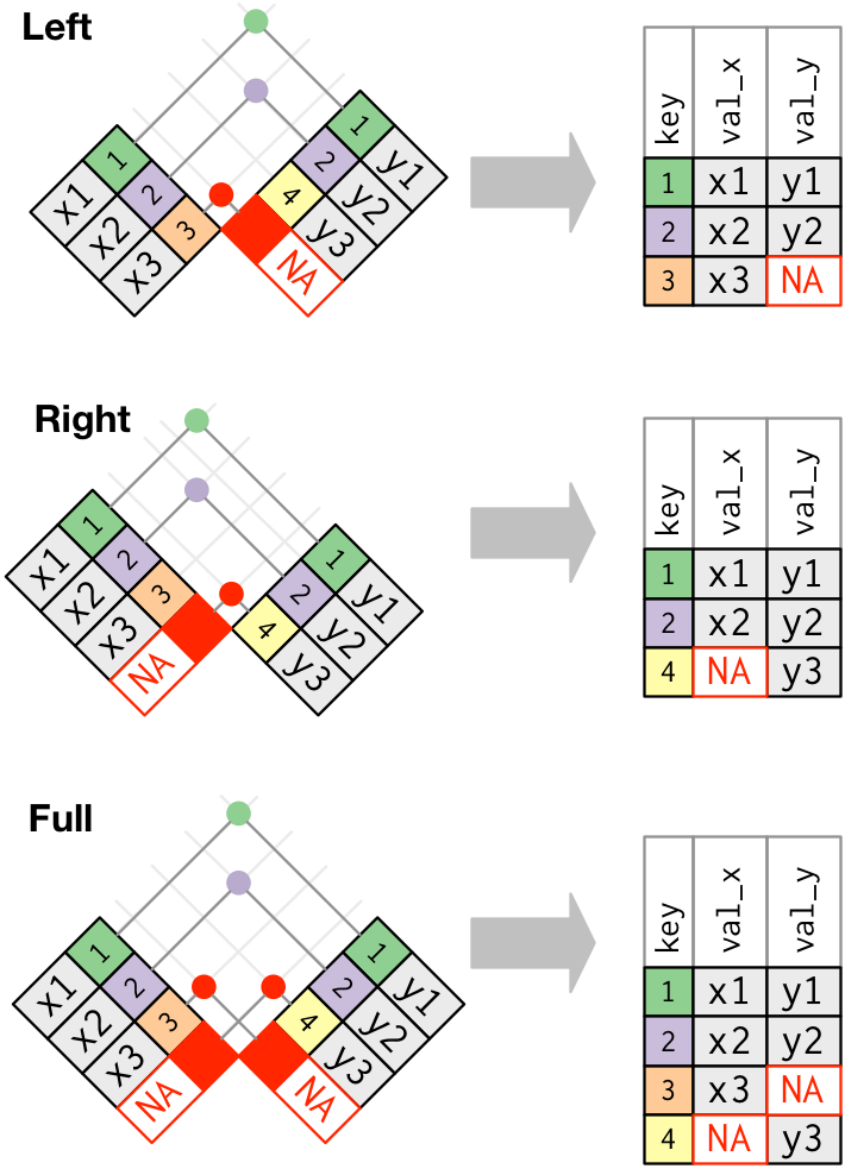


Figure 6: Keep observations that appear in at least one

Left join

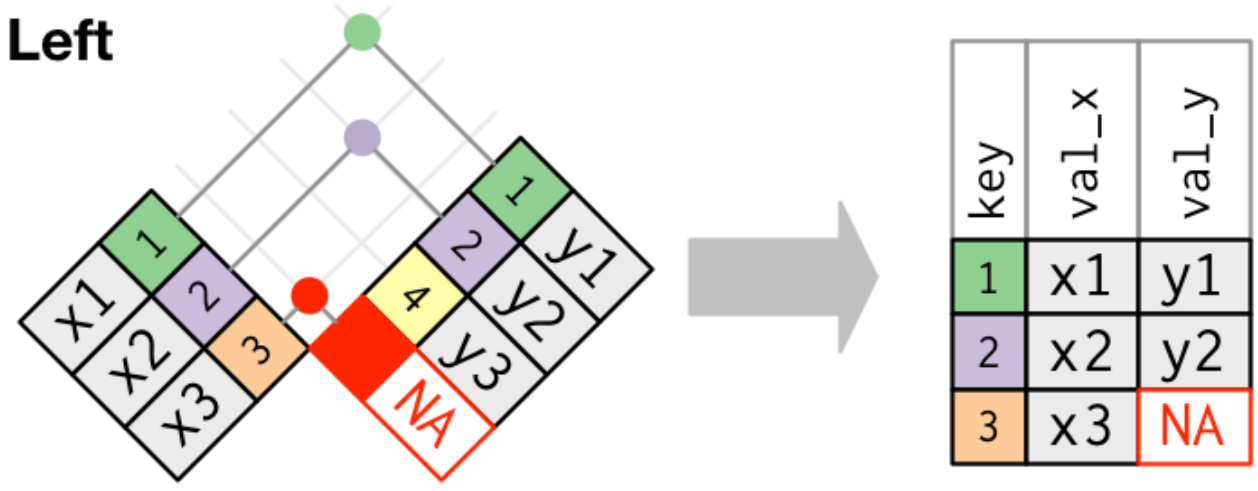


Figure 7: Keep observations that appear in `x`

Left join

```
left_join(x, y)
```

```
## Joining, by = "key"
```

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

Left join

dplyr:

```
left_join(x, y, by = "z")
```

base R:

```
merge(x, y, all.x = TRUE, by = "z")
```

SQL:

```
SELECT * FROM x LEFT OUTER JOIN y USING (z)
```

Left joins are the most common type of join.

Right join

