DSA2101

Essential Data Analytics Tools: Data Visualization

Yuting Huang

Week 7 Relational data

Midterm exam

Time: Monday March 11, 8:15-9:15am at MPSH 1A.

Things to bring on the exam day:

- A laptop with the latest `R`, `RStudio`, and Examplify installed.
- The laptop charger.
- Your NUS matriculation card.

Arrive at least 15 minutes early at the venue for necessary setups (download of data sets, etc).

Midterm exam

1. The exam will be available for download on **Examplify** from Sunday Marth 10th at 8pm.
   - Only one download is allowed.
   - Make sure you download the exam to the correct laptop that will be used during the exam.
2. Exam data files will be available on **Canvas** on **Monday 8am**.
3. The following `R` packages are required for the exam:
   - `readxl`, `lubridate`, `stringr`, `tidyverse`
4. Submit your `Rmd` file to Canvas immediately after the exam. The submission box closes at 9:30am.

Submission requirements

1. Answer all questions in a single `R` Markdown file (`.Rmd`). Make sure it can knit to HTML without error.
2. At the end of the exam:
   - **Copy and paste your entire Rmd code to the Examplify text box.**. Indentation and alignment may not be retained when pasting, and that is acceptable.
   - Save an **EXACT** copy of your `Rmd` file on your laptop for Canvas submission.
   - Do not modify your code in your submission file. Any difference found (except for indentation or alignment) between Examplify and Canvas submission will be penalized.
3. The exam ends at 9:15am. Ensure that you submit your `Rmd` to Canvas by 9:30am.

Now till the exam day

Week 7

Update `R`, `RStduio`, and Examplify
Install all required packages:
`readxl, stringr, lubridate, tidyverse`

Week 8
Sunday 8pm

Download exam from Examplify

Monday 8am

Download exam data sets from Canvas

8:15am
Exam begins

No internet access once exam begins on Examplify.

Exam begins

Submit exam on Examplify

9:15am
Exam ends

Internet access resumes after exam ends on Examplify.

Submit the `Rmd` file to Canvas **immediately after** the exam.
Submission folder closes at 9:30am

After the exam: No tutorial meetings in Week 8.

No lecture on Wednesday.

Contents

- Data transformation — Week 5
  - `filter()`, `select()`, `mutate()`, `arrange()`, and `summarize()`
  - `group_by()` and `%>%`
- Tidy data — Week 6
  - `gather()`, `spread()`, `separate()` and `unite()`
- Relational data — Week 7
  - Mutating joins: `inner_join()`, `left_join()`, ...
  - Filtering joins: `semi_join()`, `anti_join()`
  - Set operations

Recap: Tidy data

Data never arrive in the condition that we need them. They need to be reshaped and reformatted.

"Tidy" Table

Business Unit	Year	Quarter	Budget
Sales	2000	Q1	2,500,000
Marketing	2000	Q1	1,000,000
Sales	2000	Q2	2,750,000
Marketing	2000	Q2	1,250,000
Sales	2000	Q3	3,000,000
Marketing	2000	Q3	4,000,000
Sales	2000	Q4	2,000,000
Marketing	2000	Q4	500,000
Sales	2001	Q1	2,500,000
Marketing	2001	Q1	1,500,000

"UnTidy" Table

Past and projected budgets for WidgetCo.'s Sales and Marketing Org.						
Contact JDoe@widgets.ca for more information.						
	Year					
Business Unit	2000				2001	
	Q1	Q2	Q3	Q4	Q1	Q2*
Sales	2,500,000	2,750,000	3,000,000	2,000,000	2,500,000	3,000,000
Marketing	1,000,000	1,250,000	4,000,000	500,000	1,500,000	1,750,000
double check this number - JD						
*Projected Numbers						

- The tidy table is ready for use in `R`.
- The untidy table is not.

Recap: Tidy data

More challenging situations occurs when we have multiple pieces of information crammed into column names.

- We need to store these variables in separate new variables.

```
library(tidyverse)
data(who2)
colnames(who2)
```

```
##  [1] "country"   "year"      "sp_m_014"  "sp_m_1524" "sp_m_2534"
##  [6] "sp_m_3544" "sp_m_4554" "sp_m_5564" "sp_m_65"   "sp_f_014"
## [11] "sp_f_1524" "sp_f_2534" "sp_f_3544" "sp_f_4554" "sp_f_5564"
## [16] "sp_f_65"   "sn_m_014"  "sn_m_1524" "sn_m_2534" "sn_m_3544"
## [21] "sn_m_4554" "sn_m_5564" "sn_m_65"   "sn_f_014"  "sn_f_1524"
## [26] "sn_f_2534" "sn_f_3544" "sn_f_4554" "sn_f_5564" "sn_f_65"
## [31] "ep_m_014"  "ep_m_1524" "ep_m_2534" "ep_m_3544" "ep_m_4554"
## [36] "ep_m_5564" "ep_m_65"   "ep_f_014"  "ep_f_1524" "ep_f_2534"
## [41] "ep_f_3544" "ep_f_4554" "ep_f_5564" "ep_f_65"   "rel_m_014"
## [46] "rel_m_1524" "rel_m_2534" "rel_m_3544" "rel_m_4554" "rel_m_5564"
## [51] "rel_m_65"   "rel_f_014"  "rel_f_1524" "rel_f_2534" "rel_f_3544"
## [56] "rel_f_4554" "rel_f_5564" "rel_f_65"
```

Tidy up

Read the data documentation via `?who2` and examine the data set.

