Joining Data with dplyr

ETW2001 Foundations of Data Analysis and Modelling

Manjeevan Singh Seera

Outline

- Joining Data with dplyr
  - Keys
  - Mutating Joins
  - Duplicate Keys
  - Filtering Joins

MONASH University MALAYSIA

MONASH BUSINESS

Introduction

MONASH University MALAYSIA

It's rare that a data analysis involves only a single table of data. Generally you have many tables of data, and you must combine them to answer the questions that you're interested in.

Collectively, multiple tables of data are called relational data because it is the relations, not just the individual datasets, that are important.

- Relations are always defined between a pair of tables. All other relations are built up from this simple idea: the relations of three or more tables are always a property of the relations between each pair.
- Sometimes both elements of a pair can be the same table. As an example, you have a table of people, and each person has a reference to their parents.

Source: https://r4ds.had.co.nz

Relational data

To work with relational data, we need verbs that work with pairs of tables. There are 3 families of verbs designed to work with relational data:

- Mutating joins: add new variables to one data frame from matching observations in another.
- Filtering joins: filter observations from one data frame based on whether or not they match an observation in the other table.
- Set operations: treat observations as if they were set elements.

The most common place to find relational data is in a Relational Database Management System (**RDBMS**), with the most common tool, **SQL**. dplyr is simpler to use than SQL as dplyr is specialized to do data analysis: it makes common data analysis operations easier, at the expense of making it more difficult to do other things that aren't commonly needed for data analysis.

Prerequisites

Let's explore relational data from nycflights13 using the 2-table verbs from dplyr.

```
library(tidyverse)
```

```
install.packages("nycflights13")
library(nycflights13)
```

```
Installing package into '/usr/local/lib/R/site-library'
(as 'lib' is unspecified)
```

nycflights13

nycflights13 contains 4 tibbles that are related to the flights table that you used in data transformation.

airlines lets you look up the full carrier name from its abbreviated code.

```
airlines
```

A tibble: 16 × 2

carrier	name
<chr>	<chr>
9E	Endeavor Air Inc.
AA	American Airlines Inc.
AS	Alaska Airlines Inc.
B6	JetBlue Airways
DL	Delta Air Lines Inc.
EV	ExpressJet Airlines Inc.

nycflights13

airports gives information about each airport, identified by the **FAA airport code**.

```
airports
```

A tibble: 1458 × 8

faa	name	lat	lon	alt	tz	dst	tzone
<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<chr>	<chr>
04G	Lansdowne Airport	41.13047	-80.61958	1044	-5	A	America/New_York
06A	Moton Field Municipal Airport	32.46057	-85.68003	264	-6	A	America/Chicago
06C	Schaumburg Regional	41.98934	-88.10124	801	-6	A	America/Chicago
06N	Randall Airport	41.43191	-74.39156	523	-5	A	America/New_York
09J	Jekyll Island Airport	31.07447	-81.42778	11	-5	A	America/New_York
0A9	Elizabethton Municipal Airport	36.37122	-82.17342	1593	-5	A	America/New_York

nycflights13

planes gives information about each plane, identified by its **tailnum**.

```
planes
```

A tibble: 3322 × 9

tailnum	year	type	manufacturer	model	engines	seats	speed	engine
<chr>	<int>	<chr>	<chr>	<chr>	<int>	<int>	<int>	<chr>
N10156	2004	Fixed wing multi engine	EMBRAER	EMB-145XR	2	55	NA	Turbo-fan
N102UW	1998	Fixed wing multi engine	AIRBUS INDUSTRIE	A320-214	2	182	NA	Turbo-fan
N103US	1999	Fixed wing multi engine	AIRBUS INDUSTRIE	A320-214	2	182	NA	Turbo-fan
N104UW	1999	Fixed wing multi engine	AIRBUS INDUSTRIE	A320-214	2	182	NA	Turbo-fan
N10575	2002	Fixed wing multi engine	EMBRAER	EMB-145LR	2	55	NA	Turbo-fan
N105UW	1999	Fixed wing multi engine	AIRBUS INDUSTRIE	A320-214	2	182	NA	Turbo-fan

nycflights13

weather gives the weather at each NYC airport for each hour.

