Unit 04:
Albus Dumbledore and the nycflights13 dataset

Applied AI with R

Ferdinand Ferber and Wolfgang Trutschnig

Paris Lodron Universität Salzburg

4/20/24

Table of contents I

Albus Dumbledore and the nycflights13 dataset

AI generated image for the prompt "Albus Dumbledore in his office in Hogwards, casting a powerful spell in front of his computer."

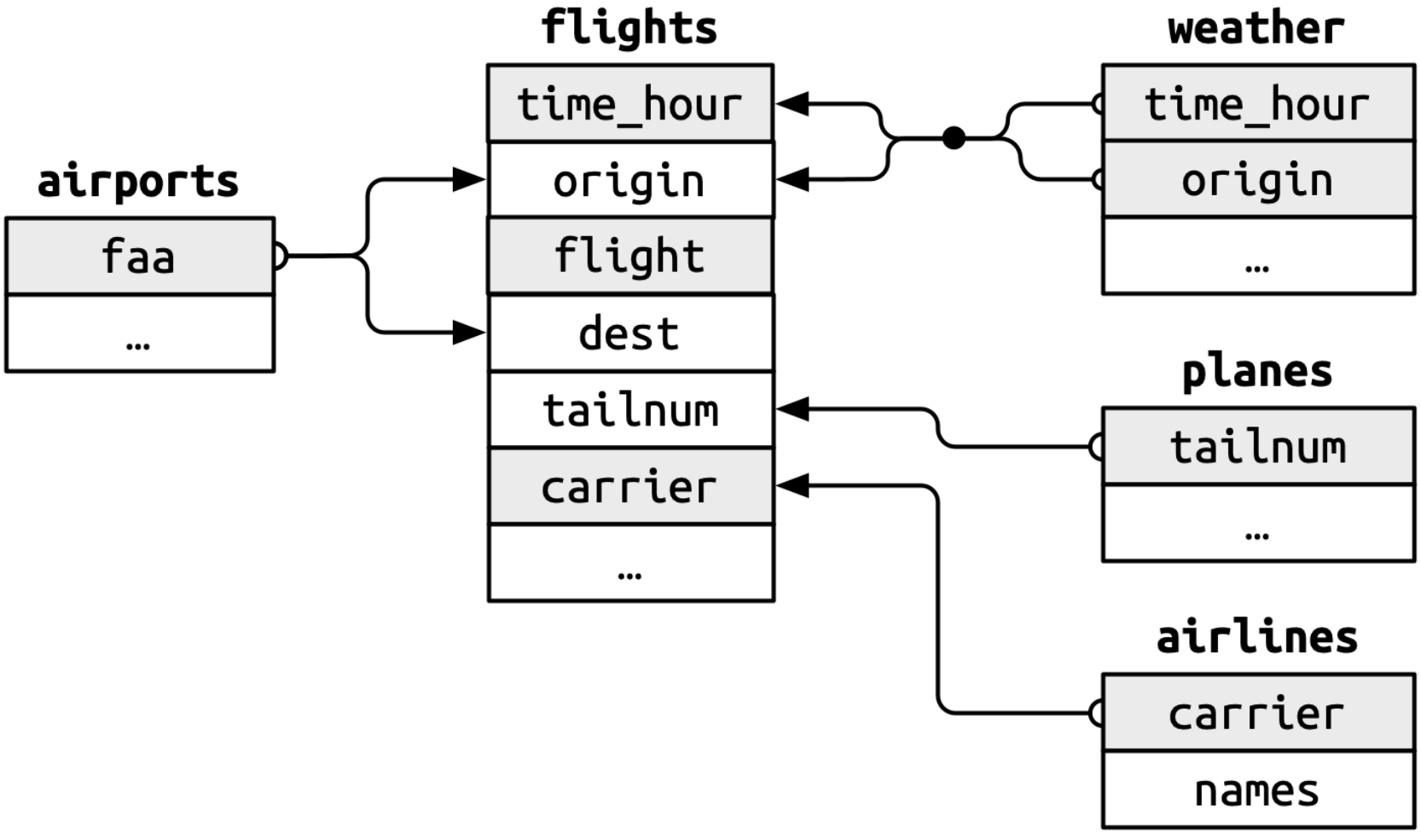
Albus Dumbledore and the nycflights13 dataset

- Lord Voldemort's Deat Eater found a new target for chaos and dispair in the Muggle world - messing with air traffic, causing delays and cancellation.
- To stop them, Dumbledore needs to be one step ahead of Voldemort's plans.
- Can he find the pattern in the `nycflights13` dataset that reaveals the Death Eater's plan?

The nycflights13 dataset

- We will use the nycflights13 dataset, which consists of several dataframes.
- This dataset is provided by the `nycflights13` package.
- Please install and load this package.

The nycflights13 dataset



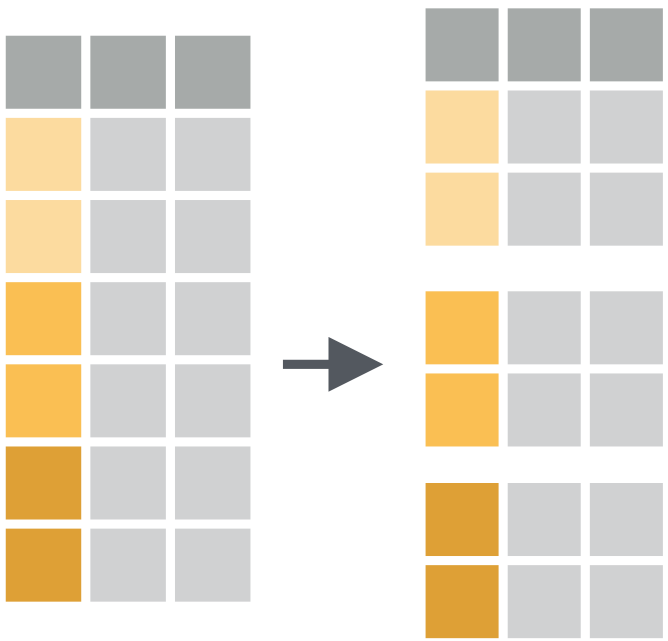
Section 1

Advanced data wrangling

Advanced data wrangling

- So far, we learned about the `{dplyr}` verbs `select()`, `filter()`, `mutate()`, `rename()`, `arrange()` and the family of `slice_*()` verbs to transform dataframes.
- Now we will introduce *grouped dataframes* and how these verbs interact with them.
- We'll see that grouping dataframes hugely increase our data wrangling capabilities.