- How many pieces of information in this data?

```r
head(who2)
```

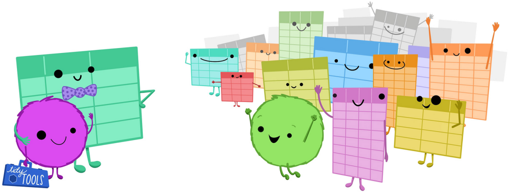
```
## # A tibble: 6 x 58
##   country      year sp_m_014 sp_m_1524 sp_m_2534 sp_m_3544 sp_m_4554 sp_m_55
##   <chr>       <dbl>    <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <db
## 1 Afghanistan  1980       NA        NA        NA        NA        NA
## 2 Afghanistan  1981       NA        NA        NA        NA        NA
## 3 Afghanistan  1982       NA        NA        NA        NA        NA
## 4 Afghanistan  1983       NA        NA        NA        NA        NA
## 5 Afghanistan  1984       NA        NA        NA        NA        NA
## 6 Afghanistan  1985       NA        NA        NA        NA        NA
## # i 50 more variables: sp_m_65 <dbl>, sp_f_014 <dbl>, sp_f_1524 <dbl>,
## #   sp_f_2534 <dbl>, sp_f_3544 <dbl>, sp_f_4554 <dbl>, sp_f_5564 <dbl>,
## #   sp_f_65 <dbl>, sn_m_014 <dbl>, sn_m_1524 <dbl>, sn_m_2534 <dbl>,
## #   sn_m_3544 <dbl>, sn_m_4554 <dbl>, sn_m_5564 <dbl>, sn_m_65 <dbl>,
## #   sn_f_014 <dbl>, sn_f_1524 <dbl>, sn_f_2534 <dbl>, sn_f_3544 <dbl>,
## #   sn_f_4554 <dbl>, sn_f_5564 <dbl>, sn_f_65 <dbl>, ep_m_014 <dbl>,
## #   ep_m_1524 <dbl>, ep_m_2534 <dbl>, ep_m_3544 <dbl>, ep_m_4554 <dbl>, ...
```

- To organize these information in separate columns, we pass a vector to the `names_to` argument, and instruct the function how to split the names into pieces via `names_sep`.

```
who2 %>%
  pivot_longer(cols = !(country:year),
               names_to = c("diagnosis", "gender", "age"),
               names_sep = "_",
               values_to = "count")
```

```
## # A tibble: 405,440 x 6
##    country      year diagnosis gender age   count
##    <chr>       <dbl> <chr>     <chr>  <chr> <dbl>
##  1 Afghanistan  1980 sp        m      014      NA
##  2 Afghanistan  1980 sp        m      1524     NA
##  3 Afghanistan  1980 sp        m      2534     NA
##  4 Afghanistan  1980 sp        m      3544     NA
##  5 Afghanistan  1980 sp        m      4554     NA
##  6 Afghanistan  1980 sp        m      5564     NA
##  7 Afghanistan  1980 sp        m      65       NA
##  8 Afghanistan  1980 sp        f      014      NA
##  9 Afghanistan  1980 sp        f      1524     NA
## 10 Afghanistan  1980 sp        f      2534     NA
## # i 405,430 more rows
```

When one table is not enough



When working with real-world data, you will often find that data are stored across **multiple** files or data frames.

- Typically, these tables have to be combined to answer the questions we are interested in.
- Many tables of data are called **relational data**.

Artwork by Allison Horst

When one table is not enough

restaurant

name	id	address	type
Taco Stand	AH13JK	1 Main St.	Mexican
Pho Place	**JJ29JJ**	192 Street Rd.	Vietnamese
Taco Stand	XJ11AS	18 W. East St.	Fusion
Pizza Heaven	CI21AA	711 K Ave.	Italian

health inspections

name	id	inspection_date	inspector	score
Taco Stand	AH13JK	2018-08-21	Sheila	97
Pho Place	**JJ29JJ**	2018-03-12	D'eonte	98
Pho Place	**JJ29JJ**	2018-01-02	Monica	66
Taco Stand	XJ11AS	2018-12-16	Mark	43
Pizza Heaven	CI21AA	2018-08-21	Anh	99

rating

name	id	stars
Taco Stand	AH13JK	4.9
Pho Place	**JJ29JJ**	4.8
Taco Stand	XJ11AS	4.2
Pizza Heaven	CI21AA	4.7

Consider a town with a number of restaurants. Across multiple data files, we have information on

- Location and type of cuisine.
- Health and safety inspections results.
- Online ratings on the restaurant.

Advantages of relational data

Storing data across multiple files has a number of benefits:

- **Efficient data storage:** Limit the need to repeat information.
- **Easier data updates:** If we need to update information, we can make the change in a single file.
- **Privacy:** We can restrict access to some of the data to ensure only those who should have access are able to read the data.

New York flights in 2013

Today, we work with **five** related tables from the `nycflights13` package:

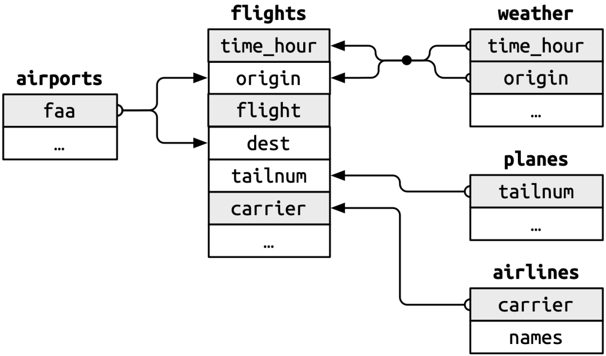
1. `flights`: All flights that departed New York City in 2013.
2. `airlines`: Carrier name and its abbreviated code.
3. `airports`: Information about airports.
4. `planes`: Plane's `tailnum` found in the FAA aircraft registry.
5. `weather`: Weather at each airport in New York at each hour.

Let's load the necessary packages.

```r
library(nycflights13)
```

New York flights data

Here is a diagram (database schema) that identifies the connections between tables:



Keys

The variable that connects each pair of data sets are called **keys**.

- A variable (or a *minimal* set of variables) that uniquely identifies an observation in a data frame.

In the schema on the previous slide,

- In the `planes` table, `tailnum` is the key variable.
- In the `weather` table, each observation is uniquely identified by a set of variables: `year`, `month`, `day`, `hour`, and `origin`.