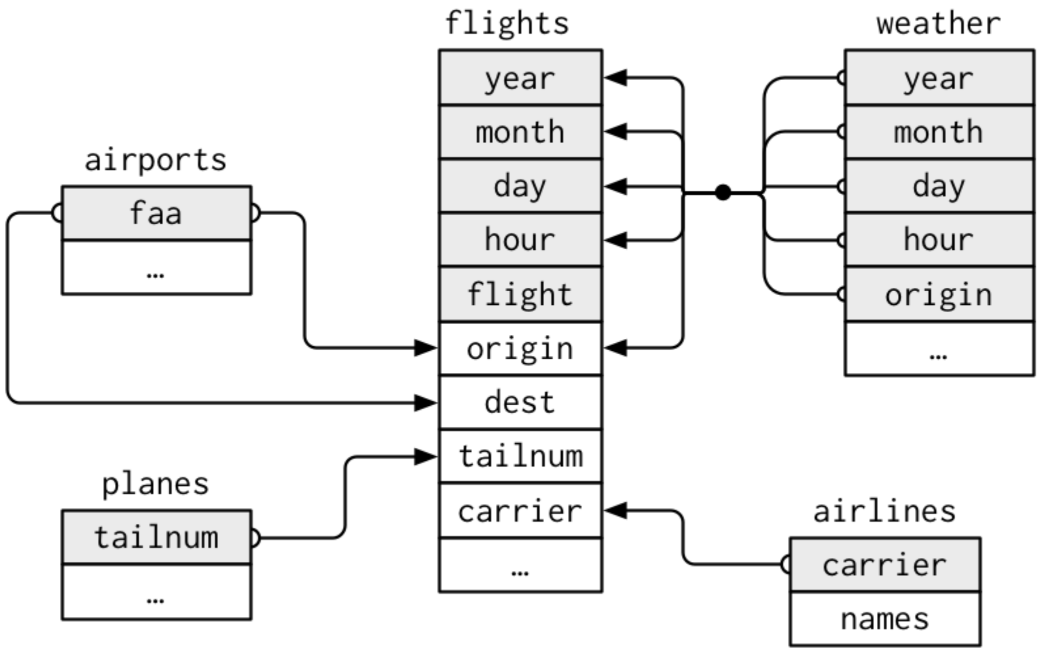
```
weather
```

A tibble: 26115 × 15

origin	year	month	day	hour	temp	dewp	humid	wind_dir	wind_speed	wind_gust	precip	pressure	visib	time_hour
<chr>	<int>	<int>	<int>	<int>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>
EWR	2013	1	1	1	39.02	26.06	59.37	270	10.35702	NA	0	1012.0	10	2013-01-01 01:00:00
EWR	2013	1	1	2	39.02	26.96	61.63	250	8.05546	NA	0	1012.3	10	2013-01-01 02:00:00
EWR	2013	1	1	3	39.02	28.04	64.43	240	11.50780	NA	0	1012.5	10	2013-01-01 03:00:00
EWR	2013	1	1	4	39.92	28.04	62.21	250	12.65858	NA	0	1012.2	10	2013-01-01 04:00:00
EWR	2013	1	1	5	39.02	28.04	64.43	260	12.65858	NA	0	1011.9	10	2013-01-01 05:00:00
EWR	2013	1	1	6	37.94	28.04	67.21	240	11.50780	NA	0	1012.4	10	2013-01-01 06:00:00
EWR	2013	1	1	7	39.02	28.04	64.43	240	14.96014	NA	0	1012.2	10	2013-01-01 07:00:00

Relationships

One way to show the relationships between the different tables is shown below.



MONASH BUSINESS

nycflights13

For nycflights13:

- flights connects to planes: via tailnum.
- flights connects to airlines: via carrier variable.
- flights connects to airports: via origin and dest variables.
- flights connects to weather: via origin (the location), and year, month, day and hour (the time).

Outline

- ✓ Joining Data with dplyr
    - ❑ Keys
    - ❑ Mutating Joins
    - ❑ Duplicate Keys
    - ❑ Filtering Joins

Keys

Variables used to connect each pair of tables are called keys. A key is a variable(s) that uniquely identifies an observation.

- In simple cases, a **single variable** is sufficient to identify an observation. As an example, each plane is uniquely identified by its tailnum.
- In other cases, **multiple variables** may be needed. As an example, to identify an observation in weather you need five variables: year, month, day, hour, and origin.

There are 2 types of keys (variable can be both primary and foreign key):

- Primary key: uniquely identifies an observation in its own table. As an example, planes$tailnum is a primary key because it uniquely identifies each plane in the planes table.
- Foreign key: uniquely identifies an observation in another table. As an example, flights$tailnum is a foreign key because it appears in the flights table where it matches each flight to a unique plane.

Count

Once we've identified the primary keys in your tables, it's good to verify that they do indeed uniquely identify each observation. One way to do that is to `count()` the primary keys and look for entries where n > 1.

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

A tibble: 0 × 2

tailnum	n
<chr>	<int>

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

A tibble: 3 × 6

year	month	day	hour	origin	n
<int>	<int>	<int>	<int>	<chr>	<int>
2013	11	3	1	EWR	2
2013	11	3	1	JFK	2
2013	11	3	1	LGA	2

Count

In some cases, a table doesn't have an explicit primary key: each row is an observation, but no combination of variables reliably identifies it.

As an example, what's the primary key in the flights table? Nothing seems unique.

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

A tibble: 29768 × 5

year	month	day	flight	n
<int>	<int>	<int>	<int>	<int>
2013	1	1	1	2
2013	1	1	3	2
2013	1	1	4	2

Keys

If a table <u>does not</u> have a **primary key**, it's sometimes useful to **add one** with `mutate()` and `row_number()`.

This makes it easier to match observations if you've done some filtering and want to check back in with the original data.

- This is called a **surrogate key**.

Keys

A primary key and the corresponding foreign key in another table form a relation. Relations are typically one-to-many.

- As an example, each flight has one plane, but each plane has many flights.

In other data, we sometimes see a 1-to-1 relationship. This can be a special case of 1-to-many. We can model many-to-many relations with a many-to-1 relation plus a 1-to-many relation.