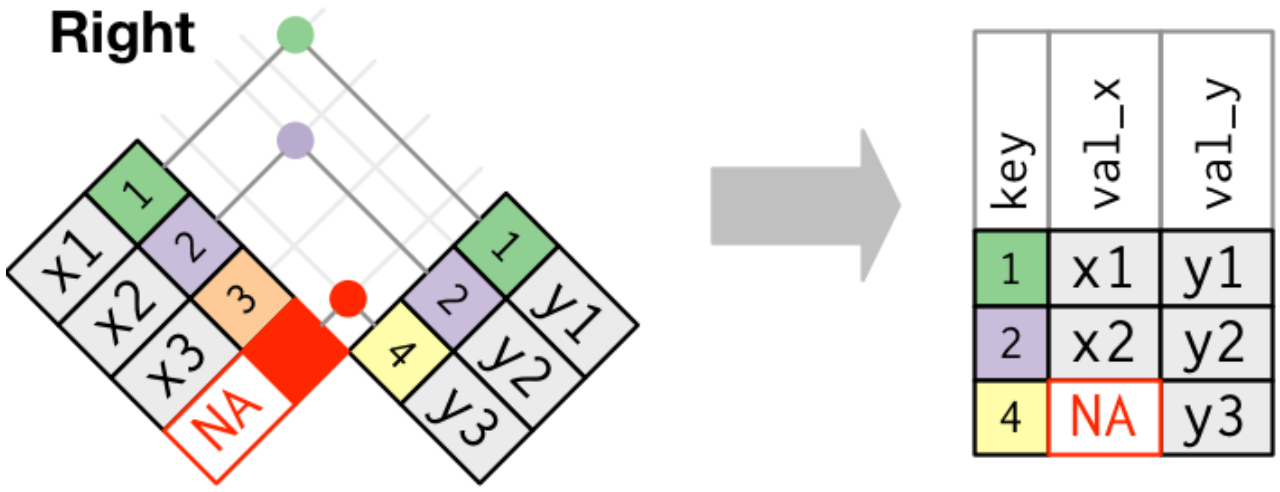


Figure 8: Keep observations that appear in `y`

Right join

```
right_join(x, y)
```

```
## Joining, by = "key"
```

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

Right join

dplyr:

```
right_join(x, y, by = "z")
```

base R:

```
merge(x, y, all.y = TRUE, by = "z")
```

SQL:

```
SELECT * FROM x RIGHT OUTER JOIN y USING (z)
```

Left versus right joins

What is the difference between the following calls?

```
left_join(x, y)
```

```
right_join(y, x)
```

```
left_join(x, y)
```

```
## Joining, by = "key"
```