Primary key

Each data join involves a pair of keys: Primary key and foreign key.

- **Primary key** uniquely identifies an observation in its own table.
- `carrier` is the primary key for the `airlines` table:

```
head(airlines)
```

```
## # A tibble: 6 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.
```

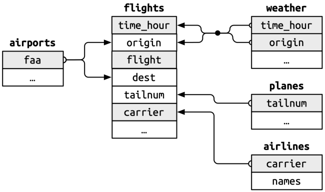
Primary key

- When more than one variable is needed, the key is called a **compound key**.
- `origin` and `time_hour` are the compound key for the `weather` table:

```r
head(weather)
```

```
## # A tibble: 6 x 15
##   origin  year month   day  hour  temp  dewp humid wind_dir wind_speed wind_
##   <chr>  <int> <int> <int> <int> <dbl> <dbl> <dbl>    <dbl>      <dbl>     <
## 1 EWR     2013     1     1     1  39.0  26.1  59.4      270      10.4
## 2 EWR     2013     1     1     2  39.0  27.0  61.6      250       8.06
## 3 EWR     2013     1     1     3  39.0  28.0  64.4      240      11.5
## 4 EWR     2013     1     1     4  39.9  28.0  62.2      250      12.7
## 5 EWR     2013     1     1     5  39.0  28.0  64.4      260      12.7
## 6 EWR     2013     1     1     6  37.9  28.0  67.2      240      11.5
## # i 4 more variables: precip <dbl>, pressure <dbl>, visib <dbl>,
## #   time_hour <dttm>
```

Foreign key

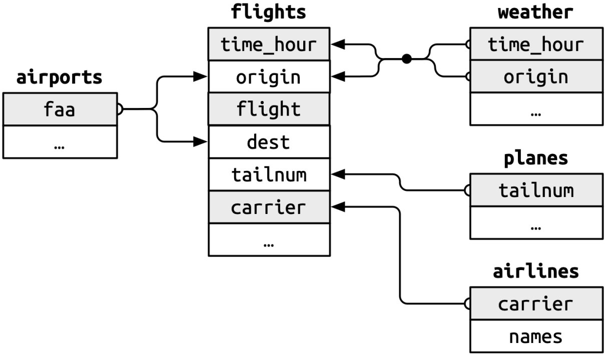


Foreign key is the counterpart of primary key. It uniquely identifies an observation in a different table.

- `flights$carrier` is a foreign key that corresponds to the primary key `airlines$carrier`.
- `flights$origin` and `flights$time_hour` is a compound foreign key that corresponds to the compound primary key `weather$origin` and `weather$time_hour`.
- A variable can be a primary and a foreign key at the same time.

Primary and foreign keys

These relationship can be summarized visually in the following.



Checking primary keys

Once you identify the primary keys for your tables, it is good practice to double-check if they are indeed unique.

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

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

- That is, `tailnum` uniquely identifies observations in the `planes` table.

```
weather %>%
  count(origin, time_hour) %>% filter(n > 1)
```

```
## # A tibble: 0 x 3
## # i 3 variables: origin <chr>, time_hour <dttm>, n <int>
```

- That is, `origin` and `time_hour` uniquely identifies observations in the `weather` table.

Checking primary keys

We should also check for missing values in the primary keys.

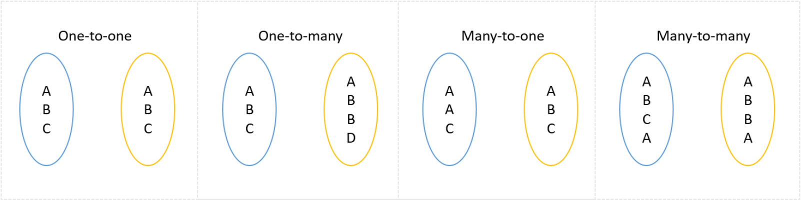
- If a value is missing, then it cannot identify an observation.

```
planes %>% filter(is.na(tailnum))
```

```
## # A tibble: 0 x 9
## # i 9 variables: tailnum <chr>, year <int>, type <chr>, manufacturer <chr>,
## #   model <chr>, engines <int>, seats <int>, speed <int>, engine <chr>
```

```
weather %>% filter(is.na(time_hour) | is.na(origin))
```

```
## # A tibble: 0 x 15
## # i 15 variables: origin <chr>, year <int>, month <int>, day <int>, hour <in
## #   temp <dbl>, dewp <dbl>, humid <dbl>, wind_dir <dbl>, wind_speed <dbl>,
## #   wind_gust <dbl>, precip <dbl>, pressure <dbl>, visib <dbl>,
## #   time_hour <dttm>
```



A primary key and the corresponding foreign key forms a **relation**.

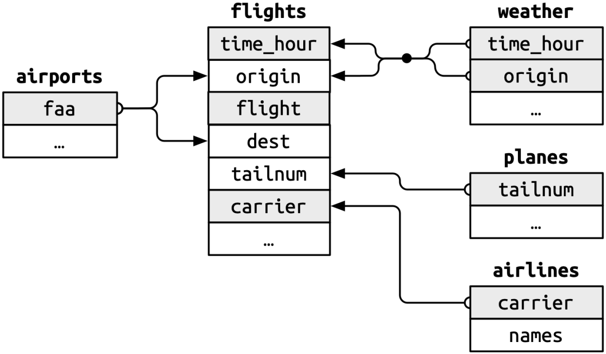
- Ideally, relationships are **one-to-one**.
- In real-life data sets, relations are typically **one-to-many** or **many-to-one**:
  - E.g., each flight has one plane, but each plane flies many flights.
- Relations can also be **many-to-many**:
  - Each airline flies to many airports, each airport hosts many airlines.

Relation between the tables

To work with relational data, we need functions that works with pairs of tables.

- **Mutating joins**: Add new variables to one data frame from matching observations in another data frame.
- **Filtering joins**: Filter observations from one data frame based on whether they can be matched to an observation in another data frame.
- **Set operations**: Treat observations as if they were set elements.