- As an example, in this data there's a many-to-many relationship between airlines and airports: each airline flies to many airports; each airport hosts many airlines.

Outline

- ✓ Joining Data with dplyr
  - ✓ Keys
  - ❑ **Mutating Joins**
  - ❑ Duplicate Keys
  - ❑ Filtering Joins

Mutating joins

The first tool we'll look at for combining a pair of tables is the mutating join.

- A mutating join allows you to combine variables from two tables. It first matches observations by their keys, then copies across variables from one table to the other.

Like `mutate()`, join functions add variables to the right. In the case you have a lot of variables, this may not bee seen (as it will be on the most right). As an example, let's create a narrower dataset.

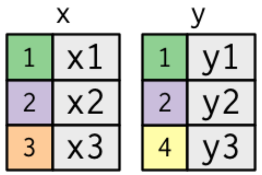
```
flights2 <- flights %>%
  select(year:day, hour, origin, dest, tailnum, carrier)
flights2
```

A tibble: 336776 × 8

year	month	day	hour	origin	dest	tailnum	carrier
<int>	<int>	<int>	<dbl>	<chr>	<chr>	<chr>	<chr>
2013	1	1	5	EWR	IAH	N14228	UA
2013	1	1	5	LGA	IAH	N24211	UA
2013	1	1	5	JFK	MIA	N619AA	AA

Understanding joins

A visual representation on how joins work.

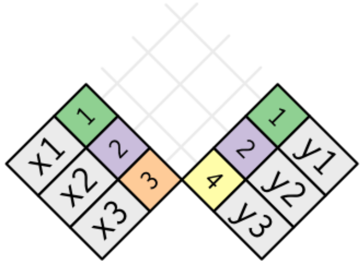



```
x <- tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     3, "x3"
)
y <- tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2",
     4, "y3"
)
```

Joins

Colored columns represents the key variable: these are used to match the rows between the tables.
Grey column represents the value column that is carried along for the ride.

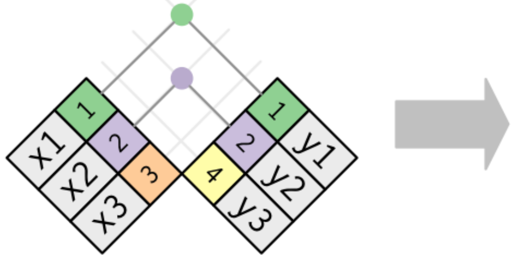
A join is a way of connecting each row in x to zero, one, or more rows in y. The diagram shows each potential match as an intersection of a pair of lines.



MONASH BUSINESS

Joins

In an actual join, matches will be indicated with dots. The number of dots = the number of matches = the number of rows in the output.



key	val_x	val_y
1	x1	y1
2	x2	y2

Inner join

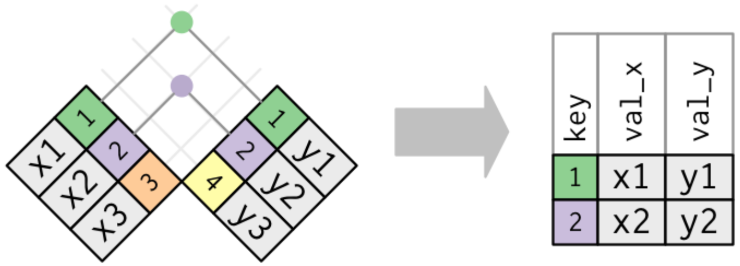
Simplest type of join is the inner join. An inner join matches pairs of observations whenever their keys are equal.

Output of an inner join is a new data frame that contains the key, the x values, and the y values. We use by to tell dplyr which variable is the key.

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

A tibble: 2 × 3

key	val_x	val_y
<dbl>	<chr>	<chr>
1	x1	y1
2	x2	y2



Unmatched rows are not included in the results, which means inner joins are not appropriate in analysis as it's easy to lose observations.