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

```
right_join(y, x)
```

```
## Joining, by = "key"
```

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

`left_join(x, y)` vs `right_join(y, x)`

- The only difference is the order of the returned columns
- Explicit right joins exist primarily for expressiveness
- Some RDBMS's do not even implement right joins (e.g., SQLite)

Full join

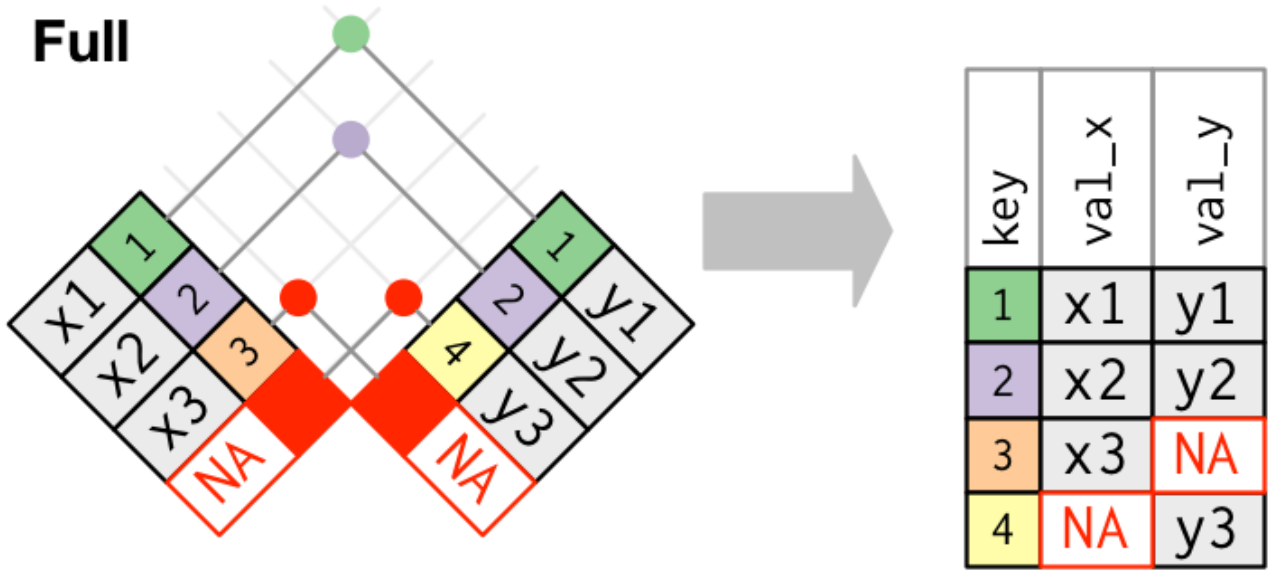


Figure 9: Keep observations that appear in either

Full join

```
full_join(x, y)
```

```
## Joining, by = "key"
```

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

Full join

dplyr:

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

base R:

```
merge(x, y, all.x = TRUE, all.y = TRUE, by = "z")
```

SQL:

```
SELECT * FROM x FULL OUTER JOIN y USING (z)
```

Duplicate keys

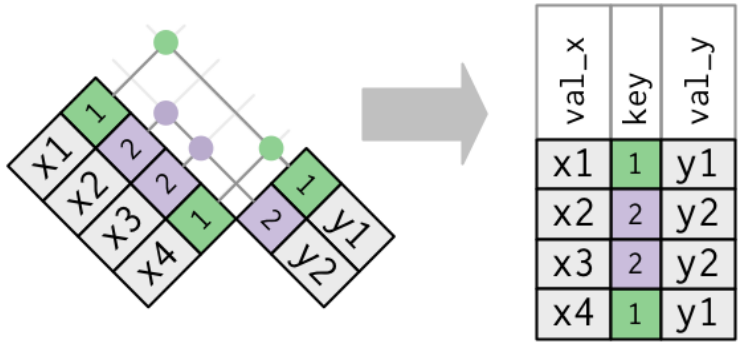


Figure 10: “One-to-many”