Grouping[1]



- The `group_by` verb modifies the group structure of the input dataframe, which changes the behaviour of most verbs.

[1] Image taken from the *dplyr cheat sheet*, Posit Software, PBC

Inspecting the group structure

- The `group_by()` verbs takes a selection of columns and places all observations that are equal on those columns into the same group.
- For debugging purposes there is a function that returns the grouping variables as a character vector:

```
# Put all flights on the same date into the same group
flights |>
    group_by(year, month, day) |>
    group_vars()
```

```
[1] "year"  "month" "day"
```

Ungroup

- The group structure can be (partially) removed by the `ungroup()` verb:

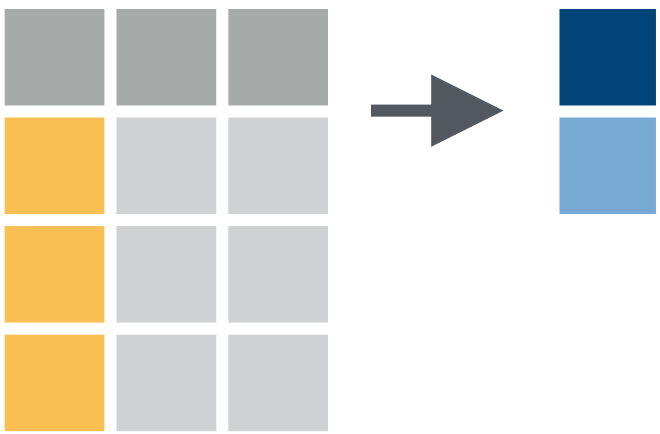
```
flights |>
  group_by(year, month, day) |>
  ungroup() |>
  group_vars()
```

```
character(0)
```

```
flights |>
  group_by(year, month, day) |>
  ungroup(month) |>
  group_vars()
```

```
[1] "year" "day"
```

Summarize[2]



- The `summarize` verb calculates aggregate/summary values for each column of the input dataframe.

[2] Image taken from the *dplyr cheat sheet*, Posit Software, PBC

Summarize

- The `summarize()` verb on an ungrouped dataframe is a bit boring.
- It returns a dataframe consisting of the specified columns and one row, where the entry is given by the provided summary expression.

```
flights |>
  summarize(mean_dep_delay = mean(dep_delay, na.rm=T),
            mean_arr_delay = mean(arr_delay, na.rm=T))
```

```
# A tibble: 1 x 2
  mean_dep_delay mean_arr_delay
           <dbl>          <dbl>
1           12.6           6.90
```

Summarize

- But on grouped dataframes, the summary is calculated for each group independently.

```
flights |>
  group_by(year, month, day) |>
  summarize(mean_dep_delay = mean(dep_delay, na.rm=T),
            mean_arr_delay = mean(arr_delay, na.rm=T)) |>
```

year	month	day	mean_dep_delay	mean_arr_delay
2013	12	22	29.2398649	23.8995485
2013	3	15	12.4356334	0.5907216
2013	4	26	18.5924413	21.3721881
2013	11	11	3.3387755	-4.5240041
2013	9	5	-0.3877973	-15.5403727

Summarize

- The `.groups` argument controls how the output dataframe is grouped.
- The default is like `drop_last`, but with a warning.

```
df <- flights |> group_by(year, month, day)

df |> summarize(n = n(), .groups = "drop_last") |>
    group_vars()
```

```
[1] "year"  "month"
```

```
df |> summarize(n = n(), .groups = "keep") |> group_vars()
```

```
[1] "year"  "month" "day"
```

```
df |> summarize(n = n(), .groups = "drop") |> group_vars()
```

```
character(0)
```

Summary functions[3]

- Some useful standard summary functions: `sum()`
  `mean()`
  `median()`
  `min()`
  `max()`
  `n()`
  `n_distinct()`

- Important: Notice, that you can also use any function that maps an atomic vector to a scalar value as a summary function.
- Nowadays the standard aggregation technique in R.

[3] Image taken from the *dplyr cheat sheet*, Posit Software, PBC

mutate

- `mutate` will be applied for each group independently:

```
flights |>
    mutate(rank_global = min_rank(arr_delay)) |>
    group_by(carrier) |>
    mutate(rank_group = min_rank(arr_delay))
```

carrier	flight	arr_delay	rank_global	rank_group
EV	4998	-18	70876	7892
B6	74	21	260489	41844
DL	2331	-4	165574	27224
VX	169	1	194343	3371
B6	653	7	221520	35095

filter

- `filter()` can be used in combination with a summary function.

```
flights |>
  group_by(carrier) |>
  filter(distance == max(distance))
```

carrier	flight	origin	dest	distance
DL	435	JFK	SFO	2586
VX	29	JFK	SFO	2586
DL	2065	JFK	SFO	2586
AS	15	EWR	SEA	2402
AA	85	JFK	SFO	2586

- …`slice_max()` would be more readable/convenient here:

Slicing

- The `slice_*()` verbs also work group-wise:

```
flights |>
  group_by(origin, dest) |>
  slice_max(dep_delay)
```

carrier	flight	origin	dest	dep_delay
B6	163	JFK	SRQ	194
UA	503	EWR	AUS	351
B6	1185	JFK	RDU	394
DL	2047	LGA	ATL	898
EV	4885	LGA	ILM	168