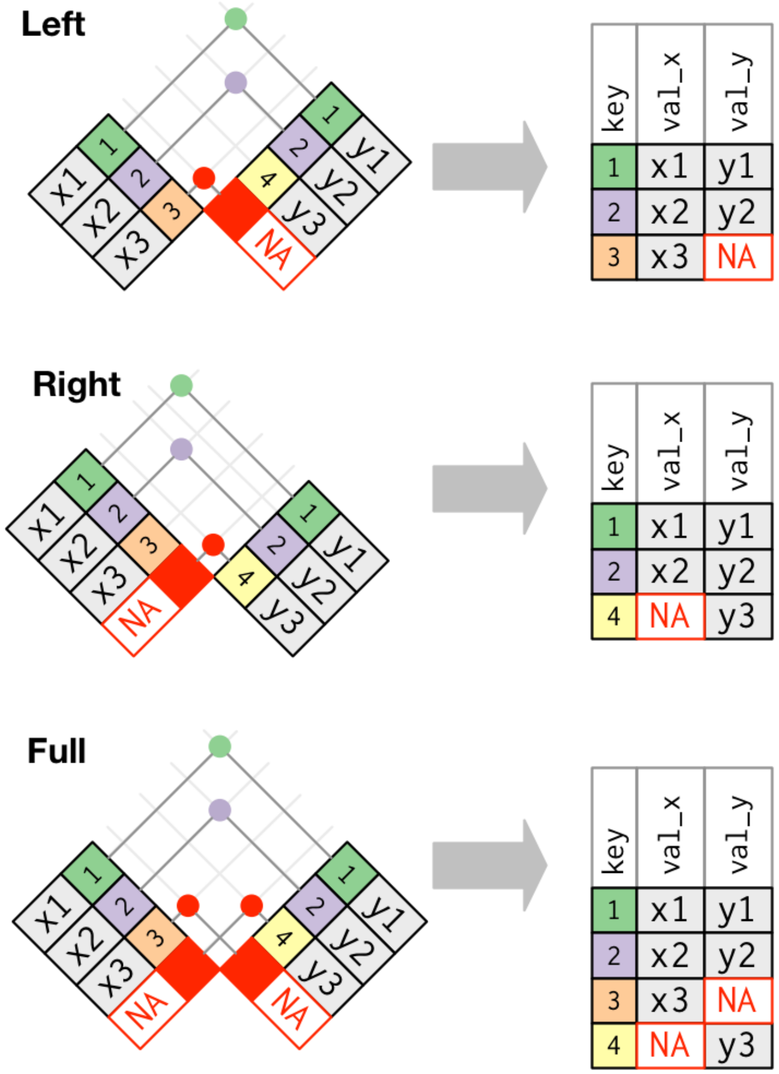
Outer joins

While **inner join** keeps observations that appear in both tables, outer join keeps observations that appear in at least one of the tables. There are **3** types of outer joins:

- A left join keeps all observations in x,
- A right join keeps all observations in y,
- A full join keeps all observations in x and y.

These joins work by adding an additional "virtual" observation to each table.

- This observation has a key that always matches (if no other key matches), and a value filled with **NA**.



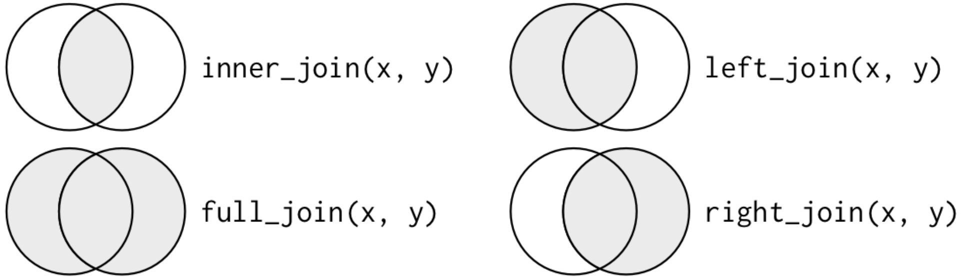
Left join

The most commonly used join is the left join: you use this whenever you look up additional data from another table, because it **preserves** the **original observations** even when there **isn't a match**.

The left join should be your **default** join: use it unless you have a strong reason to prefer one of the others.

Venn diagram

Another way to depict the different types of joins is with a Venn diagram.

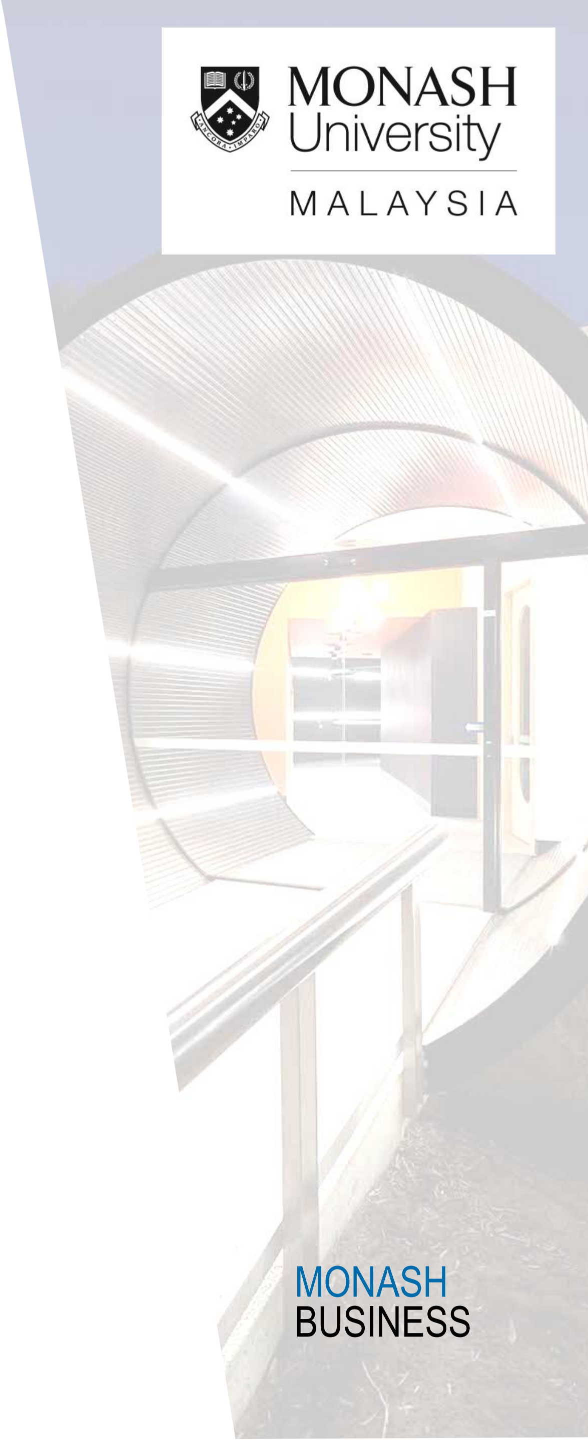


This however, is not a great representation. It might jog your memory about which join preserves the observations in which table, but it suffers from a **major limitation**: a Venn diagram can't show what happens when keys don't uniquely identify an observation.

MONASH
BUSINESS

Outline

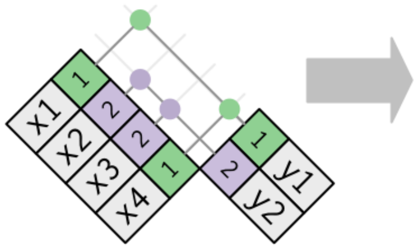
- ✓ Joining Data with dplyr
    - ✓ Keys
    - ✓ Mutating Joins
    - ❑ **Duplicate Keys**
    - ❑ Filtering Joins



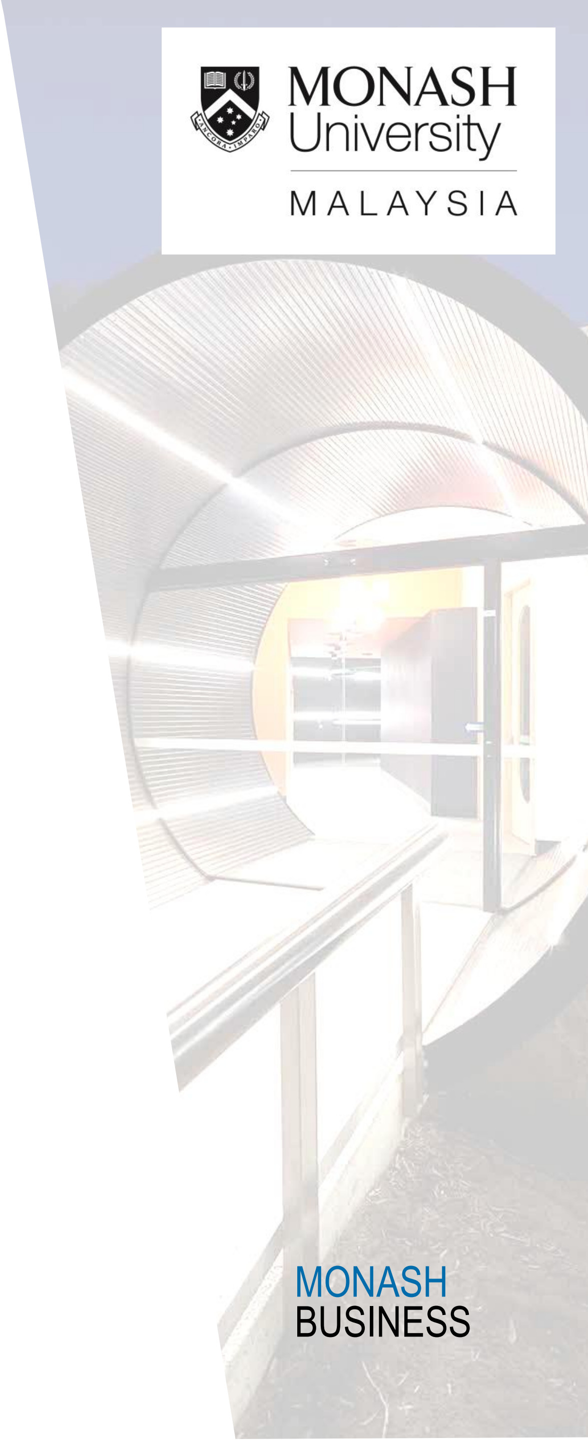
Duplicate keys

All diagrams have assumed that keys are unique. This is not always the case. Listed following is what happens when keys are not unique.

One **table** has **duplicate keys**. This is useful when you want to add in additional information as there is typically a one-to-many relationship. In figure below, primary key in y and foreign key in x.

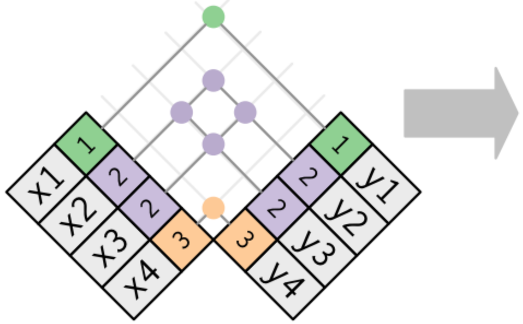


val_x	key	val_y
x1	1	y1
x2	2	y2
x3	2	y2
x4	1	y1



Cartesian product

Both **tables** have duplicate keys. This is usually an error as in neither table do the keys uniquely identify an observation. When joining duplicated keys, you get all possible combinations, the Cartesian product.