Duplicate keys

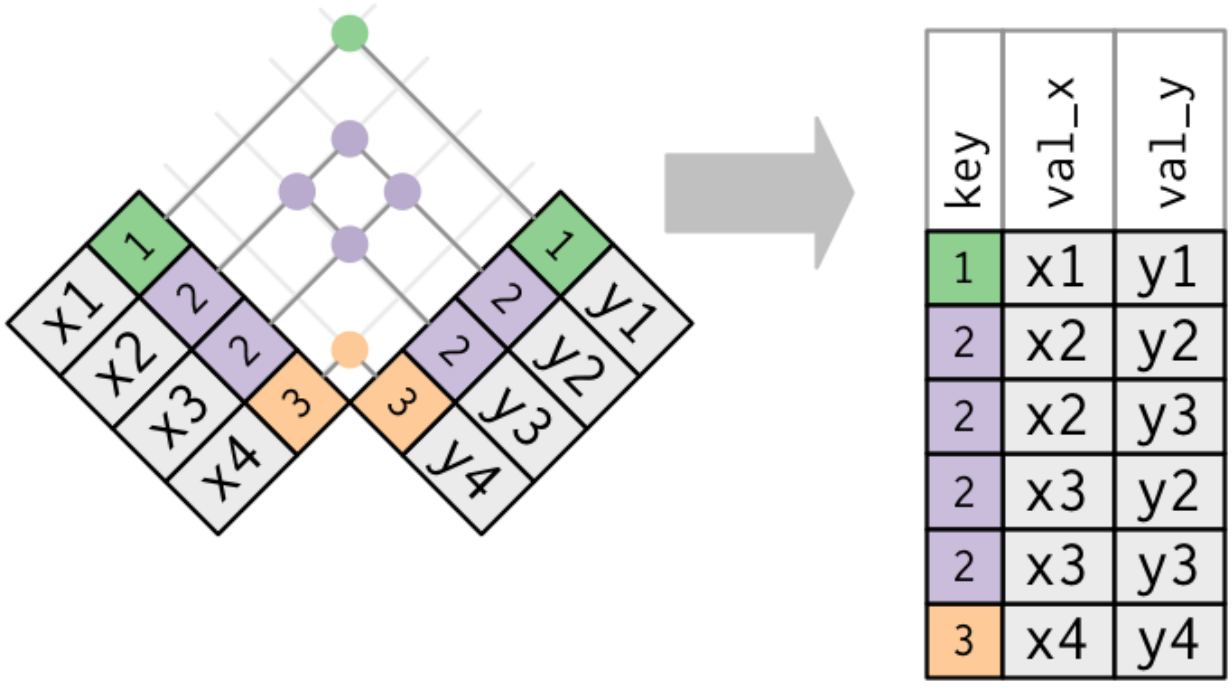


Figure 11: “Many-to-many”