Relation between the tables



Let's combine a pair of tables using **mutating join**.

- `flights` and `airlines` via `carrier`.

Mutating join

- To ease demonstration, let's first create a narrower data frame:

```
flights2 <- flights %>%
  select(time_hour, origin, dest, tailnum, carrier)
flights2
```

```
## # A tibble: 336,776 x 5
##    time_hour           origin dest  tailnum carrier
##    <dttm>              <chr>  <chr> <chr>   <chr>
##  1 2013-01-01 05:00:00 EWR    IAH   N14228  UA
##  2 2013-01-01 05:00:00 LGA    IAH   N24211  UA
##  3 2013-01-01 05:00:00 JFK    MIA   N619AA  AA
##  4 2013-01-01 05:00:00 JFK    BQN   N804JB  B6
##  5 2013-01-01 06:00:00 LGA    ATL   N668DN  DL
##  6 2013-01-01 05:00:00 EWR    ORD   N39463  UA
##  7 2013-01-01 06:00:00 EWR    FLL   N516JB  B6
##  8 2013-01-01 06:00:00 LGA    IAD   N829AS  EV
##  9 2013-01-01 06:00:00 JFK    MCO   N593JB  B6
## 10 2013-01-01 06:00:00 LGA    ORD   N3ALAA  AA
## # i 336,766 more rows
```

Keys

```r
head(airlines, 3)
```

```
## # A tibble: 3 x 2
##   carrier name
##   <chr>   <chr>
## 1 9E      Endeavor Air Inc.
## 2 AA      American Airlines Inc.
## 3 AS      Alaska Airlines Inc.
```

- `carrier` is a primary key in `airlines`.
- It is a foreign key in `flights2` as it uniquely identifies observations in a foreign table.

```r
# check uniqueness
airlines %>% count(carrier) %>% filter(n > 1)
```

```
## # A tibble: 0 x 2
## # i 2 variables: carrier <chr>, n <int>
```

Mutating join

Join the two tables via key, `carrier`.

```
flights2 %>% left_join(airlines, by = "carrier")
```

```
## # A tibble: 336,776 x 6
##    time_hour           origin dest  tailnum carrier name
##    <dttm>              <chr>  <chr> <chr>   <chr>   <chr>
##  1 2013-01-01 05:00:00 EWR    IAH   N14228  UA      United Air Lines Inc.
##  2 2013-01-01 05:00:00 LGA    IAH   N24211  UA      United Air Lines Inc.
##  3 2013-01-01 05:00:00 JFK    MIA   N619AA  AA      American Airlines Inc.
##  4 2013-01-01 05:00:00 JFK    BQN   N804JB  B6      JetBlue Airways
##  5 2013-01-01 06:00:00 LGA    ATL   N668DN  DL      Delta Air Lines Inc.
##  6 2013-01-01 05:00:00 EWR    ORD   N39463  UA      United Air Lines Inc.
##  7 2013-01-01 06:00:00 EWR    FLL   N516JB  B6      JetBlue Airways
##  8 2013-01-01 06:00:00 LGA    IAD   N829AS  EV      ExpressJet Airlines Inc.
##  9 2013-01-01 06:00:00 JFK    MCO   N593JB  B6      JetBlue Airways
## 10 2013-01-01 06:00:00 LGA    ORD   N3ALAA  AA      American Airlines Inc.
## # i 336,766 more rows
```

- The names of the airlines are added to the end of the `flights2` table.

- We can also find out what size of plane was flying:

```
flights2 %>%
  left_join(planes %>% select(tailnum, engines, seats), by = "tailnum")
```

```
## # A tibble: 336,776 x 7
##    time_hour           origin dest  tailnum carrier engines seats
##    <dttm>              <chr>  <chr> <chr>   <chr>     <int> <int>
##  1 2013-01-01 05:00:00 EWR    IAH   N14228  UA            2   149
##  2 2013-01-01 05:00:00 LGA    IAH   N24211  UA            2   149
##  3 2013-01-01 05:00:00 JFK    MIA   N619AA  AA            2   178
##  4 2013-01-01 05:00:00 JFK    BQN   N804JB  B6            2   200
##  5 2013-01-01 06:00:00 LGA    ATL   N668DN  DL            2   178
##  6 2013-01-01 05:00:00 EWR    ORD   N39463  UA            2   191
##  7 2013-01-01 06:00:00 EWR    FLL   N516JB  B6            2   200
##  8 2013-01-01 06:00:00 LGA    IAD   N829AS  EV            2    55
##  9 2013-01-01 06:00:00 JFK    MCO   N593JB  B6            2   200
## 10 2013-01-01 06:00:00 LGA    ORD   N3ALAA  AA           NA    NA
## # i 336,766 more rows
```

In a `left_join()`, columns from the right-hand table are added to the end of the left-hand table.

- It is like "creating" a new variable at the end of the original data frame.
- When it fails to find a match, it fills in the new variables with missing values.
- For example, there is no information about the plane with tail number `N3ALAA`:

```
flights2 %>%
  left_join(planes %>% select(tailnum, engines, seats), by = "tailnum") %>%
  filter(tailnum == "N3ALAA") %>% head()
```

```
## # A tibble: 6 x 7
##   time_hour           origin dest  tailnum carrier engines seats
##   <dttm>              <chr>  <chr> <chr>   <chr>     <int> <int>
## 1 2013-01-01 06:00:00 LGA    ORD   N3ALAA  AA           NA    NA
## 2 2013-01-02 18:00:00 LGA    ORD   N3ALAA  AA           NA    NA
## 3 2013-01-03 06:00:00 LGA    ORD   N3ALAA  AA           NA    NA
## 4 2013-01-07 19:00:00 LGA    ORD   N3ALAA  AA           NA    NA
## 5 2013-01-08 17:00:00 JFK    ORD   N3ALAA  AA           NA    NA
## 6 2013-01-16 06:00:00 LGA    ORD   N3ALAA  AA           NA    NA
```

Understanding joins

In the following, we will learn four **mutating join** functions.