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

Other implementations

All 4 types of **mutating joins** can be performed by `base::merge()`.

```
dplyr               merge
inner_join(x, y)    merge(x, y)
left_join(x, y)     merge(x, y, all.x = TRUE)
right_join(x, y)    merge(x, y, all.y = TRUE),
full_join(x, y)     merge(x, y, all.x = TRUE, all.y = TRUE)
```

The advantages of the specific **dplyr** verbs is that they more clearly convey the intent of your code: the difference between the joins is really important but concealed in the arguments of `merge()`.

- dplyr's joins are considerably **faster** and don't mess with the **order** of the rows.

MONASH University MALAYSIA

MONASH BUSINESS

Defining the key columns

Pairs of tables have **always** been joined by a single variable, and that variable has the **same name** in both tables. That constraint was encoded using `by = "key"`.

The default, `by = NULL`, uses all variables that appear in both tables, the so called natural join. As an example, the flights and weather tables match on their common variables: year, month, day, hour and origin.

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

```
Joining, by = c("year", "month", "day", "hour", "origin")
```

A tibble: 336776 × 18

year	month	day	hour	origin	dest	tailnum	carrier	temp	dewp	humid	wind_dir	wind_speed	wind_gust	precip	pressure	visib	time_hour
<int>	<int>	<int>	<dbl>	<chr>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>
2013	1	1	5	EWR	IAH	N14228	UA	39.02	28.04	64.43	260	12.65858	NA	0	1011.9	10	2013-01-01 05:00:00
2013	1	1	5	LGA	IAH	N24211	UA	39.92	24.98	54.81	250	14.96014	21.86482	0	1011.4	10	2013-01-01 05:00:00
2013	1	1	5	JFK	MIA	N619AA	AA	39.02	26.96	61.63	260	14.96014	NA	0	1012.1	10	2013-01-01 05:00:00

Defining the key columns

A character vector, `by = "x"`. This is like a natural join, but uses only some of the common variables. As an example, flights and planes have year variables, but they mean different things so we only want to join by **tailnum**.

```r
flights2 %>%
  left_join(planes, by = "tailnum")
```

A tibble: 336776 × 16

year.x	month	day	hour	origin	dest	tailnum	carrier	year.y	type	manufacturer	model	engines	seats	speed	engine
<int>	<int>	<int>	<dbl>	<chr>	<chr>	<chr>	<chr>	<int>	<chr>	<chr>	<chr>	<int>	<int>	<int>	<chr>
2013	1	1	5	EWR	IAH	N14228	UA	1999	Fixed wing multi engine	BOEING	737-824	2	149	NA	Turbo-fan
2013	1	1	5	LGA	IAH	N24211	UA	1998	Fixed wing multi engine	BOEING	737-824	2	149	NA	Turbo-fan
2013	1	1	5	JFK	MIA	N619AA	AA	1990	Fixed wing multi engine	BOEING	757-223	2	178	NA	Turbo-fan

Defining the key columns

A named character vector: `by = c("a" = "b")`. This matches variable **a** in table **x** to variable **b** in table **y**. The variables from x will be used in the output.

As an example, if we want to draw a map, we need to **combine** the flights data with the airports data which contains the location (lat and lon) of each airport. Each flight has an origin and destination airport, so we need to specify which one we want to join to.

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

A tibble: 336776 × 15

year	month	day	hour	origin	dest	tailnum	carrier	name	lat	lon	alt	tz	dst	tzone
<int>	<int>	<int>	<dbl>	<chr>	<chr>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<chr>	<chr>
2013	1	1	5	EWR	IAH	N14228	UA	George Bush Intercontinental	29.98443	-95.34144	97	-6	A	America/Chicago
2013	1	1	5	LGA	IAH	N24211	UA	George Bush Intercontinental	29.98443	-95.34144	97	-6	A	America/Chicago
2013	1	1	5	JFK	MIA	N619AA	AA	Miami Intl	25.79325	-80.29056	8	-5	A	America/New_York