Exercise

- How does the average departure delay develop over months? Use `group_by` and `summarize` to calculate it and plot it using a suitable `{ggplot2}` geom.
- Which of the three NYC airports have a better on-time percentage for departing flights?
- Hint: First classify every flight as *on time* or *delayed*. Then group flights by origin airport and calculate the percentage using the `summarize` verb. Use `sum()` and `n()`.
- Which plane (given by its `tailnum`) has the highest average speed?

mutate and summarize on multiple columns

- Sometimes it's useful to apply the same transformation across multiple columns.
- `across` has two primary arguments: The first arguments selects columns to operate on (with syntax like in `select`) and the second is a (list of) function(s) to apply to each selected column.

```
flights |>
    summarize(dep_delay = mean(dep_delay),
              arr_delay = mean(arr_delay))

flights |>
    summarize(across(c(dep_delay, arr_delay), mean))
```

where

- To select columns satisfying a given predicate function, use `where`:

```
flights |>
    summarize(across(where(is.character), n_distinct))
```

```
# A tibble: 1 x 4
  carrier tailnum origin  dest
    <int>   <int>  <int> <int>
1      16    4044      3   105
```

filter on multiple columns

- There are two special companion functions for `filter` to do filtering based on multiple columns:
- `if_any(.cols, .fns)` keeps all rows where the predicate is true for *at least one* selected columns. Analogous for `if_all(.cols, .fns)`.

```
# Keeps all rows where at least one column is not NA
flights |>
    filter(if_any(everything(), \(x) !is.na(x)))
```

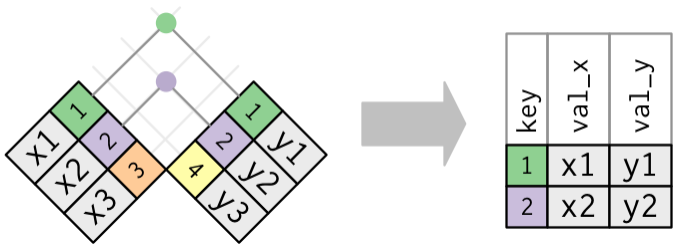
Exercise

- Use `mutate()` together with `across()` to transform every string column to a factorial column.
- Calculate the median for every numerical column.
- Use `filter()` and `if_any()` or `if_all()` to select all flights that had more than 60 minutes delay in any column containing the word "delay".

Combining dataframes

- Data comes often in a `relational` form, where it is split across a number of tables/dataframes which contain cross references.
- There are three families of verbs that work with two tables at a time:
- **Joins** add new variables to a dataframe from matching rows in another
- **Filtering joins** filter observations from a dataframe based on whether or not they match an observation in another
- **Set operations** treat observations as set elements

Inner Join[4]

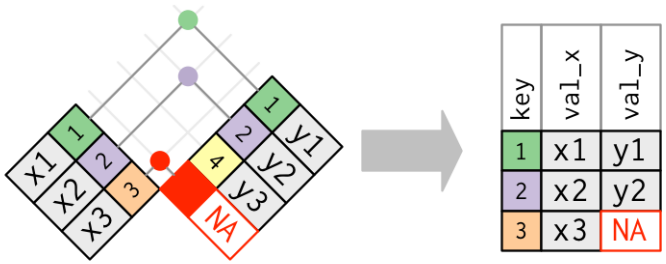


- An *inner join* matches pairs of observations whenever their keys are equal.

- Syntax in R: `inner_join(x, y, by)`
- the by argument is option: per default, R uses ALL column names appearing in both dataframes (also for the other joining types).

[4] Image taken from *R for Data Science*, Wickham and Grolemund

Left Join[5]



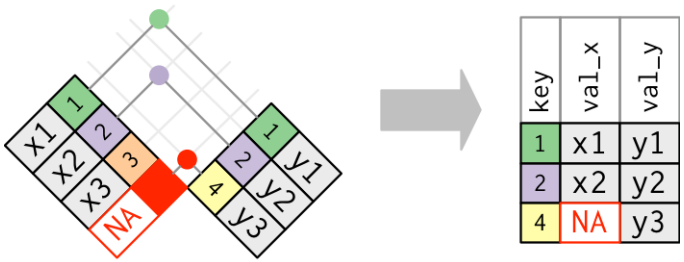
key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA

- A *left join* keeps all observations in x and adds matches from y.
- If no match is found, all variables are filled with `NA`

- Syntax in R: `left_join(x, y, by)`

[5] Image taken from *R for Data Science*, Wickham and Grolemund

Right Join[6]

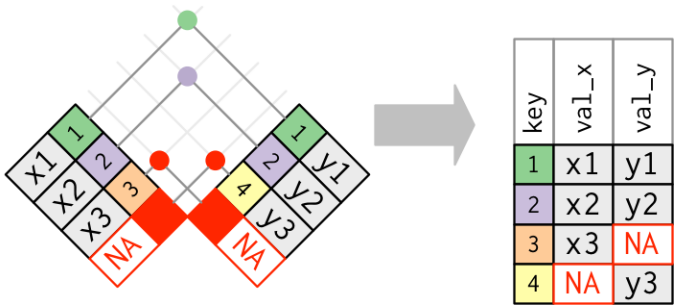


- A *right join* keeps all observations in y and adds matches from x.
- If no match is found, all variables are filled with `NA`

Syntax in `R`: `right_join(x, y, by)`

[6] Image taken from *R for Data Science*, Wickham and Grolemund

Full Join[7]



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

- A *full join* keeps all observations in x and y.
- Syntax in R: `full_join(x, y, by)`

[7] Image taken from *R for Data Science*, Wickham and Grolemund

Example: Joins

- In the following example we have a natural left join (the join variables are all common variables of both dataframes).
- Columns `year:origin` are the keys, column `dest` is from `flights` and column `temp` is from `weather`.

```
flights |> left_join(weather) |>
    select(year:day, hour, origin, dest, temp)
```

year	month	day	hour	origin	dest	temp
2013	4	30	11	JFK	LAX	62.06
2013	7	24	10	EWR	ATL	82.94
2013	5	13	22	LGA	SYR	50.00
2013	8	25	19	LGA	ORD	75.02
2013	6	6	16	JFK	LAX	64.94