Differently-named keys

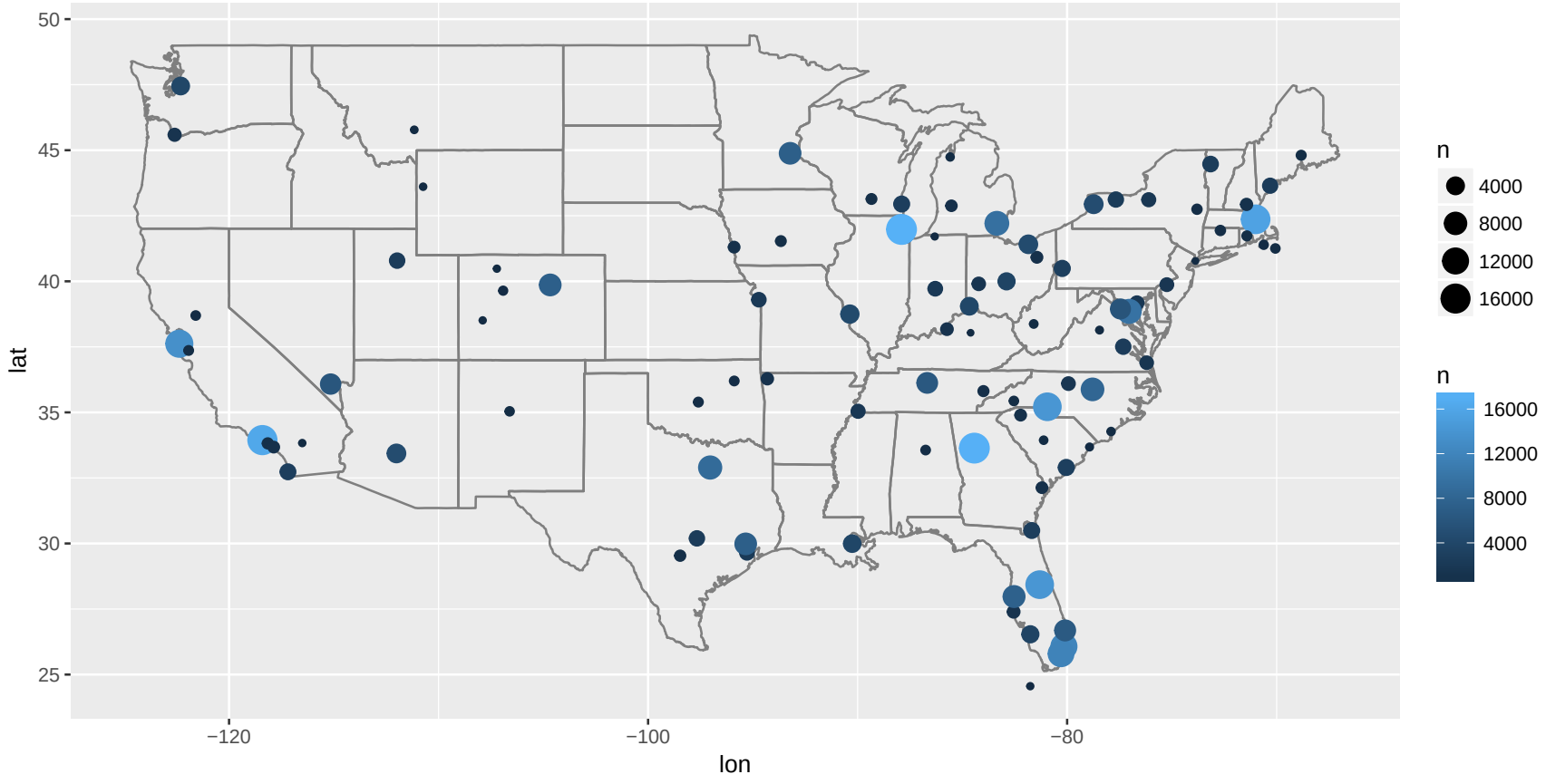
```
x2 <- x %>% rename(a = key)
y2 <- y %>% rename(b = key)
left_join(x2, y2, by = c("a" = "b"))
```

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

Plot most popular flight destinations

```
flights %>%
  filter(dest != "HNL", dest != "ANC") %>%
  count(dest) %>%
  left_join(airports, by = c("dest" = "faa")) %>%
  ggplot(aes(lon, lat, size=n, color=n)) +
  borders("state") +
  geom_point() +
  coord_quickmap()
```

```
## Warning: Removed 4 rows containing missing values (geom_point
```



Get total number of flights flown by each type of plane

```
flights %>%
  left_join(planes, by = "tailnum") %>%
  count(type)
```

```
## # A tibble: 4 x 2
##                          type      n
##                         <chr>  <int>
## 1  Fixed wing multi engine 282074
## 2 Fixed wing single engine   1686
## 3                Rotorcraft    410
## 4                      <NA>  52606
```

Mutating joins

Similar to `filter()`, **filtering joins** do not add new variables, but instead *subset the observations* from one table (`x`) based on observations in another (`y`)

- **Semi joins** *keep* all observations in `x` that have a match in `y`
- **Anti joins** *drop* all observations in `x` that have a match in `y`

Semi joins

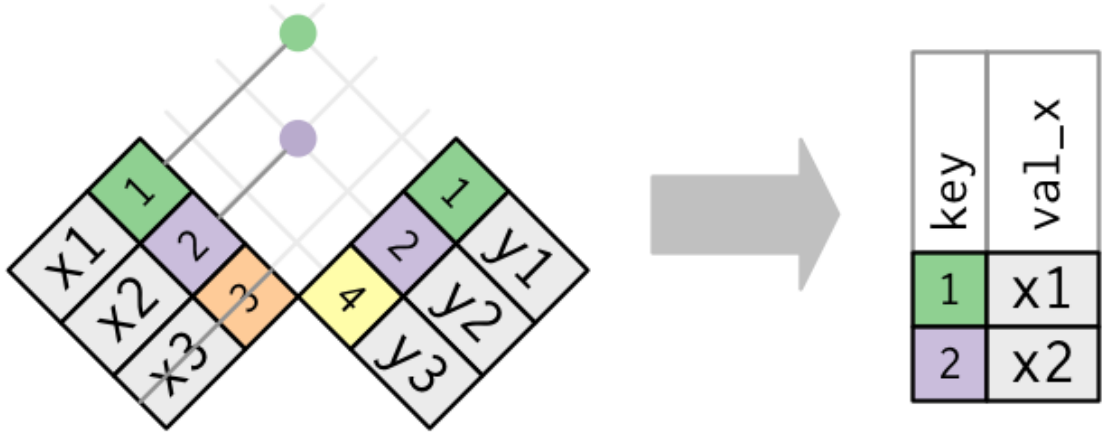


Figure 12: Keeps all observations that match

Semi joins

```
semi_join(x, y)
```

```
## Joining, by = "key"
```

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

Semi joins with duplicate keys

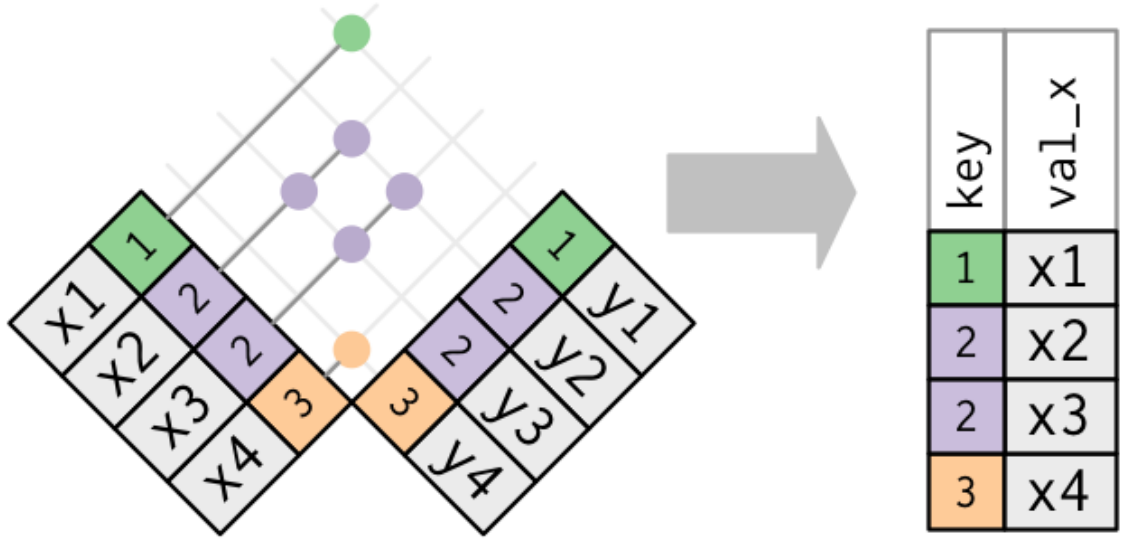


Figure 13: Keeps all observations that match

Semi joins never duplicate rows (unlike mutating joins).

Keep only flights from most popular destinations

```
top_dest <- flights %>%
  count(dest) %>%
  filter(n > 10000)

semi_join(flights, top_dest)
```

```
## Joining, by = "dest"

## # A tibble: 131,440 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      542            540         2      923
##  2  2013     1     1      554            600        -6      812
##  3  2013     1     1      554            558        -4      740
##  4  2013     1     1      555            600        -5      913
##  5  2013     1     1      557            600        -3      838
##  6  2013     1     1      558            600        -2      753
##  7  2013     1     1      558            600        -2      924
##  8  2013     1     1      558            600        -2      923
##  9  2013     1     1      559            559         0      702
## 10  2013     1     1      600            600         0      851
## # ... with 131,430 more rows, and 12 more variables: sched_arr_time
```

Anti joins

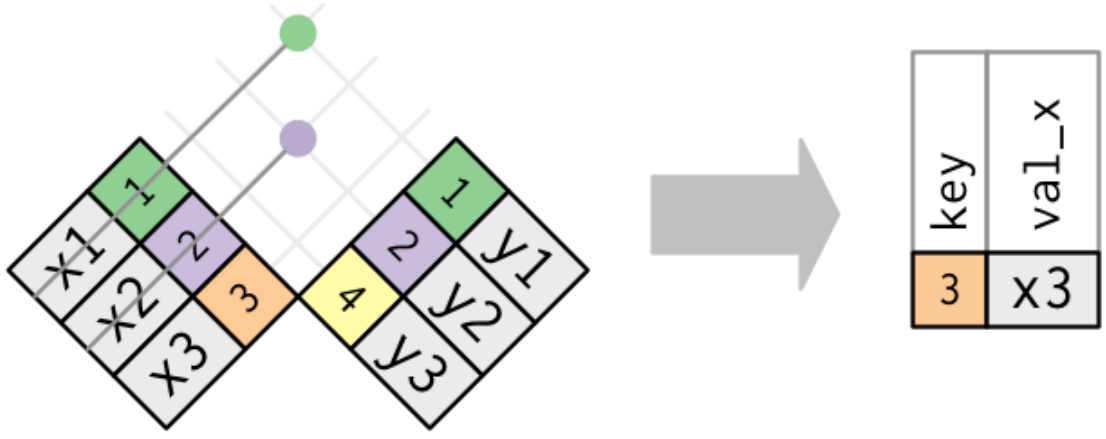


Figure 14:

Anti joins

```
anti_join(x, y)
```

```
## Joining, by = "key"
```

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

Why did we have missing values when plotting destination airports?

```
flights %>%
  anti_join(airports, by = c("dest" = "faa")) %>%
  count(dest)
```

```
## # A tibble: 4 x 2
##    dest     n
##   <chr> <int>
## 1   BQN   896
## 2   PSE   365
## 3   SJU  5819
## 4   STT   522
```

Set operations

Set operations take two tables with the same variables and compare whole rows (i.e., every variable in each row), to treat observations like members of sets.

- `intersect(x, y)` returns observations in both `x` and `y`
- `union(x, y)` returns all unique observations in `x` and `y`
- `setdiff(x, y)` returns the observations in `x` that aren't in `y`

These are less commonly used for data analysis than the joins.

Set operations

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

Intersection

```
intersect(df1, df2)
```

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

Union

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

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

Difference

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

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