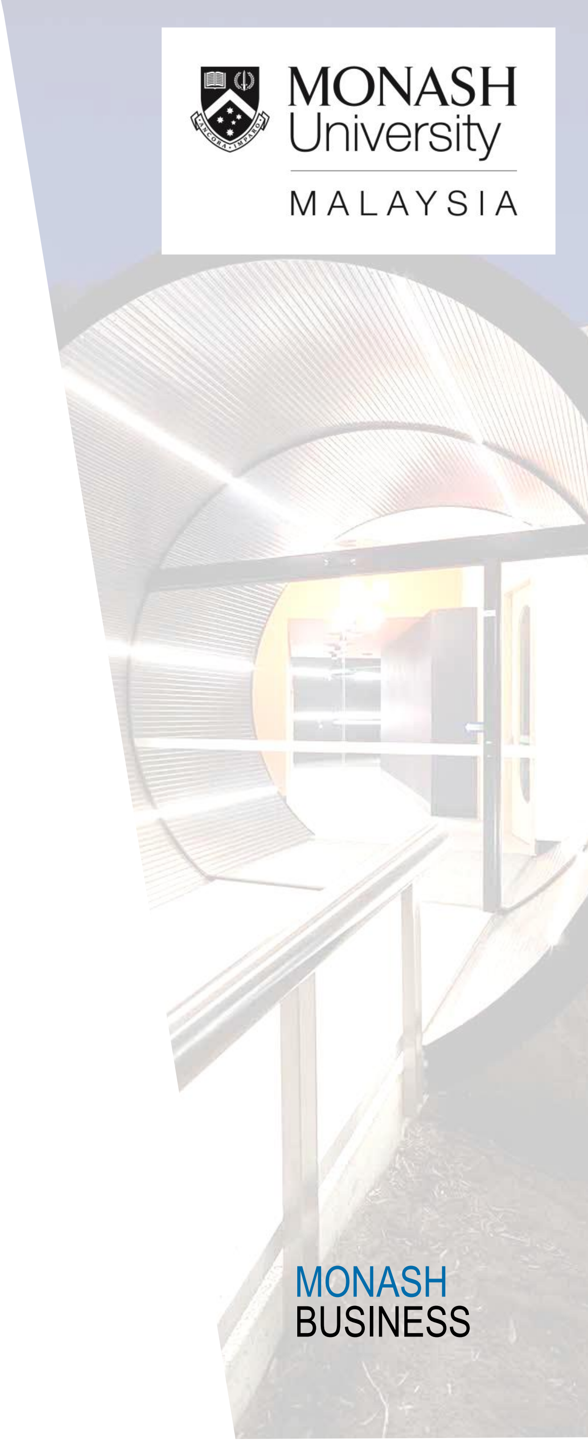
Defining the key columns

Another join, where origin is faa.

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

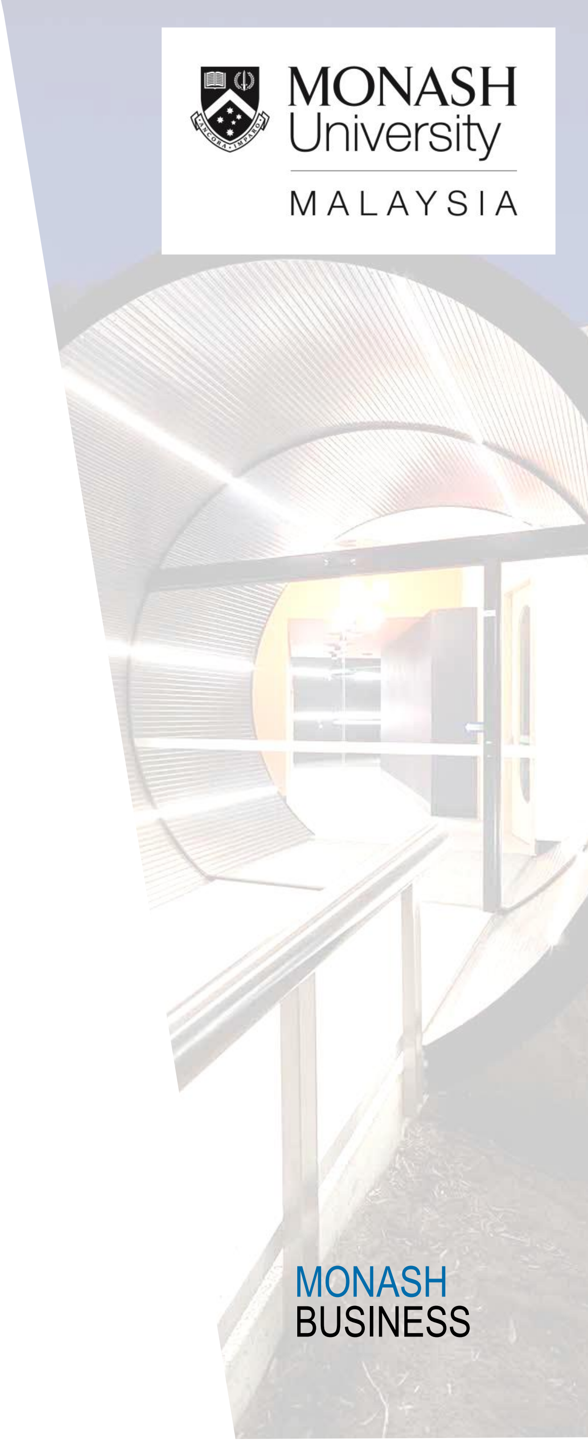
A tibble: 336776 × 15

year	month	day	hour	origin	dest	tailnum	carrier	name	lat	lon	alt	tz	dst	tzone
<int>	<int>	<int>	<dbl>	<chr>	<chr>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<chr>	<chr>
2013	1	1	5	EWR	IAH	N14228	UA	George Bush Intercontinental	29.98443	-95.34144	97	-6	A	America/Chicago
2013	1	1	5	LGA	IAH	N24211	UA	George Bush Intercontinental	29.98443	-95.34144	97	-6	A	America/Chicago
2013	1	1	5	JFK	MIA	N619AA	AA	Miami Intl	25.79325	-80.29056	8	-5	A	America/New_York



Outline

- ✓ Joining Data with dplyr
    - ✓ Keys
    - ✓ Mutating Joins
    - ✓ Duplicate Keys
    - ❑ Filtering Joins



Filtering joins

Filtering joins matches observations in same way as mutating joins, however, affects **observations**, not **variables**. There are 2 types:

- `semi_join(x, y)` keeps all observations in x that have a match in y,
- `anti_join(x, y)` drops all observations in x that have a match in y.

Semi-joins are useful for matching filtered summary tables back to the original rows. As an example, imagine you've found the top 10 most popular destinations.

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

A tibble: 10 × 2

dest	n
<chr>	<int>
ORD	17283
ATL	17215
LAX	16174
BOS	15508
MCO	14082
CLT	14064
SFO	13331
FLL	12055
MIA	11728
DCA	9705

Construct filter

Now let's find each **flight** that went to one of those **destinations**., using a filter.

```
flights %>%
  filter(dest %in% top_dest$dest)
```

A tibble: 141145 × 19

year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distance	hour	minute	time_hour
<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>	<int>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>
2013	1	1	542	540	2	923	850	33	AA	1141	N619AA	JFK	MIA	160	1089	5	40	2013-01-01 05:00:00
2013	1	1	554	600	-6	812	837	-25	DL	461	N668DN	LGA	ATL	116	762	6	0	2013-01-01 06:00:00
2013	1	1	554	558	-4	740	728	12	UA	1696	N39463	EWR	ORD	150	719	5	58	2013-01-01 05:00:00

MONASH University MALAYSIA

MONASH BUSINESS

Semi join

It is hard to extend the approach to **multiple variables**. As an example, imagine you have found the **10 days** with highest average delays. How would you construct the filter statement that used year, month, and day to match it back to flights?

Use a semi-join which connects the **2 tables** like a **mutating join**, but instead of adding new columns, it only keeps the rows in x that have a match in y.

```
flights %>%
  semi_join(top_dest)
```

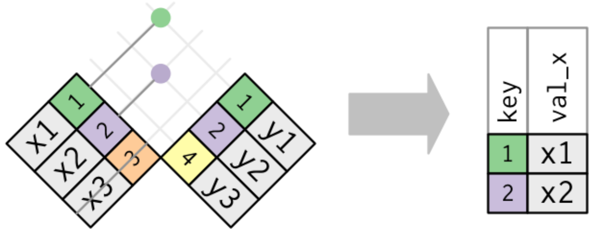
```
Joining, by = "dest"
```

A tibble: 141145 × 19

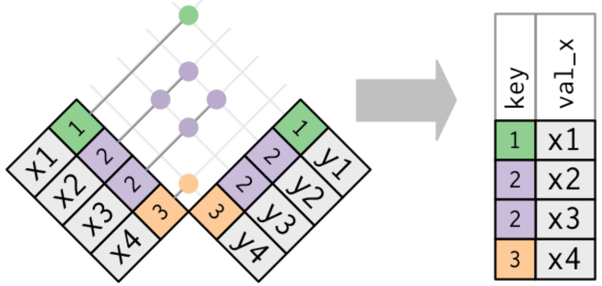
year	month	day	dep_time	sched_dep_time	dep_delay	arr_time	sched_arr_time	arr_delay	carrier	flight	tailnum	origin	dest	air_time	distance	hour	minute	time_hour
<int>	<int>	<int>	<int>	<int>	<dbl>	<int>	<int>	<dbl>	<chr>	<int>	<chr>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	<dbl>	<dttm>
2013	1	1	542	540	2	923	850	33	AA	1141	N619AA	JFK	MIA	160	1089	5	40	2013-01-01 05:00:00
2013	1	1	554	600	-6	812	837	-25	DL	461	N668DN	LGA	ATL	116	762	6	0	2013-01-01 06:00:00
2013	1	1	554	558	-4	740	728	12	UA	1696	N39463	EWR	ORD	150	719	5	58	2013-01-01 05:00:00

Joins

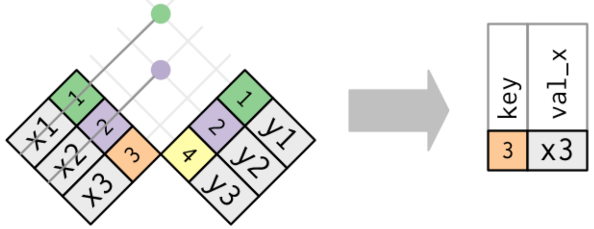
A semi-join graphically looks like this.



Only **existence** of a match is **important**; it doesn't matter which observation is matched. This means that filtering joins **never duplicate** rows like mutating joins do.



The inverse of a semi-join is an anti-join. An anti-join keeps the rows that don't have a match.



MONASH University MALAYSIA

MONASH BUSINESS

Anti joins

MONASH University MALAYSIA

Anti-joins are useful for diagnosing join mismatches. As an example, when connecting flights and planes, we might be interested to know that there are **many flights** that don't have a match in **planes**.

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

A tibble: 722 × 2

tailnum	n
<chr>	<int>
NA	2512
N725MQ	575
N722MQ	513
N723MQ	507
N713MQ	483
N735MQ	396

THANK YOU

FIND OUT MORE AT MONASH.EDU.MY
LIKE @MONASH UNIVERSITY MALAYSIA ON FACEBOOK
FOLLOW @MONASHMALAYSIA ON TWITTER