Join attributes

- We can control the join attributes via the `by` argument.
- If the variable names differ, we use `join_by(leftcol == rightcol)` to join on the columns `leftcol` and `rightcol`.
- If the variable name is the same in both dataframes, on can use the shortcut `join_by(commoncol)`.

```
flights |> left_join(planes, by = join_by(tailnum)) |>
    select(origin, dest, tailnum, model)
```

origin	dest	tailnum	model
JFK	BUF	N656JB	A320-232
LGA	MCO	N984DL	MD-88
EWR	IND	N13949	EMB-145LR
JFK	DCA	N930XJ	CL-600-2D24
JFK	SAT	N924XJ	CL-600-2D24

Join suffix

- When a non-join-attribute variable is in both dataframes, a suffix will be appended to resolve disambiguities.
- The default suffix is `.x` for the left table and `.y` for the right one. This can be modified:

```
flights |>
    left_join(planes, by = join_by(tailnum),
              suffix = c(".flight", ".plane")) |>
    select(year.flight, origin, dest,
           tailnum, year.plane)
```

year.flight	origin	dest	tailnum	year.plane
2013	EWR	TUL	N41104	2002
2013	JFK	BOS	N3CMAA	NA
2013	JFK	JAX	N229JB	2006
2013	EWR	FLL	N16234	1999

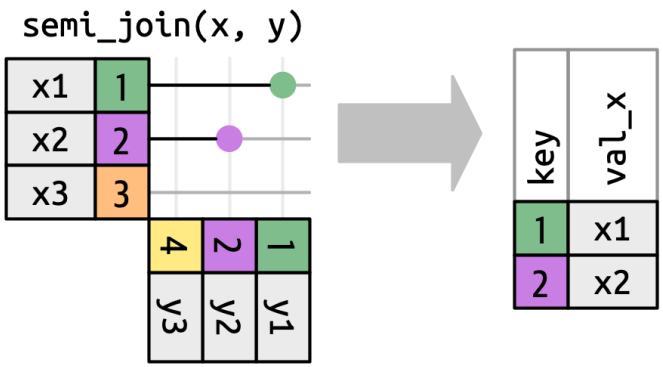
Exercise

- Recreate the following dataframe:

year	month	day	holiday
2013	1	1	New Years Day
2013	7	4	Independence Day
2013	11	29	Thanksgiving Day
2013	12	25	Christmas Day

- Join the `flights` dataframe with this `special_days` dataframe. Which type of join is appropriate?
- Compare the number and delays of flights on normal days and on these special days.
- Produce an informative plot on the comparison.

Semi Join[8]

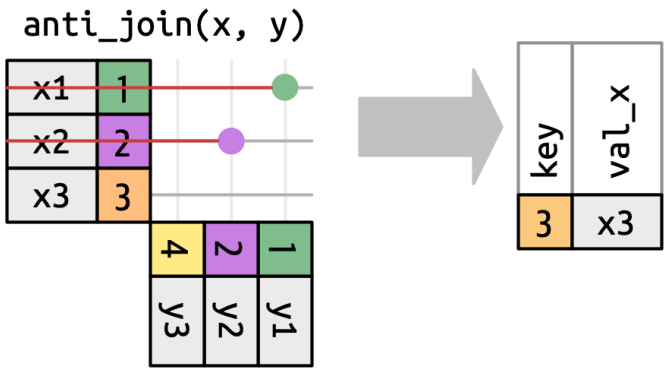


- A *semi join* returns all rows of x that have a match in y, i.e. every observation of x that would be included in an inner join.

- Syntax in R: `semi_join(x, y, by)`

[8] Image taken from *R for Data Science*, Wickham and Grolemund

Anti Join[9]



- An *anti join* returns all rows of x that have *no* match in y, i.e. every observation of x that would be excluded in an inner join.
- It can be seen as a filter via different dataframes.

- Syntax in R: `anti_join(x, y, by)`

[9] Image taken from *R for Data Science*, Wickham and Grolemund

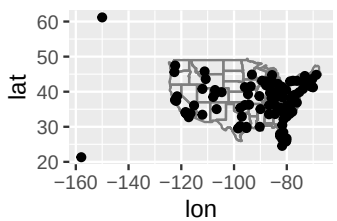
Exercise

- Use an appropriate filtering join to filter to all airports (in the `airport` dataframe) that appear as a destination in the `flights` dataframe

Exercise

- Consider the following code (install the `maps` package first):

```
your_filtered_airports |>
  ggplot(aes(x = lon, y = lat)) + borders("state") +
  geom_point() + coord_quickmap()
```



- Colour each airport by the average destination delay.

Exercise

- Use an appropriate filtering join to figure out which flights don't have a `tailnum` that corresponds to an entry in the `planes` dataframe
- What carriers are dominating this list?
- What kind of flights are those in

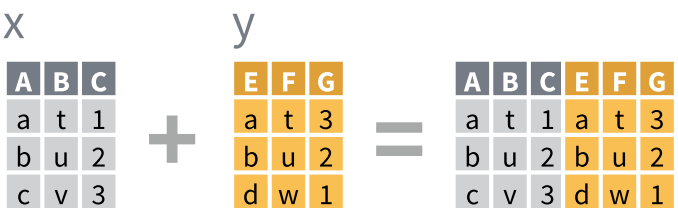
```
anti_join(flights, airports,
          by = join_by(dest == faa))
```

Set operations

One can treat dataframes as sets (of rows) and do the usual set operations on them. All operations ingnore duplicates.

