DSA2101

Essential Data Analytics Tools: Data Visualization

Yuting Huang

Week 7 Relational data

Midterm: Tuesday March 7th at LT32

This is an in-class, in-person exam.

Things to bring on the exam day:

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

Please **arrive at least 10 minutes early** on the exam day, for necessary setups (download of data sets etc).

Midterm: Tuesday March 7th at LT32

1. The exam will be available for download on **Examplify** from Monday March 6th 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 Tuesday 7:55am.
3. The following `R` packages are required for the exam:
   - `readxl`, `lubridate`, `stringr`, `tidyverse`
4. At the end of the exam. you need to submit your `Rmd` code to both Examplify and Canvas.

More on submission of the exam

Submit your `Rmd` code to **both Examplify and Canvas**:

- Copy and paste your `Rmd` code in the Examplify text box.
- Save an **exact copy** of the `Rmd` on your laptop for submission to Canvas **immediately after** the exam.
- Do not modify your code in the submission file. Any difference found between Examplify and Canvas submissions will be penalized.

Now till the exam day

Now	Update `R`, `RStduio`, and Examplify Make sure the required packages are installed
	Solve the sample questions
Week 8 Monday 8pm	Download exam from Examplify
Tuesday 7:55am	Download exam data files from Canvas

No internet access once exam begins on Examplify.

Exam begins

Copy and paste your `Rmd` code to Examplify
Save an **exact copy** of the `Rmd` on your laptop

Internet access resumes after exam ends on Examplify.

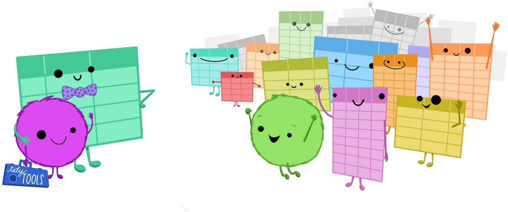
Submit the `Rmd` file to Canvas **immediately after** the exam

After the exam: **No lecture on Friday; no tutorial meetings in Week 8.**

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

When one tibble is not enough



It is rare that data analysis involves only a single table.

- 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

New York flights data

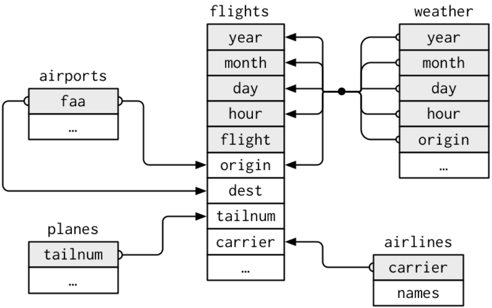
Today, we work with **five** tables from the `nycflights13` package:

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 for each hour.

New York flights data

```
library(tidyverse)
library(nycflights13)
```

Here is a diagram that identifies the keys that links the tables together:



Keys

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

- A variable (or a set of variables) that uniquely identifies an observation.
- 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 and foreign key

- A **primary key** uniquely identifies an observation in its own table.
- **Foreign key** is the counterpart of primary key. It uniquely identifies an observation in another table.
  - `flights$tailnum` is a foreign key, because it appears in `flights` and matches each flight to a unique plane `planes`.
- A variable can be a primary and a foreign key at the same time.
  - `weather$origin` is part of `weather`'s primary key, and also a foreign key for the `airports` table.

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

Uniqueness of keys

One way to check the uniqueness is to `count()` the primary keys, and look for entries with `n` greater than one.

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

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

Sometimes, a table does not have an explicit primary key:

- Each row is an observation, but no combination of variables reliably identifies it.

For example, what is the primary key in the `flights` table?

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

```
## # A tibble: 64,928 x 5
##     year month   day tailnum     n
##    <int> <int> <int> <chr>   <int>
##  1  2013     1     1 N0EGMQ      2
##  2  2013     1     1 N11189      2
##  3  2013     1     1 N11536      2
##  4  2013     1     1 N11544      3
##  5  2013     1     1 N11551      2
##  6  2013     1     1 N12540      2
##  7  2013     1     1 N12567      2
##  8  2013     1     1 N13123      2
##  9  2013     1     1 N13538      3
## 10  2013     1     1 N13566      3
## # ... with 64,918 more rows
```

Surrogate key

If a table lacks a primary key, it is sometimes useful to add one with `mutate()` and `row_number()`.