- Inner join: `inner_join()`.
- Outer joins: `left_join()`, `right_join()`, `full_join()`.

To understand how joins work, let's create simpler data sets and use visual representations:

- In the following, we use the `tibble()` function to create the simple data frames.

```
x = tibble(key = c(1, 2, 3),
           val_x = c("x1", "x2", "x3"))

y = tibble(key = c(1, 2, 4),
           val_y = c("y1", "y2", "y3"))
```

Understanding joins

The tables we just created look like:

x

key	var_x
1	x1
2	x2
3	x3

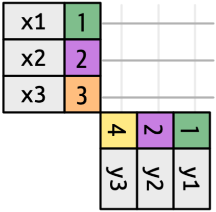
y

key	var_y
1	y1
2	y2
4	y3

- The colored column represents the **key** variable.
- The grey column represents the value.
- For simplicity, we show a single key variable, but the idea generalizes to multiple keys and multiple values.

Defining a join

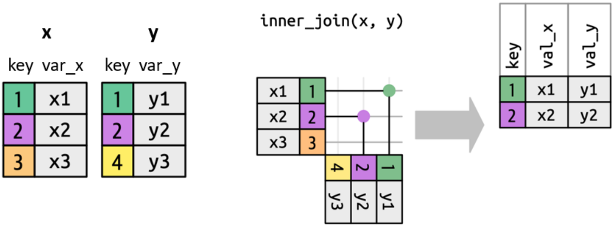
A join is a way of connecting each row in table `x` to zero, one, or more rows in table `y`.



- If you look closely, you may notice that we switched the order of the key and value columns in table `x`.
- This is to emphasize that joins matches based on the **key** variable.

Defining a join

In an actual join, matches will be indicated with dots.



- Number of dots = number of matches.
- Different types of joins will result in different number of rows.

Inner join

The simplest type of join is `inner_join()`.

- An inner join matches pairs of observations whenever their keys are equal.
- It keeps observations that appear in **both** tables, and removes all unmatched ones.

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

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

Outer joins

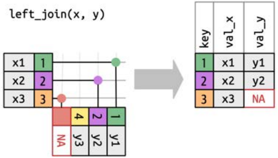
An **outer join** keeps observations that appear in **at least one** of the tables.

1. `left_join()`: Keeps all rows in `x`, including those not matched in `y`.
2. `right_join()`: Keeps all rows in `y`, including those not matched in `x`.
3. `full_join()`: Keeps all rows in both tables, regardless of matches.

These joins work by adding "virtual" observations to each table. The matched observations have their original values, the unmatched ones are filled with `NA`.

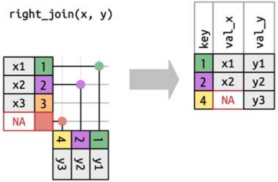
```r
x %>% left_join(y, by = "key")
```

```
## # 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>
```



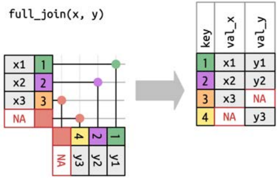

```r
x %>% right_join(y, by = "key")
```

```
## # 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
```

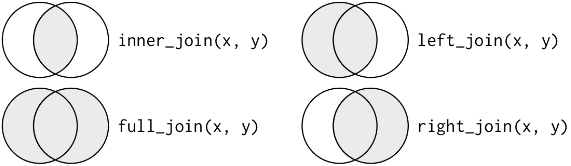



```r
x %>% full_join(y, by = "key")
```

```
## # 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
```



Use `left_join()` as your default join.



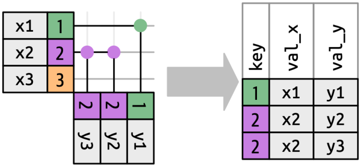
- The most common join is `left_join()`, as it preserves the original observation even when there isn't a match.
- **`left_join()` should be your default join**, unless you have a strong reason to prefer one of the others.

Row matching

So far, we've explored what happens if a row in `x` matches zero or one row in `y`.

This is not always the case.

1. If one table has duplicated keys, then the matching row will be duplicated as well.



Duplicated keys

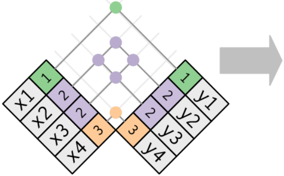
1. If one table has duplicated keys, then the matching row will be duplicated as well.

```
x = tibble(key = c(1, 2, 3),
           val_x = c("x1", "x2", "x3"))
y = tibble(key = c(1, 2, 2),
           val_y = c("y1", "y2", "y3"))
x %>% inner_join(y, by = "key")
```

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

Duplicated keys

2. If both table have duplicated keys, you get all possible combinations, the Cartesian product:
    - However, this is usually a data error.
    - In most cases, you need to have **unique keys** for at least one of your tables.



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

Duplicated keys

2. If both table have duplicated keys, you get all possible combinations, the Cartesian product:

```
x = tibble(key = c(1, 2, 2, 3),
           val_x = c("x1", "x2", "x3", "x4"))
y = tibble(key = c(1, 2, 2, 3),
           val_y = c("y1", "y2", "y3", "y4"))
x %>% left_join(y, by = "key")
```

```
## # A tibble: 6 x 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     2 x2    y3
## 4     2 x3    y2
## 5     2 x3    y3
## 6     3 x4    y4
```

Many-to-many joins are particularly problematic because they can result in a size explosion of the object returned from the join.

- This will have a large impact on the performance of your code.

Back to the New York flights data

- Let us return to the flights data, `flights2`

```
flights2
```

```
## # A tibble: 336,776 x 5
##    time_hour           origin dest  tailnum carrier
##    <dttm>              <chr>  <chr> <chr>   <chr>
##  1 2013-01-01 05:00:00 EWR    IAH   N14228  UA
##  2 2013-01-01 05:00:00 LGA    IAH   N24211  UA
##  3 2013-01-01 05:00:00 JFK    MIA   N619AA  AA
##  4 2013-01-01 05:00:00 JFK    BQN   N804JB  B6
##  5 2013-01-01 06:00:00 LGA    ATL   N668DN  DL
##  6 2013-01-01 05:00:00 EWR    ORD   N39463  UA
##  7 2013-01-01 06:00:00 EWR    FLL   N516JB  B6
##  8 2013-01-01 06:00:00 LGA    IAD   N829AS  EV
##  9 2013-01-01 06:00:00 JFK    MCO   N593JB  B6
## 10 2013-01-01 06:00:00 LGA    ORD   N3ALAA  AA
## # i 336,766 more rows
```

Defining the key columns

There are several ways to specify the key variables.

1. Specify the argument `by = "key"`.

```
flights2 %>% left_join(airlines, by = "carrier")
```

```
## # A tibble: 336,776 x 6
##    time_hour           origin dest  tailnum carrier name
##    <dttm>              <chr>  <chr> <chr>   <chr>   <chr>
##  1 2013-01-01 05:00:00 EWR    IAH   N14228  UA      United Air Lines Inc.
##  2 2013-01-01 05:00:00 LGA    IAH   N24211  UA      United Air Lines Inc.
##  3 2013-01-01 05:00:00 JFK    MIA   N619AA  AA      American Airlines Inc.
##  4 2013-01-01 05:00:00 JFK    BQN   N804JB  B6      JetBlue Airways
##  5 2013-01-01 06:00:00 LGA    ATL   N668DN  DL      Delta Air Lines Inc.
##  6 2013-01-01 05:00:00 EWR    ORD   N39463  UA      United Air Lines Inc.
##  7 2013-01-01 06:00:00 EWR    FLL   N516JB  B6      JetBlue Airways
##  8 2013-01-01 06:00:00 LGA    IAD   N829AS  EV      ExpressJet Airlines Inc.
##  9 2013-01-01 06:00:00 JFK    MCO   N593JB  B6      JetBlue Airways
## 10 2013-01-01 06:00:00 LGA    ORD   N3ALAA  AA      American Airlines Inc.
## # i 336,766 more rows
```

2. Leave the `by` argument empty, then the function uses the common variables in the two tables.

```
flights2 %>% left_join(weather)
```

```
## # A tibble: 336,776 x 18
##    time_hour           origin dest  tailnum carrier  year month   day  hour
##    <dttm>              <chr>  <chr> <chr>   <chr>   <int> <int> <int> <int>
##  1 2013-01-01 05:00:00 EWR    IAH   N14228  UA       2013     1     1     5
##  2 2013-01-01 05:00:00 LGA    IAH   N24211  UA       2013     1     1     5
##  3 2013-01-01 05:00:00 JFK    MIA   N619AA  AA       2013     1     1     5
##  4 2013-01-01 05:00:00 JFK    BQN   N804JB  B6       2013     1     1     5
##  5 2013-01-01 06:00:00 LGA    ATL   N668DN  DL       2013     1     1     6
##  6 2013-01-01 05:00:00 EWR    ORD   N39463  UA       2013     1     1     5
##  7 2013-01-01 06:00:00 EWR    FLL   N516JB  B6       2013     1     1     6
##  8 2013-01-01 06:00:00 LGA    IAD   N829AS  EV       2013     1     1     6
##  9 2013-01-01 06:00:00 JFK    MCO   N593JB  B6       2013     1     1     6
## 10 2013-01-01 06:00:00 LGA    ORD   N3ALAA  AA       2013     1     1     6
## # i 336,766 more rows
## # i 9 more variables: temp <dbl>, dewp <dbl>, humid <dbl>, wind_dir <dbl>,
## #   wind_speed <dbl>, wind_gust <dbl>, precip <dbl>, pressure <dbl>,
## #   visib <dbl>
```

- The functions joins the tables using `time_hour`, and `origin`.

3. Use a character vector `by = c("a" = "b")`. This is useful when the names of the key variables are different in two tables.

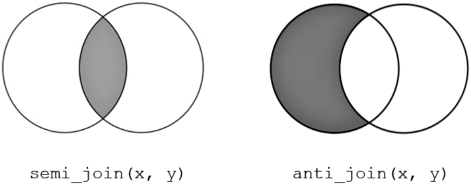
```
flights2 %>% left_join(airports, by = c("dest" = "faa"))
```

```
## # A tibble: 336,776 x 12
##    time_hour           origin dest  tailnum carrier name         lat   lon
##    <dttm>              <chr>  <chr> <chr>   <chr>   <chr>      <dbl> <dbl> <
##  1 2013-01-01 05:00:00 EWR    IAH   N14228  UA      George Bu~  30.0 -95.3
##  2 2013-01-01 05:00:00 LGA    IAH   N24211  UA      George Bu~  30.0 -95.3
##  3 2013-01-01 05:00:00 JFK    MIA   N619AA  AA      Miami Intl  25.8 -80.3
##  4 2013-01-01 05:00:00 JFK    BQN   N804JB  B6      <NA>        NA    NA
##  5 2013-01-01 06:00:00 LGA    ATL   N668DN  DL      Hartsfiel~  33.6 -84.4
##  6 2013-01-01 05:00:00 EWR    ORD   N39463  UA      Chicago O~  42.0 -87.9
##  7 2013-01-01 06:00:00 EWR    FLL   N516JB  B6      Fort Laud~  26.1 -80.2
##  8 2013-01-01 06:00:00 LGA    IAD   N829AS  EV      Washingto~  38.9 -77.5
##  9 2013-01-01 06:00:00 JFK    MCO   N593JB  B6      Orlando I~  28.4 -81.3
## 10 2013-01-01 06:00:00 LGA    ORD   N3ALAA  AA      Chicago O~  42.0 -87.9
## # i 336,766 more rows
## # i 3 more variables: tz <dbl>, dst <chr>, tzone <chr>
```

Filtering joins

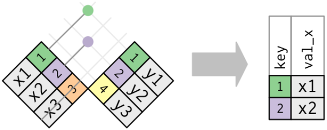
Filtering joins match observations in the same way as mutating joins, but affect the observations.

1. `semi_join(x, y)`: keeps all observations in `x` that have a match in `y`
   - It is similar to `inner_join()`, except that no columns are added.
2. `anti_join(x, y)`: drops all observations in `x` that have a match in `y`
   - It is useful for diagnosing join mismatches.

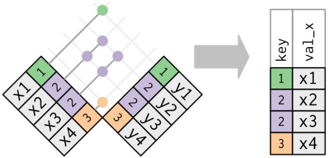


`semi_join()`

`semi_join()` keeps only the **matched** observations in `x`.



If there are duplicated keys in `x`, then all those rows are kept.



semi_join()

Find all flights that flew to the top 10 most popular destinations:

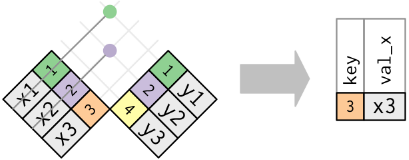
```r
top_dest <- flights %>%
  count(dest, sort = TRUE) %>% slice_max(n, n = 10)

flights2 %>% semi_join(top_dest)
```

```
## # A tibble: 141,145 x 5
##    time_hour           origin dest  tailnum carrier
##    <dttm>              <chr>  <chr> <chr>   <chr>
##  1 2013-01-01 05:00:00 JFK    MIA   N619AA  AA
##  2 2013-01-01 06:00:00 LGA    ATL   N668DN  DL
##  3 2013-01-01 05:00:00 EWR    ORD   N39463  UA
##  4 2013-01-01 06:00:00 EWR    FLL   N516JB  B6
##  5 2013-01-01 06:00:00 JFK    MCO   N593JB  B6
##  6 2013-01-01 06:00:00 LGA    ORD   N3ALAA  AA
##  7 2013-01-01 06:00:00 JFK    LAX   N29129  UA
##  8 2013-01-01 06:00:00 EWR    SFO   N53441  UA
##  9 2013-01-01 05:00:00 JFK    BOS   N708JB  B6
## 10 2013-01-01 06:00:00 LGA    FLL   N595JB  B6
## # i 141,135 more rows
```

anti_join()

`anti_join()` keeps only the **unmatched** observations in `x`.



- It is useful for diagnosing join mismatches.

anti_join()

- If we want to know whether there are `flights` that don't have a match in `planes`:

```
flights %>%
  anti_join(planes, by = "tailnum") %>%
  count(tailnum, sort = TRUE)
```

```
## # A tibble: 722 x 2
##    tailnum     n
##    <chr>   <int>
##  1 <NA>     2512
##  2 N725MQ    575
##  3 N722MQ    513
##  4 N723MQ    507
##  5 N713MQ    483
##  6 N735MQ    396
##  7 N0EGMQ    371
##  8 N534MQ    364
##  9 N542MQ    363
## 10 N531MQ    349
## # i 712 more rows
```

What does missing `tailnum` mean?

```
flights %>%
  filter(is.na(tailnum)) %>%
  select(tailnum, ends_with("time"))
```

```
## # A tibble: 2,512 x 6
##    tailnum dep_time sched_dep_time arr_time sched_arr_time air_time
##    <chr>      <int>          <int>    <int>          <int>    <dbl>
##  1 <NA>          NA           1545       NA           1910       NA
##  2 <NA>          NA           1601       NA           1735       NA
##  3 <NA>          NA            857       NA           1209       NA
##  4 <NA>          NA            645       NA            952       NA
##  5 <NA>          NA            845       NA           1015       NA
##  6 <NA>          NA           1830       NA           2044       NA
##  7 <NA>          NA            840       NA           1001       NA
##  8 <NA>          NA            820       NA            958       NA
##  9 <NA>          NA           1645       NA           1838       NA
## 10 <NA>          NA            755       NA           1012       NA
## # i 2,502 more rows
```

- These are the flights that were cancelled.

Potential joining problems

The data you have seen today have been cleaned up so you have as few problems as possible.

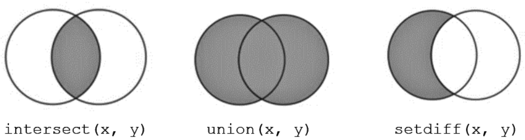
Your own data is unlikely to be so nice. So there are a few things you should do with your own data to make your joins go more smoothly.

1. Identify the primary keys in each variable.
   - Use `count()` in conjuncture with `filter()`.
2. Check that none of the variables in the primary key are missing. If a value is missing, it cannot identify an observation.
   - Use `filter()` with `is.na()`.
3. Check that foreign keys match primary keys in another table.
   - The best way to do this is an `anti_join()`.

Set operations

The final type of two-table functions are the set operators. They are not used as frequently, but they are occasionally useful.

1. `intersect(x, y)`: returns only observations in both `x` and `y`
2. `union(x, y)`: returns unique observations in `x` and `y`
3. `setdiff(x, y)`: returns observations in `x`, but not `y`.



Set operations

df1

x	y
1	1
2	1

df2

x	y
1	1
1	2

- Consider the following two tibbles:

```
df1 = tibble(x = c(1, 2),
             y = c(1, 1))

df2 = tibble(x = c(1, 1),
             y = c(1, 2))
```

- `df1` and `df2` have the same number of columns. Column names are also the same.
- Set operations work with a **complete row**, comparing the values of every variable.
- They expect the `x` and `y` inputs to have the same variables, and **treat the observations like sets**.

intersect() and union()

1. `intersect()` returns only the observations that present in both tables

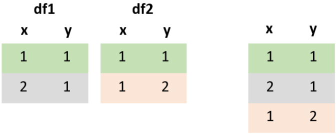
```r
intersect(df1, df2)
```

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

2. `union()` returns unique observations. Note that we get 3 rows, instead of 4.

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

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

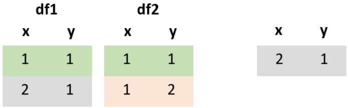


Two possibilities of `setdiff()`

3. `setdiff()` returns observations in the first input that does not appear in the second input.

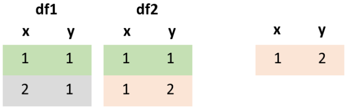
```r
setdiff(df1, df2)
```

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



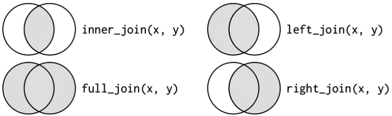

```r
setdiff(df2, df1)
```

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

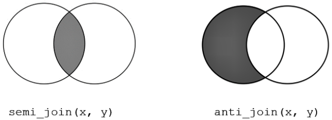


Summary of relational data

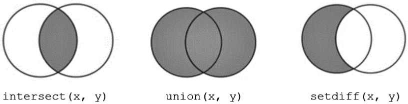
- **Mutating joins**:
  Match by key variables and keep columns of both inputs.



- **Filtering joins**:
  Match by key variables and keep columns of the first input.



- **Set operations**:
  Expect column names to be the same in two inputs and compare values of every row.



Case study: `nycflights13`

Given the information in the `nycflights13` data sets, we may be interested in the following questions:

1. Which airline has the highest number of delayed departures? Find the name of the airline.
2. On average, to which airport do flights arrive most early? Find the name of the airport.
3. In which month in 2013 do flights have the longest delays? Visualize your results in a graph.

1. Which airline has the highest number of delayed departures.

```
flights %>%
  filter(dep_delay > 0) %>%
  count(carrier, sort = TRUE) %>%
  left_join(airlines, by = "carrier") %>%
  top_n(10, n)
```

```
## # A tibble: 10 x 3
##    carrier     n name
##    <chr>   <int> <chr>
##  1 UA      27261 United Air Lines Inc.
##  2 EV      23139 ExpressJet Airlines Inc.
##  3 B6      21445 JetBlue Airways
##  4 DL      15241 Delta Air Lines Inc.
##  5 AA      10162 American Airlines Inc.
##  6 MQ       8031 Envoy Air
##  7 9E       7063 Endeavor Air Inc.
##  8 WN       6558 Southwest Airlines Co.
##  9 US       4775 US Airways Inc.
## 10 VX       2225 Virgin America
```

- We can also examine the *proportion* of flights that were delayed.

```
flights %>%
  group_by(carrier) %>%
  summarize(pct_delayed = mean(dep_delay > 0, na.rm = TRUE)*100) %>%
  arrange(desc(pct_delayed)) %>%
  left_join(airlines, by = "carrier") %>%
  top_n(10, pct_delayed)
```

```
## # A tibble: 10 x 3
##    carrier pct_delayed name
##    <chr>         <dbl> <chr>
##  1 WN             54.3 Southwest Airlines Co.
##  2 FL             51.9 AirTran Airways Corporation
##  3 F9             50   Frontier Airlines Inc.
##  4 UA             47.0 United Air Lines Inc.
##  5 EV             45.1 ExpressJet Airlines Inc.
##  6 VX             43.4 Virgin America
##  7 YV             42.8 Mesa Airlines Inc.
##  8 9E             40.6 Endeavor Air Inc.
##  9 B6             39.6 JetBlue Airways
## 10 MQ             31.9 Envoy Air
```

2. On average, to which airport do flights arrive most early?

```
flights %>%
  group_by(dest) %>%
  summarize(mean_arr_delay = mean(arr_delay, na.rm = TRUE)) %>%
  arrange(mean_arr_delay) %>%
  left_join(airports, by = c("dest" = "faa"))
```

```
## # A tibble: 105 x 9
##    dest  mean_arr_delay name                      lat    lon   alt    tz dst   t
##    <chr>          <dbl> <chr>                   <dbl>  <dbl> <dbl> <dbl> <chr> <
##  1 LEX          -22     "Blue Grass"             38.0  -84.6   979    -5 A     A
##  2 PSP          -12.7   "Palm Springs Intl"      33.8 -117.    477    -8 A     A
##  3 SNA           -7.87  "John Wayne Arpt O~      33.7 -118.     56    -8 A     A
##  4 STT           -3.84   <NA>                    NA      NA     NA    NA <NA>  <
##  5 ANC           -2.5   "Ted Stevens Ancho~      61.2 -150.    152    -9 A     A
##  6 HNL           -1.37  "Honolulu Intl"          21.3 -158.     13   -10 N     P
##  7 SEA           -1.10  "Seattle Tacoma In~      47.4 -122.    433    -8 A     A
##  8 MVY           -0.286 "Martha\\\\'s Vine~      41.4  -70.6    67    -5 A     A
##  9 LGB           -0.0620 "Long Beach"            33.8 -118.     60    -8 A     A
## 10 SLC            0.176 "Salt Lake City In~      40.8 -112.   4227    -7 A     A
## # i 95 more rows
```

3. In which month do flights tend to have the longest arrival delays?

```
flights %>%
  group_by(month) %>%
  summarize(mean_arr_delay = mean(arr_delay, na.rm = TRUE)) %>%
  arrange(desc(mean_arr_delay))
```

```
## # A tibble: 12 x 2
##    month mean_arr_delay
##    <int>          <dbl>
##  1     7         16.7
##  2     6         16.5
##  3    12         14.9
##  4     4         11.2
##  5     1          6.13
##  6     8          6.04
##  7     3          5.81
##  8     2          5.61
##  9     5          3.52
## 10    11          0.461
## 11    10         -0.167
## 12     9         -4.02
```

Visualizing the data

```
flights %>%
  group_by(month) %>%
  summarize(mean_arr_delay = mean(arr_delay, na.rm = TRUE)) -> df1

barplot(df1$mean_arr_delay, names.arg = df1$month,
        main = "Average delay by month", border = NA, col = "lightblue")
```