- `intersect(df1, df2)` returns the common rows
- `union(df1, df2)` returns all rows of both dataframes
- `setdiff(df1, df2)` returns all rows in `df1` that are not in `df2`
- `symdiff(df1, df2)` returns the symmetric difference, i.e. all rows that are in `df1` but not in `df2` plus all rows that are in `df2` but not in `df1`

Binding columns[10]

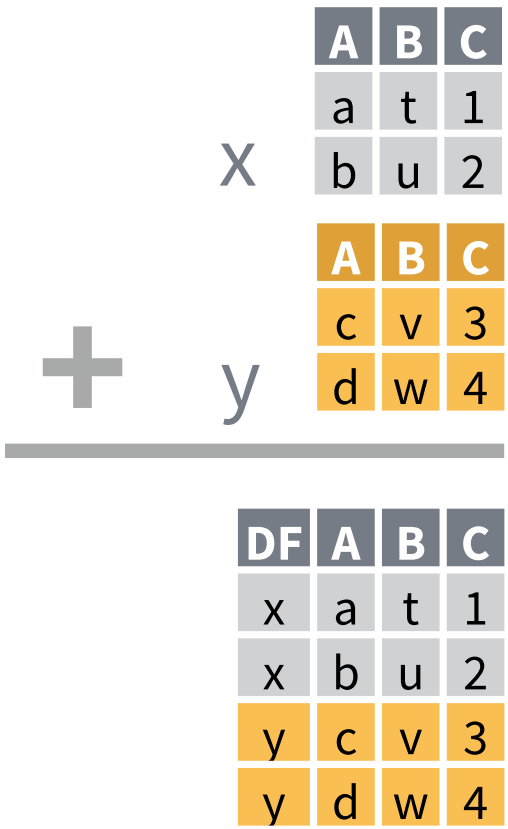


- Returns `df1` and `df2` placed side by side as a single table.

Syntax in `R`: `bind_cols(df1, df2)`

[10] Image taken from the *dplyr cheat sheet*, Posit Software, PBC

Binding rows[11]



- Returns `df1` and `df2` placed on top of the other as a single table.

Syntax in `R`: `bind_rows(df1, df2)`

[11] Image taken from the *dplyr cheat sheet*, Posit Software, PBC

Nesting

- R allows dataframes to contain columns of dataframes.
- This is called a *nested* dataframe and it's useful for functions that work on whole dataframes (e.g. modeling functions).

```
flights |> group_by(origin) |> nest()
```

```
# A tibble: 3 x 2
# Groups:   origin [3]
  origin data
  <chr>  <list>
1 EWR    <tibble [120,835 x 18]>
2 LGA    <tibble [104,662 x 18]>
3 JFK    <tibble [111,279 x 18]>
```

Unnesting

- The inverse operation of `nest()` is `unnest(col)` and will expand all dataframes in `col`.

Example: Nesting

- Nesting can be very useful when we want to work with groups as a whole.

```
# Randomly select two groups
flights |>
  group_by(tailnum) |>
  nest() |>
  ungroup() |>
  slice_sample(n = 2) |>
  unnest(data)
```

tailnum	year	month	day	dep_time
N615QX	2013	6	18	1053
N615QX	2013	3	6	1506
N543UW	2013	5	1	1257

Section 2

Advanced data visualization

Advanced data visualization

- So far we acquired a fair amount of `{ggplot2}` knowledge that lets us create a wide range of useful and beutiful plots.
- Now it's time to dig deeper into the *grammar of graphics* and the underlaying principles of `{ggplot2}`.

The components of a ggplot

Every `ggplot` consists of

- A default dataframe
- One or more layers
- For each mapped aeasthetics one scale
- A coordinate system
- A faceting specification

Every aesthetics comes with a default scale. The default coordiate system is `coord_cartesian()` and the default faceting specification is `facet_null()`.

The components of a layer

Each layer consists of - An input dataframe - A selected geom - An aesthetical mapping - A statistical transformation - A position adjustment

You already know the input dataframe, the geom and the aesthetical mapping. If no input dataframe is supplied for the layer, then the default dataframe for the whole plot will be used automatically.