- Note that we sort the data prior to making the surrogate key. In this way, the order of the rows has some meaning.

```
flights %>%
  arrange(year, month, day, tailnum, sched_dep_time) %>%
  mutate(flight_id = row_number()) %>%
  select(year, month, day, tailnum, sched_dep_time, flight_id) %>%
  head(3)
```

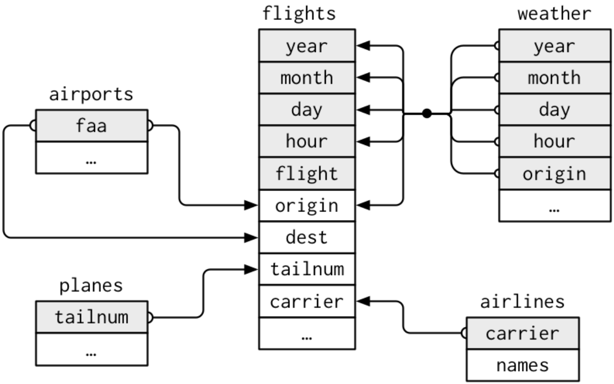
```
## # A tibble: 3 x 6
##    year month   day tailnum sched_dep_time flight_id
##   <int> <int> <int> <chr>            <int>     <int>
## 1  2013     1     1 N0EGMQ            1510         1
## 2  2013     1     1 N0EGMQ            2100         2
## 3  2013     1     1 N11107             630         3
```

Relations

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:
  - E.g., each flight has one plane, but each plane has many flights.
- Relations can also be many-to-many:
  - Each airline flies to many airports, each airport hosts many airlines.

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(year:day, hour, origin, dest, tailnum, carrier)
flights2 %>% head(6)
```

```
## # A tibble: 6 x 8
##    year month   day  hour origin dest  tailnum carrier
##   <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>
## 1  2013     1     1     5 EWR    IAH   N14228  UA
## 2  2013     1     1     5 LGA    IAH   N24211  UA
## 3  2013     1     1     5 JFK    MIA   N619AA  AA
## 4  2013     1     1     5 JFK    BQN   N804JB  B6
## 5  2013     1     1     6 LGA    ATL   N668DN  DL
## 6  2013     1     1     5 EWR    ORD   N39463  UA
```

Mutating join

We join `flights2` with `airlines` via the key `carrier`.

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

```
airlines %>% head(6)
```

```
## # 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.
```

Mutating join

Merge the two tables by adding the airline name from `airlines` to `flights2` via `carrier`

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

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

Mutating join

The result of joining `airlines` to `flights2` is an additional variable called `name`.

- It is like "creating" a new variable on the right-hand side of the original data frame using the `mutate()` function.

In the following, we will learn 4 mutating join functions.

- One inner join: `inner_join()`.
- Three types of outer joins: `left_join()`, `right_join()`, `full_join()`.

Understanding joins

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

- In the following, we use `tribble()` to create tibbles using an easier to read row-by-row layout.

```
x = tribble(
  ~key, ~val_x,
  1,    "x1",
  2,    "x2",
  3,    "x3"
)

y = tribble(
  ~key, ~val_y,
  1,   "y1",
  2,   "y2",
  4,   "y3"
)
```

Understanding joins

The tables we just created look like:

x

key	var_x
1	x1
2	x2
3	x3

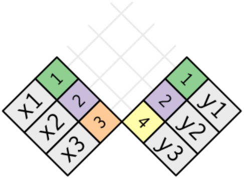
y

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 in a straightforward way to multiple keys and multiple values.

Defining a join

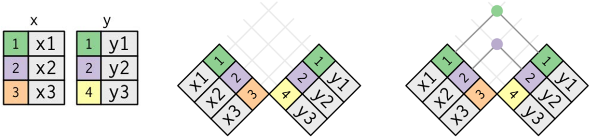
A join is a way of connecting each row in table `x` to zero, one, or more rows in table `y`.



- 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.



- The number of dots = the 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
```

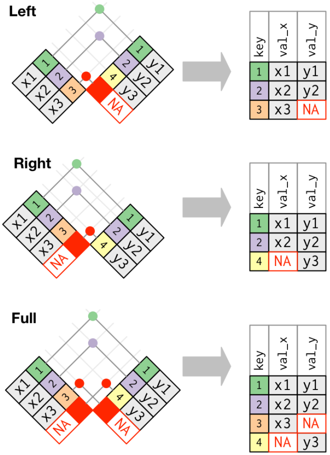
Outer joins

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 `x` and `y`, regardless of matches.

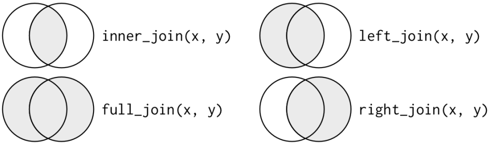
These joins work by adding "virtual" observations to each table. The matched observations have their original values, the unmatched ones are filled with `NA`.

Three types of outer joins



Three types of outer joins

Another way to depict the three types of joins:



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

```
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>
```

```
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
```

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

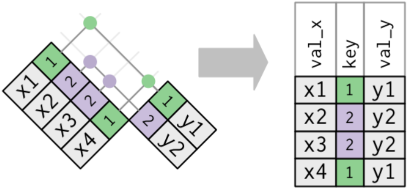
```
## # A tibble: 4 x 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
## 3     3 x3    <NA>
## 4     4 <NA>  y3
```

Duplicated keys

So far, all the diagrams have assumed that the keys are unique.

This is not always the case.

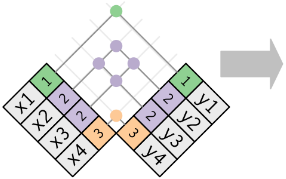
1. If one table has duplicated keys, then the matching row will be duplicated as well.



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

New York flights data

Let us return to the (narrow) New York flights data, `flights2`

```
flights2
```

```
## # A tibble: 336,776 x 8
##     year month   day  hour origin dest  tailnum carrier
##    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>
##  1  2013     1     1     5 EWR    IAH   N14228  UA
##  2  2013     1     1     5 LGA    IAH   N24211  UA
##  3  2013     1     1     5 JFK    MIA   N619AA  AA
##  4  2013     1     1     5 JFK    BQN   N804JB  B6
##  5  2013     1     1     6 LGA    ATL   N668DN  DL
##  6  2013     1     1     5 EWR    ORD   N39463  UA
##  7  2013     1     1     6 EWR    FLL   N516JB  B6
##  8  2013     1     1     6 LGA    IAD   N829AS  EV
##  9  2013     1     1     6 JFK    MCO   N593JB  B6
## 10  2013     1     1     6 LGA    ORD   N3ALAA  AA
## # ... with 336,766 more rows
```

Defining the key columns

There are several ways to specify the primary/foreign keys.

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

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

```
## # A tibble: 336,776 x 9
##     year month   day  hour origin dest  tailnum carrier name
##    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>   <chr>
##  1  2013     1     1     5 EWR    IAH   N14228  UA      United Air Lines Inc
##  2  2013     1     1     5 LGA    IAH   N24211  UA      United Air Lines Inc
##  3  2013     1     1     5 JFK    MIA   N619AA  AA      American Airlines In
##  4  2013     1     1     5 JFK    BQN   N804JB  B6      JetBlue Airways
##  5  2013     1     1     6 LGA    ATL   N668DN  DL      Delta Air Lines Inc.
##  6  2013     1     1     5 EWR    ORD   N39463  UA      United Air Lines Inc
##  7  2013     1     1     6 EWR    FLL   N516JB  B6      JetBlue Airways
##  8  2013     1     1     6 LGA    IAD   N829AS  EV      ExpressJet Airlines
##  9  2013     1     1     6 JFK    MCO   N593JB  B6      JetBlue Airways
## 10  2013     1     1     6 LGA    ORD   N3ALAA  AA      American Airlines In
## # ... with 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
##     year month   day  hour origin dest  tailnum carrier  temp  dewp humid
##    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>   <dbl> <dbl> <dbl>
##  1  2013     1     1     5 EWR    IAH   N14228  UA       39.0  28.0  64.4
##  2  2013     1     1     5 LGA    IAH   N24211  UA       39.9  25.0  54.8
##  3  2013     1     1     5 JFK    MIA   N619AA  AA       39.0  27.0  61.6
##  4  2013     1     1     5 JFK    BQN   N804JB  B6       39.0  27.0  61.6
##  5  2013     1     1     6 LGA    ATL   N668DN  DL       39.9  25.0  54.8
##  6  2013     1     1     5 EWR    ORD   N39463  UA       39.0  28.0  64.4
##  7  2013     1     1     6 EWR    FLL   N516JB  B6       37.9  28.0  67.2
##  8  2013     1     1     6 LGA    IAD   N829AS  EV       39.9  25.0  54.8
##  9  2013     1     1     6 JFK    MCO   N593JB  B6       37.9  27.0  64.3
## 10  2013     1     1     6 LGA    ORD   N3ALAA  AA       39.9  25.0  54.8
## # ... with 336,766 more rows, and 7 more variables: wind_dir <dbl>,
## #   wind_speed <dbl>, wind_gust <dbl>, precip <dbl>, pressure <dbl>,
## #   visib <dbl>, time_hour <dttm>
```

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

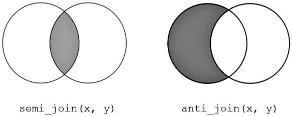
```
flights2 %>%
  left_join(airports, by = c("dest" = "faa"))
```

```
## # A tibble: 336,776 x 15
##     year month   day  hour origin dest  tailnum carrier name     lat   lon
##    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>   <chr>  <dbl> <dbl> <
##  1  2013     1     1     5 EWR    IAH   N14228  UA      Georg~  30.0 -95.3
##  2  2013     1     1     5 LGA    IAH   N24211  UA      Georg~  30.0 -95.3
##  3  2013     1     1     5 JFK    MIA   N619AA  AA      Miami~  25.8 -80.3
##  4  2013     1     1     5 JFK    BQN   N804JB  B6      <NA>    NA    NA
##  5  2013     1     1     6 LGA    ATL   N668DN  DL      Harts~  33.6 -84.4
##  6  2013     1     1     5 EWR    ORD   N39463  UA      Chica~  42.0 -87.9
##  7  2013     1     1     6 EWR    FLL   N516JB  B6      Fort ~  26.1 -80.2
##  8  2013     1     1     6 LGA    IAD   N829AS  EV      Washi~  38.9 -77.5
##  9  2013     1     1     6 JFK    MCO   N593JB  B6      Orlan~  28.4 -81.3
## 10  2013     1     1     6 LGA    ORD   N3ALAA  AA      Chica~  42.0 -87.9
## # ... with 336,766 more rows, and 3 more variables: tz <dbl>, dst <chr>,
## #   tzone <chr>
```

Filtering joins

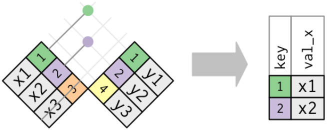
Filtering joins match observations in the same way as mutating joins, but affect the observations.

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`

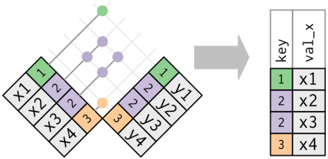


`semi_join()`

`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:

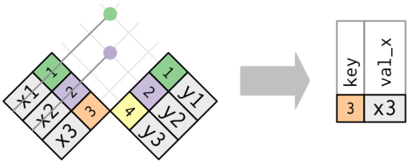
```
top_dest = flights %>%
  # Count and show the most popular dest at the top
  count(dest, sort = TRUE) %>% head(10)

flights2 %>%
  semi_join(top_dest)
```

```
## # A tibble: 141,145 x 8
##     year month   day  hour origin dest  tailnum carrier
##    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>
##  1  2013     1     1     5 JFK    MIA   N619AA  AA
##  2  2013     1     1     6 LGA    ATL   N668DN  DL
##  3  2013     1     1     5 EWR    ORD   N39463  UA
##  4  2013     1     1     6 EWR    FLL   N516JB  B6
##  5  2013     1     1     6 JFK    MCO   N593JB  B6
##  6  2013     1     1     6 LGA    ORD   N3ALAA  AA
##  7  2013     1     1     6 JFK    LAX   N29129  UA
##  8  2013     1     1     6 EWR    SFO   N53441  UA
##  9  2013     1     1     5 JFK    BOS   N708JB  B6
## 10  2013     1     1     6 LGA    FLL   N595JB  B6
## # ... with 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
## # ... with 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
## # ... with 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.

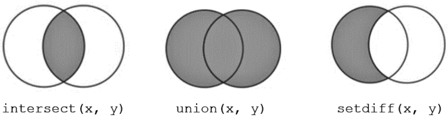
Your own data is unlikely to be so nice. So there are a few things you should do with your own data to make your joins go more smoothly.

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

Consider the following two tibbles:

```
df1 = tribble(
  ~x, ~y,
   1,  1,
   2,  1
)

df2 = tribble(
  ~x, ~y,
   1,  1,
   1,  2
)
```

```
df1; df2
```

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

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

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

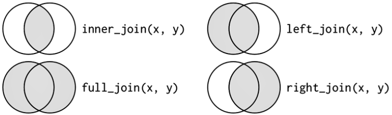
```
## # A tibble: 1 x 2
##       x     y
##   <dbl> <dbl>
## 1     2     1
```

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

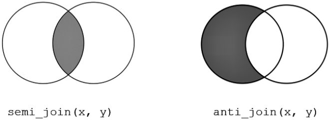
```
## # A tibble: 1 x 2
##       x     y
##   <dbl> <dbl>
## 1     1     2
```

Summary of relational data

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



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



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