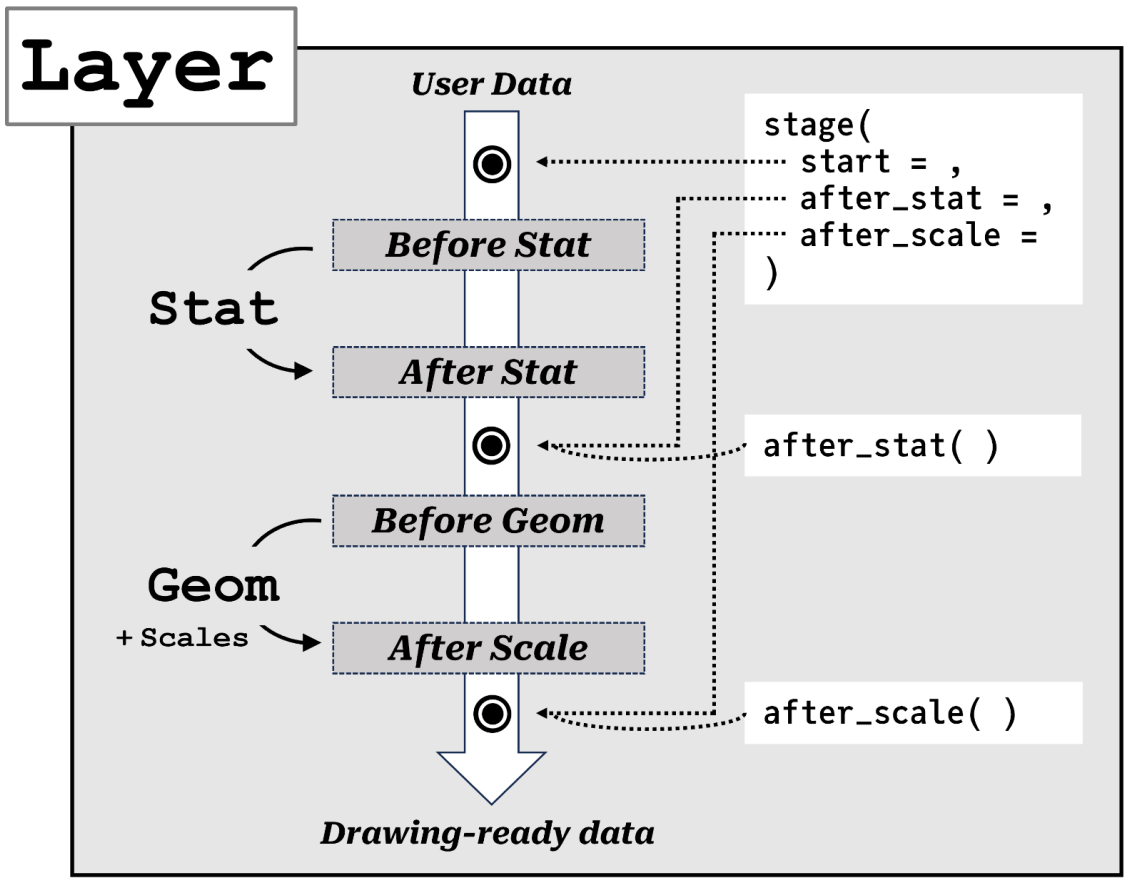
Types of geoms

There are two types of geoms:

- **Individual geoms** (e.g. points) correspond to exactly one observation in the input dataframe.
- **Collective geoms** (e.g. boxplots) correspond to a group of observations in the input dataframe.
- Every geom has a set of aesthetics (visual properties). A point for example has an x position, a y position, a colour, a shape and so on. A boxplot has a width, a median value, a lower and upper hinge (the 25% and 75% quantiles) and so on.
- The aesthetic mapping defines which column in the input dataframe corresponds to which aesthetics. In the boxplot case there are obviously no columns for median and quantiles in the input dataframe, so how does that work?

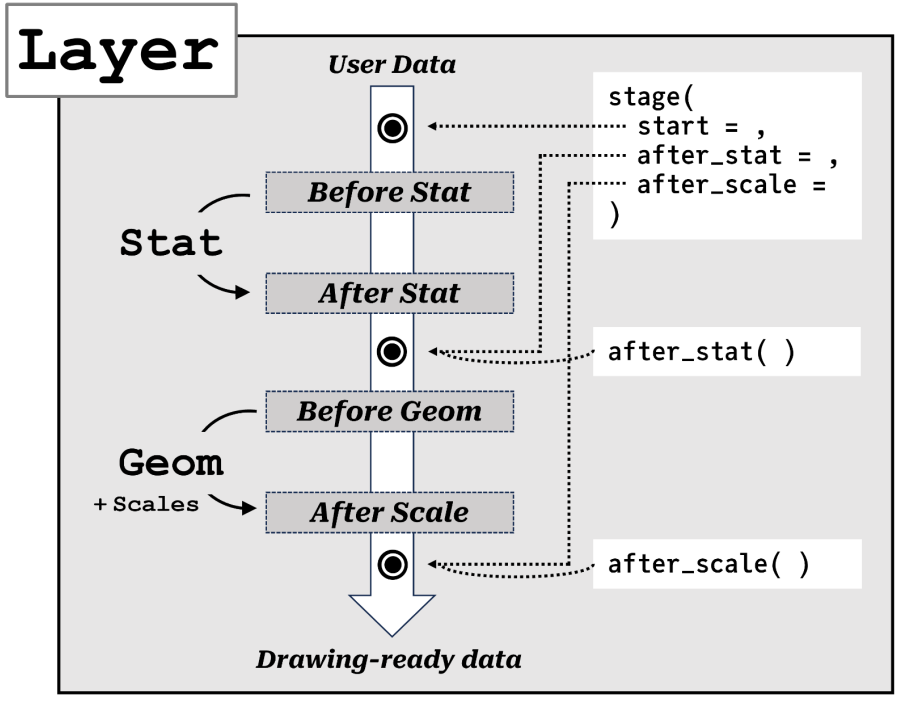
Sublayer modularity[12]

- The input dataframe undergoes several transformations before it is ready for drawing:



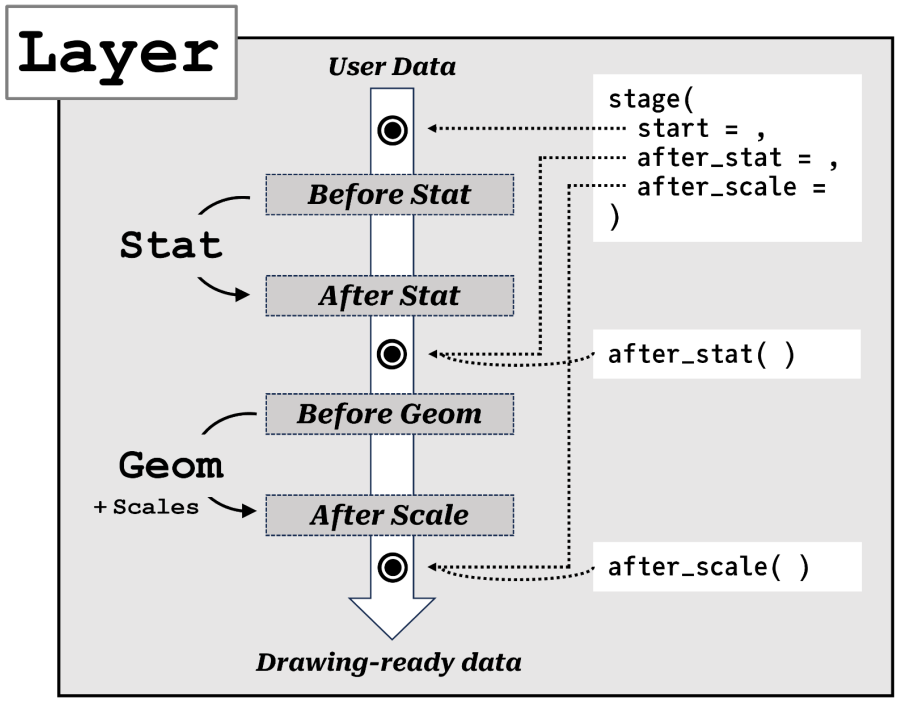
[12]This image was taken from the manual of {ggtrace}

Sublayer modularity



- In the first step, {ggplot2} takes the input dataframe and computes, based on the *start aesthetical mapping*, a new dataframe (usually this is just a select operation, but you can do everying you can do in a `mutate()` call also in an `aes()` call).
- Then the statistical transformation is applied (e.g. calculating the quantile information for boxplot)

Sublayer modularity



- Then the geom can transform this dataframe further (usually it doesn't) and the scales are applied, mapping from the data space to the aesthetical space
- Finall, the *after scales aesthetical mapping* is applied and the result (called *layer data*) is shipped back to the drawing engine.

Sublayer modularity

- Usually, the user doesn't need to think about the fact that there are three different aesthetical mappings.
- The following calls are identical due to {ggplot2}s defaults:

```
flights |> ggplot() +
  geom_bar(aes(x = origin, y = after_stat(count)))
```

```
flights |> ggplot() +
  geom_bar(aes(x = origin))
```

Statistical transformations

Notice that every layer has a statistical transformation. For `geom_point()` we can see that it is the *identity transformation*:

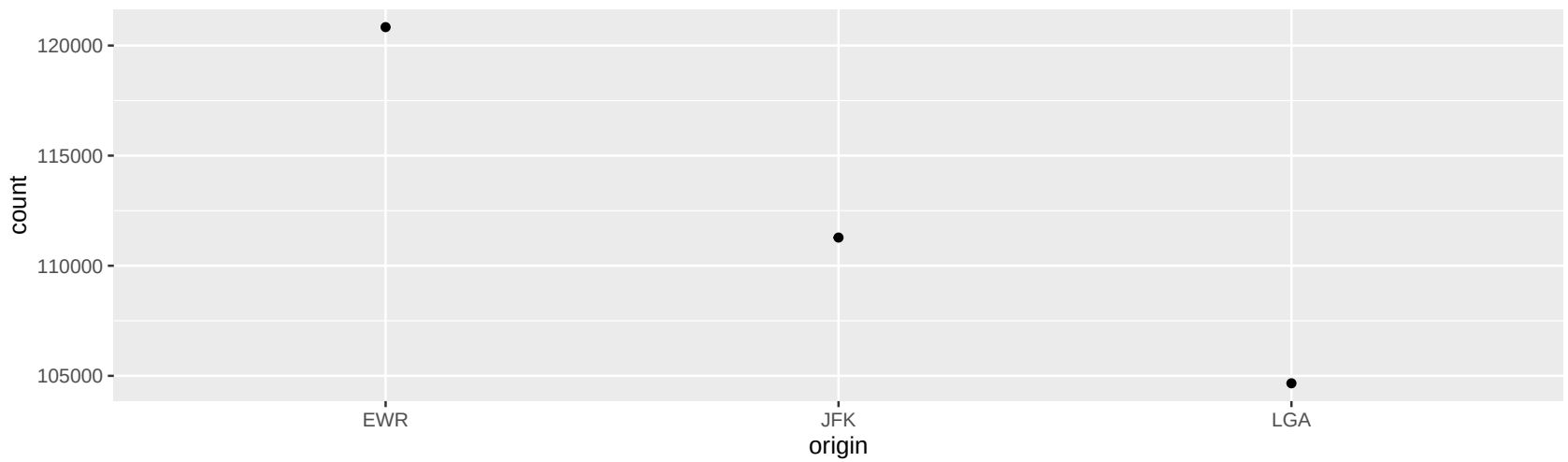
```
  geom_point
```

```
function (mapping = NULL, data = NULL, stat = "identity", position = "identity"
    ..., na.rm = FALSE, show.legend = NA, inherit.aes = TRUE)
{
    layer(data = data, mapping = mapping, stat = stat, geom = GeomPoint,
        position = position, show.legend = show.legend, inherit.aes = inherit.a
        params = list2(na.rm = na.rm, ...))
}
<bytecode: 0x000001fb99ad0898>
<environment: namespace:ggplot2>
```

Statistical transformations

- As you saw on the previous slide, you can override the `stat` argument of a layer. Here we use the `count` stat on a `point` geom:

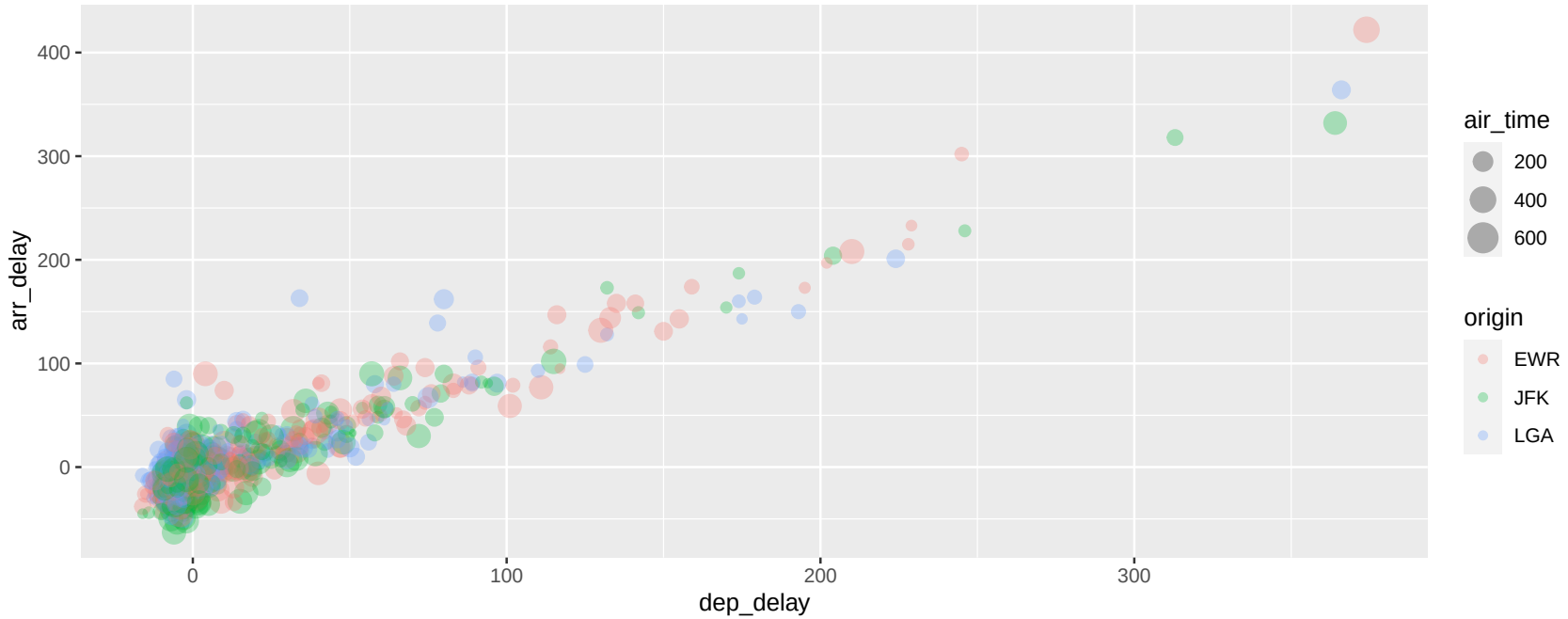
```
flights |> ggplot() +
  geom_point(aes(x = origin, y = after_stat(count)),
             stat = "count")
```



Scales

Consider the following plot:

```
flights |> ggplot(aes(x = dep_delay, y = arr_delay,
                      colour = origin,
                      size = air_time)) +
  geom_point(alpha = 0.3)
```



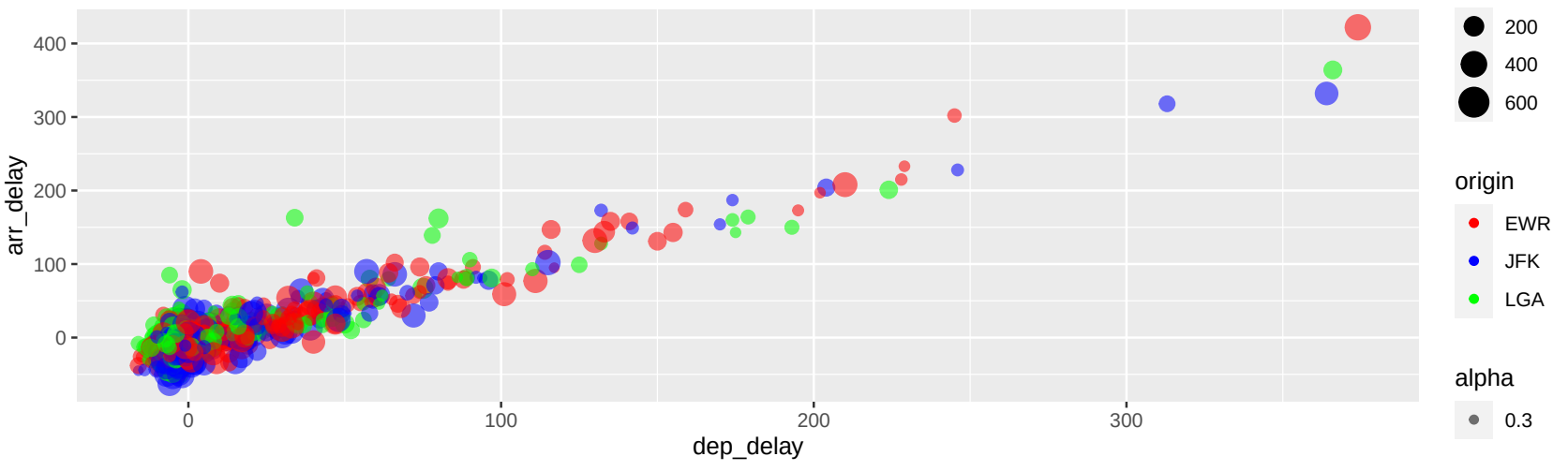
Scales

- In the last example we mapped the column `origin` to the aesthetics `colour`, but we never specified which origin airport (JFK, EWR, LGA) got which colour (red, green, ...). This was done by the *scale* of that aesthetics.
- A scale consists of a function that maps numeric values (*continuous scale*) or factorial values (*discrete scale*) to the possible values of the aesthetics, and its inverse. The inverse function is called *guide* and is used to generate the legends (which colour denotes which airport?).
- Every aesthetics brings its own default scales (one discrete and one continuous), so we didn't need to specify it in the last example.

Scales

It doesn't make sens to have different scales for each layer, so scales are added to the plot globally:

```
flights |> ggplot(aes(x = dep_delay, y = arr_delay,
                      colour = origin, size = air_time,
                      alpha = 0.3)) +
    geom_point() +
    scale_colour_manual(values=c("red", "blue", "green"))
```



Position adjustments

There are five options to adjust positioning:

- `position = "identity"` will place each object exactly where it falls in the context of the graph. In the case of our bar plot the bars would overlap.
- `position = "stack"` will place objects ontop of each other
- `position = "fill"` is similar to stacking, but makes each set of stacked bars the same height (for comparing proportions).
- `position = "dodge"` places overlapping objects beside one another.
- `position = "jitter"` adds a small amount of noise to each object to combat overplotting

Position adjustments

Here is an example for the first four position adjustments in the case of a bar plot: