

Data Manipulation

Jim Harner

1/12/2021

The `dplyr` package is part of the `tidyverse`. It provides a grammar of data manipulation using a set of verbs for transforming tibbles (or data frames) in R or across various backend data sources.

```
library(dplyr, warn.conflicts = FALSE)
library(lubridate)
```

```
##
## Attaching package: 'lubridate'
##
## The following objects are masked from 'package:dplyr':
##
##   intersect, setdiff, union
##
## The following objects are masked from 'package:base':
##
##   date, intersect, setdiff, union
```

This section illustrates `dplyr` using the NYC flight departures data as a context.

```
library(nycflights13)
```

3.1 Data manipulation with `dplyr`

This section explores the main functions in `dplyr` which Hadley Wickham describes as a *grammar of data manipulation*—the counterpoint to his *grammar of graphics* in `ggplot2`.

The github repo for `dplyr` not only houses the R code, but also vignettes for various use cases. The introductory vignette is a good place to start and can be viewed by typing the following on the command line: `vignette("dplyr", package = "dplyr")` or by opening the `dplyr.Rmd` file in the vignettes directory of the `dplyr` repo. The material for this section is based on content from Hadley Wickham's Introduction to `dplyr` Vignette.

`dplyr` was designed to:

- provide commonly used data manipulation tools;
- have fast performance for in-memory operations;
- abstract the interface between the data manipulation operations and the data source.

`dplyr` operates on data frames, but it also operates on tibbles, a trimmed-down version of a data frame (`tbl_df`) that provides better checking and printing. Tibbles are particularly good for large data sets since they only print the first 10 rows and the first 7 columns by default although additional information is provided about the rows and columns.

The real power of `dplyr` is that it abstracts the data source, i.e., whether it is a data frame, a database, or Spark.

All the `dplyr` vignettes use the `nycflights13` data which contain the 336,776 flights that departed from New York City in 2013. The `flights` tibble is one of several data sets in the package.

```
dim(flights)
```

```
## [1] 336776      19
```

```
flights # or print(flights)
```

```
## # A tibble: 336,776 x 19
```

```
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
## 1  2013     1     1     517           515           2     830           819
## 2  2013     1     1     533           529           4     850           830
## 3  2013     1     1     542           540           2     923           850
## 4  2013     1     1     544           545          -1    1004          1022
## 5  2013     1     1     554           600          -6     812           837
## 6  2013     1     1     554           558          -4     740           728
## 7  2013     1     1     555           600          -5     913           854
## 8  2013     1     1     557           600          -3     709           723
## 9  2013     1     1     557           600          -3     838           846
##10  2013     1     1     558           600          -2     753           745
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dtm>
```

The variable names in `flights` are self explanatory, but note that `flights` does not print like a regular data frame. This is because it is a *tibble*, which is designed for data with a lot of rows and/or columns, i.e., big data. The `print` function combines features of `head` and `str` in providing information about the tibble. Alternatively, we can use `str()` to give information about tibbles or data frames.

```
str(flights)
```

```
## tibble [336,776 x 19] (S3: tbl_df/tbl/data.frame)
## $ year      : int [1:336776] 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 ...
## $ month     : int [1:336776] 1 1 1 1 1 1 1 1 1 1 1 1 ...
## $ day       : int [1:336776] 1 1 1 1 1 1 1 1 1 1 1 1 ...
## $ dep_time  : int [1:336776] 517 533 542 544 554 554 555 557 557 558 ...
## $ sched_dep_time: int [1:336776] 515 529 540 545 600 558 600 600 600 600 ...
## $ dep_delay : num [1:336776] 2 4 2 -1 -6 -4 -5 -3 -3 -2 ...
## $ arr_time  : int [1:336776] 830 850 923 1004 812 740 913 709 838 753 ...
## $ sched_arr_time: int [1:336776] 819 830 850 1022 837 728 854 723 846 745 ...
## $ arr_delay : num [1:336776] 11 20 33 -18 -25 12 19 -14 -8 8 ...
## $ carrier   : chr [1:336776] "UA" "UA" "AA" "B6" ...
## $ flight    : int [1:336776] 1545 1714 1141 725 461 1696 507 5708 79 301 ...
## $ tailnum   : chr [1:336776] "N14228" "N24211" "N619AA" "N804JB" ...
## $ origin    : chr [1:336776] "EWR" "LGA" "JFK" "JFK" ...
## $ dest      : chr [1:336776] "IAH" "IAH" "MIA" "BQN" ...
## $ air_time  : num [1:336776] 227 227 160 183 116 150 158 53 140 138 ...
## $ distance  : num [1:336776] 1400 1416 1089 1576 762 ...
## $ hour      : num [1:336776] 5 5 5 5 6 5 6 6 6 6 ...
## $ minute    : num [1:336776] 15 29 40 45 0 58 0 0 0 0 ...
## $ time_hour : POSIXct[1:336776], format: "2013-01-01 05:00:00" "2013-01-01 05:00:00" ...
```

The `time_hour` variable in the `flights` data is encoded using the `POSIXct` format, which is identical to the format used for `time_hour` in the `weather` data of Section 3.1.4. The `time_hour` variable can be computed using the `make_datetime` function from the `ludridate` package with `year`, `month`, `day`, and `hour`

as arguments. The flights table could be joined to the weather table using `time_hour` and `origin` as keys, which at least in principle allows us to model `dep_delay` in terms of the weather variables.

We could also define a `time_min` variable as follows:

```
make_datetime(year = flights$year, month = flights$month, day = flights$day,
              hour = flights$hour, min = flights$minute)[1:5]
```

```
## [1] "2013-01-01 05:15:00 UTC" "2013-01-01 05:29:00 UTC"
## [3] "2013-01-01 05:40:00 UTC" "2013-01-01 05:45:00 UTC"
## [5] "2013-01-01 06:00:00 UTC"
```

This would allow us to model `dep_delay` at a finer level of granularity, but unfortunately the weather variables are only measured to the nearest hour.

3.1.1 Single Table Verbs

`dplyr` provides a suite of verbs for data manipulation:

- `filter`: select rows in a data frame;
- `arrange`: reorder rows in a data frame;
- `select`: select columns in a data frame;
- `distinct`: find unique values in a table;
- `mutate`: add new columns to a data frame;
- `summarise`: collapses a data frame to a single row;
- `sample_n`: take a random sample of rows.

Filter and Slice `filter()` allows the selection of rows using Boolean operations, e.g., `&` or `|`.

```
# The following is equivalent to filter(flights, month == 1, day == 1).
filter(flights, month == 1 & day == 1)
```

```
## # A tibble: 842 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
## 1  2013     1     1     517             515           2     830             819
## 2  2013     1     1     533             529           4     850             830
## 3  2013     1     1     542             540           2     923             850
## 4  2013     1     1     544             545          -1    1004            1022
## 5  2013     1     1     554             600          -6     812             837
## 6  2013     1     1     554             558          -4     740             728
## 7  2013     1     1     555             600          -5     913             854
## 8  2013     1     1     557             600          -3     709             723
## 9  2013     1     1     557             600          -3     838             846
## 10 2013     1     1     558             600          -2     753             745
## # ... with 832 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dtm>
```

```
# In base R this would be done as:
# flights[flights$month == 1 & flights$day == 1, ]
```

Using the `|` operator is also easy.

```
filter(flights, month == 1 | month == 2)
```

```
## # A tibble: 51,955 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>       <dbl>    <int>         <int>
## 1  2013     1     1     517           515         2      830           819
## 2  2013     1     1     533           529         4      850           830
## 3  2013     1     1     542           540         2      923           850
## 4  2013     1     1     544           545        -1     1004          1022
## 5  2013     1     1     554           600        -6      812           837
## 6  2013     1     1     554           558        -4      740           728
## 7  2013     1     1     555           600        -5      913           854
## 8  2013     1     1     557           600        -3      709           723
## 9  2013     1     1     557           600        -3      838           846
## 10 2013     1     1     558           600        -2      753           745
## # ... with 51,945 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dtm>
```

Rows can also be selected by position using slice:

```
slice(flights, 1:3)
```

```
## # A tibble: 3 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>       <dbl>    <int>         <int>
## 1  2013     1     1     517           515         2      830           819
## 2  2013     1     1     533           529         4      850           830
## 3  2013     1     1     542           540         2      923           850
## # ... with 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dtm>
```

Arrange `arrange()` orders a data frame by a set of column names (or more complicated expressions). If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

```
arrange(flights, dep_delay)
```

```
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>       <dbl>    <int>         <int>
## 1  2013    12     7    2040           2123        -43      40           2352
## 2  2013     2     3    2022           2055        -33     2240          2338
## 3  2013    11    10    1408           1440        -32     1549          1559
## 4  2013     1    11    1900           1930        -30     2233          2243
## 5  2013     1    29    1703           1730        -27     1947          1957
## 6  2013     8     9     729           755        -26     1002           955
## 7  2013    10    23    1907           1932        -25     2143          2143
## 8  2013     3    30    2030           2055        -25     2213          2250
## 9  2013     3     2    1431           1455        -24     1601          1631
## 10 2013     5     5     934           958        -24     1225          1309
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
```

```
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dtm>
# Or with 'arr_delay' descending:
arrange(flights, desc(dep_delay))
```

```
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>         <dbl>     <int>         <int>
## 1  2013     1     9     641             900         1301     1242         1530
## 2  2013     6    15    1432            1935         1137     1607         2120
## 3  2013     1    10    1121            1635         1126     1239         1810
## 4  2013     9    20    1139            1845         1014     1457         2210
## 5  2013     7    22     845            1600         1005     1044         1815
## 6  2013     4    10    1100            1900          960     1342         2211
## 7  2013     3    17    2321             810          911      135         1020
## 8  2013     6    27     959            1900          899     1236         2226
## 9  2013     7    22    2257             759          898      121         1026
## 10 2013    12     5     756            1700          896     1058         2020
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dtm>
```

Select and Rename `select()` allows you to focus on the variables of interest:

```
# Select columns by name
select(flights, year, month, day)
```

```
## # A tibble: 336,776 x 3
##   year month   day
##   <int> <int> <int>
## 1  2013     1     1
## 2  2013     1     1
## 3  2013     1     1
## 4  2013     1     1
## 5  2013     1     1
## 6  2013     1     1
## 7  2013     1     1
## 8  2013     1     1
## 9  2013     1     1
## 10 2013     1     1
## # ... with 336,766 more rows
```

```
# Select all columns between year and day (inclusive)
select(flights, year:day)
```

```
## # A tibble: 336,776 x 3
##   year month   day
##   <int> <int> <int>
## 1  2013     1     1
## 2  2013     1     1
## 3  2013     1     1
## 4  2013     1     1
## 5  2013     1     1
## 6  2013     1     1
## 7  2013     1     1
## 8  2013     1     1
```

```
## 9 2013 1 1
## 10 2013 1 1
## # ... with 336,766 more rows
```

```
# Select all columns except those from year to day (inclusive)
select(flights, -(year:day))
```

```
## # A tibble: 336,776 x 16
##   dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay carrier
##   <int>      <int>      <dbl>   <int>      <int>      <dbl> <chr>
## 1     517         515         2     830         819        11 UA
## 2     533         529         4     850         830        20 UA
## 3     542         540         2     923         850        33 AA
## 4     544         545        -1    1004        1022       -18 B6
## 5     554         600        -6     812         837       -25 DL
## 6     554         558        -4     740         728        12 UA
## 7     555         600        -5     913         854        19 B6
## 8     557         600        -3     709         723       -14 EV
## 9     557         600        -3     838         846        -8 B6
## 10    558         600        -2     753         745         8 AA
## # ... with 336,766 more rows, and 9 more variables: flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dtm>
```

`dplyr::select()` is similar to `base::select()`, but is included in `dplyr` to have a comprehensive, consistent architecture for data manipulation.

It is possible to rename variables with `select`, but `rename` is a better choice since `select` drops any unnamed variables:

```
rename(flights, tail_num = tailnum)
```

```
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>      <int>      <dbl>   <int>      <int>
## 1  2013     1     1     517         515         2     830         819
## 2  2013     1     1     533         529         4     850         830
## 3  2013     1     1     542         540         2     923         850
## 4  2013     1     1     544         545        -1    1004        1022
## 5  2013     1     1     554         600        -6     812         837
## 6  2013     1     1     554         558        -4     740         728
## 7  2013     1     1     555         600        -5     913         854
## 8  2013     1     1     557         600        -3     709         723
## 9  2013     1     1     557         600        -3     838         846
## 10 2013     1     1     558         600        -2     753         745
## # ... with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tail_num <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dtm>
```

Distinct `distinct()` finds unique values in a table:

```
distinct(flights, tailnum)
```

```
## # A tibble: 4,044 x 1
##   tailnum
##   <chr>
## 1 N14228
```

```
## 2 N24211
## 3 N619AA
## 4 N804JB
## 5 N668DN
## 6 N39463
## 7 N516JB
## 8 N829AS
## 9 N593JB
## 10 N3ALAA
## # ... with 4,034 more rows
```

```
distinct(flights, origin, dest)
```

```
## # A tibble: 224 x 2
##   origin dest
##   <chr> <chr>
## 1 EWR    IAH
## 2 LGA    IAH
## 3 JFK    MIA
## 4 JFK    BQN
## 5 LGA    ATL
## 6 EWR    ORD
## 7 EWR    FLL
## 8 LGA    IAD
## 9 JFK    MCO
## 10 LGA    ORD
## # ... with 214 more rows
```

This is similar to `base::unique()` but is faster.

Mutate and Transmute `mutate()` transforms variables, i.e., adds new columns that are functions of existing columns.

```
mutate(flights,
       gain = arr_delay - dep_delay,
       speed = distance / air_time * 60)
```

```
## # A tibble: 336,776 x 21
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
## 1  2013     1     1     517             515           2     830           819
## 2  2013     1     1     533             529           4     850           830
## 3  2013     1     1     542             540           2     923           850
## 4  2013     1     1     544             545          -1    1004          1022
## 5  2013     1     1     554             600          -6     812           837
## 6  2013     1     1     554             558          -4     740           728
## 7  2013     1     1     555             600          -5     913           854
## 8  2013     1     1     557             600          -3     709           723
## 9  2013     1     1     557             600          -3     838           846
## 10 2013     1     1     558             600          -2     753           745
## # ... with 336,766 more rows, and 13 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dtm>,
## #   gain <dbl>, speed <dbl>
```

`dplyr::mutate()` works similarly to `base::transform()`, but `transform()` does not allow you to refer to

columns that you've just created. For example, the following would not work with `transform()`, since the second argument depends on the first:

```
mutate(flights,
  gain = arr_delay - dep_delay,
  gain_per_hour = gain / (air_time / 60))
```

```
## # A tibble: 336,776 x 21
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>         <dbl>    <int>         <int>
## 1  2013     1     1     517             515           2      830           819
## 2  2013     1     1     533             529           4      850           830
## 3  2013     1     1     542             540           2      923           850
## 4  2013     1     1     544             545          -1     1004          1022
## 5  2013     1     1     554             600          -6      812           837
## 6  2013     1     1     554             558          -4      740           728
## 7  2013     1     1     555             600          -5      913           854
## 8  2013     1     1     557             600          -3      709           723
## 9  2013     1     1     557             600          -3      838           846
## 10 2013     1     1     558             600          -2      753           745
## # ... with 336,766 more rows, and 13 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dtm>,
## #   gain <dbl>, gain_per_hour <dbl>
```

Note: The new variables are not actually part of `flights` as can be seen by printing `flights`, but the new tibble can be used as part of a workflow. Alternately, a new tibble, e.g., `flights_gain` could be created by: `flights_gain <- mutate(...)`.

If you only want to keep the new variables, use `transmute()`:

```
transmute(flights,
  gain = arr_delay - dep_delay,
  gain_per_hour = gain / (air_time / 60)
)
```

```
## # A tibble: 336,776 x 2
##   gain gain_per_hour
##   <dbl>         <dbl>
## 1     9           2.38
## 2    16           4.23
## 3    31          11.6
## 4   -17          -5.57
## 5   -19          -9.83
## 6    16           6.4
## 7    24           9.11
## 8   -11          -12.5
## 9    -5           -2.14
## 10   10           4.35
## # ... with 336,766 more rows
```

Now let's add a `time_min` variables to the `flights` data using the four time variables. The modulo operator is used in which the quotient (hour) and remainder (min) are extracted from `sched_dep_time`.

```
mutate(flights,
  time_min = make_datetime(year, month, day,
    sched_dep_time %/% 100,
```



```
sched_dep_time %% 100))$time_min[1:5]
```

```
## [1] "2013-01-01 05:15:00 UTC" "2013-01-01 05:29:00 UTC"
## [3] "2013-01-01 05:40:00 UTC" "2013-01-01 05:45:00 UTC"
## [5] "2013-01-01 06:00:00 UTC"
```

Sample `sample_n()` and `sample_frac()` are used to take a random sample of rows for a fixed number and a fixed fraction, respectively.

```
sample_n(flights, 10)
```

```
## # A tibble: 10 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
## 1  2013     3     2    1753           1753           0    2054           2044
## 2  2013     7    28    1323           1328          -5    1546           1530
## 3  2013     8    27    1824           1828          -4    2016           2034
## 4  2013     4    27     832            830           2     955           1017
## 5  2013    10    18     828            840          -12     959           1000
## 6  2013     9    30     857            854           3    1141           1213
## 7  2013     2    21     832            816          16    1152           1135
## 8  2013     5    10    1410           1329          41    1515           1445
## 9  2013     8    16    1200           1200           0    1427           1430
## 10 2013    11    30    1821           1829          -8    2015           2033
## # ... with 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dtm>
```

```
sample_frac(flights, 0.01)
```

```
## # A tibble: 3,368 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##   <int> <int> <int>   <int>         <int>         <dbl>   <int>         <int>
## 1  2013     8    23     856            900          -4    1126           1153
## 2  2013     3    21    1532           1508          24    1858           1819
## 3  2013     8    30     830            835          -5    1113           1141
## 4  2013    12     3    1710           1714          -4    1830           1835
## 5  2013     2     4    1941           1845          56    2130           2030
## 6  2013    10    25     910            915          -5    1122           1154
## 7  2013     8     9    2111           1920         111    2230           2045
## 8  2013     7    24    1520           1525          -5    1705           1725
## 9  2013     9    21    1726           1137        349    1904           1336
## 10 2013     8    30     620            625          -5     844            915
## # ... with 3,358 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dtm>
```

The argument `replace = TRUE` samples with replacement, e.g., for a bootstrap sample. The `weight` argument allows you to weight the observations.

The above verbs have a common syntax.

- the first argument is a data frame (or tibble);
- subsequent arguments describe what to do to the data frame;

- the result is data frame (or tibble).

These properties allow the user to form a workflow chain or pipeline with the verbs and other compatible functions.

3.1.2 Grouped Operations

These above verbs become very powerful when you apply them to groups of observations within a dataset. In `dplyr`, this is done by the `group_by()` function. It breaks a dataset into specified groups of rows. When you then apply the verbs above on the resulting object they'll be automatically applied "by group."

We now split the complete dataset into individual planes and then summarise each plane by counting the number of flights and computing the average distance and arrival delay.

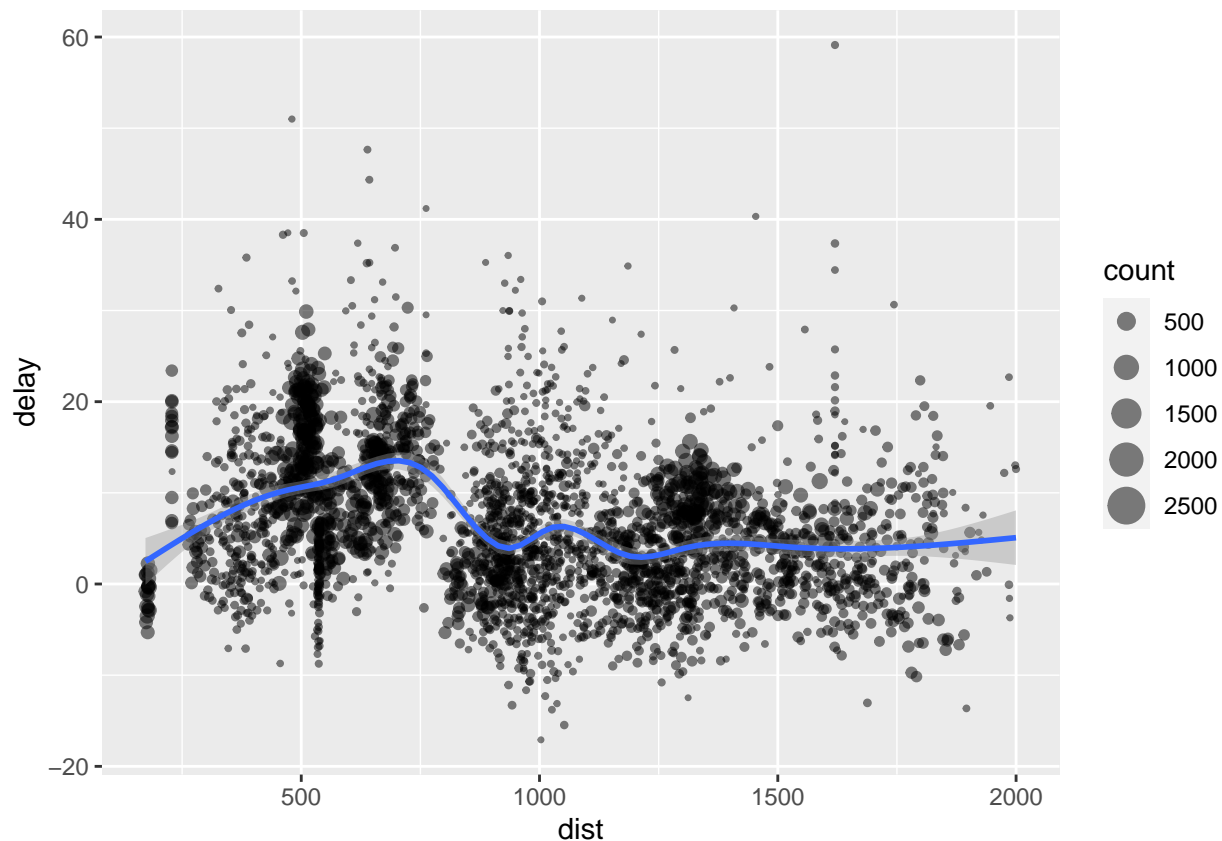
```
by_tailnum <- group_by(flights, tailnum)
delay <- summarise(by_tailnum,
  count = n(),
  dist = mean(distance, na.rm = TRUE),
  delay = mean(arr_delay, na.rm = TRUE))
delay <- filter(delay, count > 20, dist < 2000)
delay
```

```
## # A tibble: 2,962 x 4
##   tailnum count  dist  delay
##   <chr>   <int> <dbl> <dbl>
## 1 NOEGMQ    371  676.   9.98
## 2 N10156   153  758.  12.7
## 3 N102UW    48  536.   2.94
## 4 N103US    46  535.  -6.93
## 5 N104UW    47  535.   1.80
## 6 N10575   289  520.  20.7
## 7 N105UW    45  525.  -0.267
## 8 N107US    41  529.  -5.73
## 9 N108UW    60  534.  -1.25
## 10 N109UW   48  536.  -2.52
## # ... with 2,952 more rows
```

We can then see if the average delay is related to the average distance flown by a plane.

```
library(ggplot2)
ggplot(delay, aes(dist, delay)) +
  geom_point(aes(size = count), alpha = 1/2) +
  geom_smooth() +
  scale_size_area()

## 'geom_smooth()' using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
## Warning: Removed 1 rows containing non-finite values (stat_smooth).
## Warning: Removed 1 rows containing missing values (geom_point).
```



The average delay increases for short distance (with a lot of variation), but then levels out.

This course does not focus on graphics, but we will use simple graphics in various workflows. The principal graphics packages that integrate into workflows include:

- Grammar of graphics

`ggplot2` is a plotting system for R, based on the Leland Wilkinson's grammar of graphics. It takes care of many of the details that make plotting a hassle (like drawing legends) as well as providing a powerful model of graphics that makes it easy to produce complex multi-layered graphics.

- Interactive grammar of graphics

`ggvis` makes it easy to describe interactive web graphics in R. It combines:

- a grammar of graphics from `ggplot2`,
- reactive programming from `shiny`, and
- data transformation pipelines from `dplyr`.

You use `summarise()` with aggregate functions, which take a vector of values and return a single number. There are many useful examples of such functions in base R, e.g., `mean()`, `sum()`, and `sd()`.

`dplyr` adds:

- `n()`: the number of observations in the current group;
- `n_distinct(x)`: the number of unique values in `x`;
- `first(x)`, `last(x)`, and `nth(x, n)`: the first, last, and `n`th observation in `x`.

You can also use your own functions.

For example, we could use these to find the number of planes and the number of flights that go to each possible destination:

```
destinations <- group_by(flights, dest)
summarise(destinations,
  planes = n_distinct(tailnum),
  flights = n()
)
```

```
## # A tibble: 105 x 3
##   dest planes flights
##   <chr> <int> <int>
## 1 ABQ    108    254
## 2 ACK     58    265
## 3 ALB    172    439
## 4 ANC      6      8
## 5 ATL   1180   17215
## 6 AUS    993   2439
## 7 AVL    159    275
## 8 BDL    186    443
## 9 BGR     46    375
## 10 BHM     45    297
## # ... with 95 more rows
```

When you group by multiple variables, each summary peels off one level of the grouping. Thus, you can progressively roll-up a dataset:

```
daily <- group_by(flights, year, month, day)
(per_day <- summarise(daily, flights = n()))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##   year month day flights
##   <int> <int> <int> <int>
## 1 2013     1   1     842
## 2 2013     1   2     943
## 3 2013     1   3     914
## 4 2013     1   4     915
## 5 2013     1   5     720
## 6 2013     1   6     832
## 7 2013     1   7     933
## 8 2013     1   8     899
## 9 2013     1   9     902
## 10 2013     1  10     932
## # ... with 355 more rows
```

```
(per_month <- summarise(per_day, flights = sum(flights)))
```

```
## # A tibble: 12 x 3
## # Groups:   year [1]
##   year month flights
##   <int> <int> <int>
## 1 2013     1  27004
## 2 2013     2  24951
## 3 2013     3  28834
## 4 2013     4  28330
## 5 2013     5  28796
```

```
## 6 2013      6 28243
## 7 2013      7 29425
## 8 2013      8 29327
## 9 2013      9 27574
## 10 2013     10 28889
## 11 2013     11 27268
## 12 2013     12 28135
```

```
(per_year <- summarise(per_month, flights = sum(flights)))
```

```
## # A tibble: 1 x 2
##   year flights
##   <int>   <int>
## 1  2013  336776
```

3.1.3 Chaining

The dplyr API is *functional*, i.e., the function calls don't have *side-effects*. That means you must always save intermediate results, which doesn't lead to elegant code. One solution is to do it step-by-step.

```
a1 <- group_by(flights, year, month, day)
a2 <- select(a1, arr_delay, dep_delay)
```

```
## Adding missing grouping variables: 'year', 'month', 'day'
```

```
a3 <- summarise(a2,
  arr = mean(arr_delay, na.rm = TRUE),
  dep = mean(dep_delay, na.rm = TRUE))
a4 <- filter(a3, arr > 30 | dep > 30)
a4
```

```
## # A tibble: 49 x 5
## # Groups:   year, month [11]
##   year month   day   arr   dep
##   <int> <int> <int> <dbl> <dbl>
## 1  2013     1    16  34.2  24.6
## 2  2013     1    31  32.6  28.7
## 3  2013     2    11  36.3  39.1
## 4  2013     2    27  31.3  37.8
## 5  2013     3     8  85.9  83.5
## 6  2013     3    18  41.3  30.1
## 7  2013     4    10  38.4  33.0
## 8  2013     4    12  36.0  34.8
## 9  2013     4    18  36.0  34.9
## 10 2013     4    19  47.9  46.1
## # ... with 39 more rows
```

This is not a good idea for big data.

If you want to save storage, another way is to wrap the function calls inside each other.

```
filter(
  summarise(
    select(
      group_by(flights, year, month, day),
      arr_delay, dep_delay
    ),
    arr = mean(arr_delay, na.rm = TRUE),
```

```

    dep = mean(dep_delay, na.rm = TRUE)
  ),
  arr > 30 | dep > 30
)

```

Adding missing grouping variables: 'year', 'month', 'day'

```

## # A tibble: 49 x 5
## # Groups:   year, month [11]
##   year month   day   arr   dep
##   <int> <int> <int> <dbl> <dbl>
## 1  2013     1    16  34.2  24.6
## 2  2013     1    31  32.6  28.7
## 3  2013     2    11  36.3  39.1
## 4  2013     2    27  31.3  37.8
## 5  2013     3     8  85.9  83.5
## 6  2013     3    18  41.3  30.1
## 7  2013     4    10  38.4  33.0
## 8  2013     4    12  36.0  34.8
## 9  2013     4    18  36.0  34.9
## 10 2013     4    19  47.9  46.1
## # ... with 39 more rows

```

However, this is difficult to read because the order of the operations is from inside to out. Thus, the arguments are a long way away from the function. To get around this problem, `dplyr` provides the `%>%` operator. `x %>% f(y)` turns into `f(x, y)` so you can use it to rewrite multiple operations that you can read left-to-right, top-to-bottom:

```

flights %>%
  group_by(year, month, day) %>%
  select(arr_delay, dep_delay) %>%
  summarise(
    arr = mean(arr_delay, na.rm = TRUE),
    dep = mean(dep_delay, na.rm = TRUE)
  ) %>%
  filter(arr > 30 | dep > 30)

```

Adding missing grouping variables: 'year', 'month', 'day'

```

## # A tibble: 49 x 5
## # Groups:   year, month [11]
##   year month   day   arr   dep
##   <int> <int> <int> <dbl> <dbl>
## 1  2013     1    16  34.2  24.6
## 2  2013     1    31  32.6  28.7
## 3  2013     2    11  36.3  39.1
## 4  2013     2    27  31.3  37.8
## 5  2013     3     8  85.9  83.5
## 6  2013     3    18  41.3  30.1
## 7  2013     4    10  38.4  33.0
## 8  2013     4    12  36.0  34.8
## 9  2013     4    18  36.0  34.9
## 10 2013     4    19  47.9  46.1
## # ... with 39 more rows

```

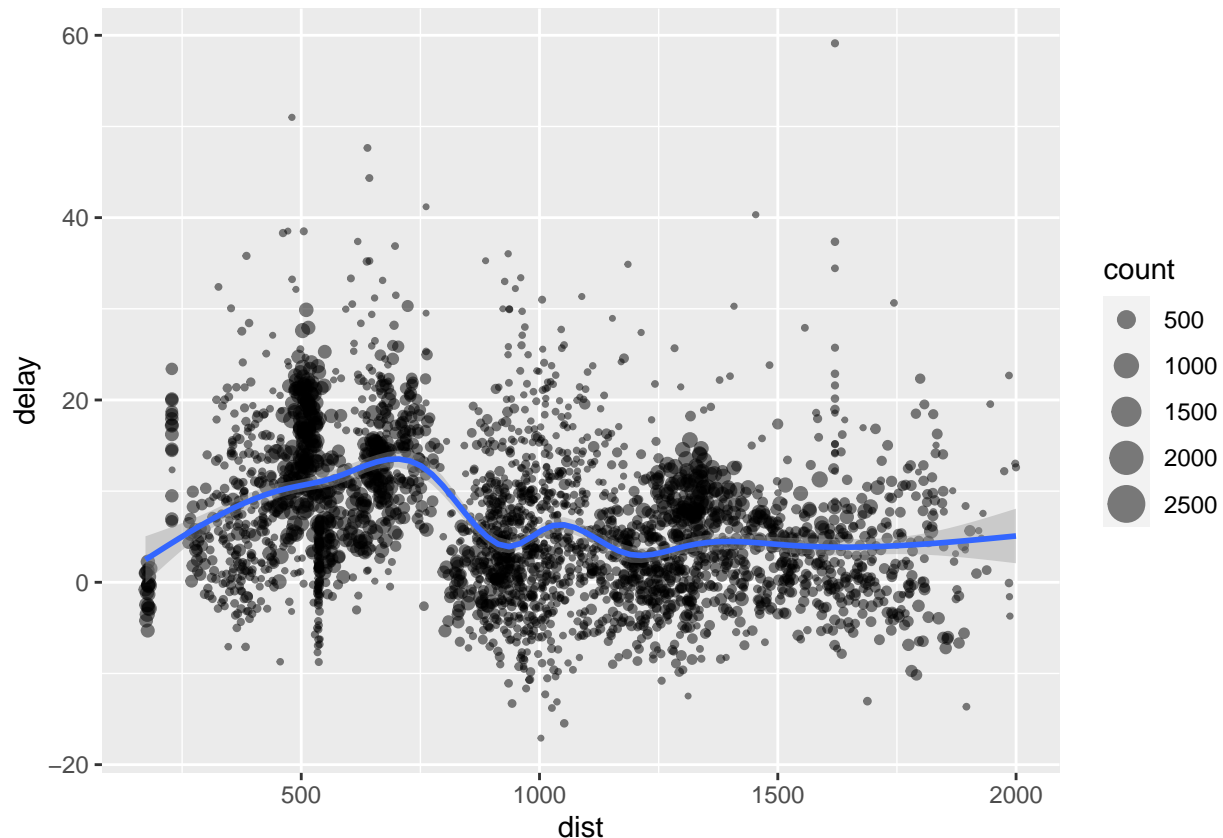
The `%>%` R operator is somewhat like UNIX pipes in which the standard output of one command becomes

the standard input of the next. Thus, we sometimes call `%>%` the R pipe operator.

However, `%>%` is very powerful since it can be used with many R functions including graphics functions in R packages such as `ggplot2` and `ggvis`.

Let's redo our grouped `tailnum` example using `%>%`:

```
group_by(flights, tailnum) %>%  
  summarise(  
    count = n(),  
    dist = mean(distance, na.rm = TRUE),  
    delay = mean(arr_delay, na.rm = TRUE)) %>%  
  filter(  
    count > 20, dist < 2000) %>%  
  ggplot(  
    aes(dist, delay)) +  
    geom_point(aes(size = count), alpha = 1/2) +  
    geom_smooth() +  
    scale_size_area()  
  
## 'geom_smooth()' using method = 'gam' and formula 'y ~ s(x, bs = "cs")'  
## Warning: Removed 1 rows containing non-finite values (stat_smooth).  
## Warning: Removed 1 rows containing missing values (geom_point).
```



What makes this work is that the first argument is a data frame and the output is a data frame. Do you see the potential of building very powerful workflows?

3.1.4 Combining Tables

It's rare that a data analysis involves only a single table of data. In practice, you'll normally have many tables that contribute to an analysis, and you need flexible tools to combine them.

The material for this section is extracted from Hadley Wickham's *dplyr Two-table Vignette*.

In *dplyr*, there are three families of verbs that work with two tables at a time:

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

This discussion assumes that you have tidy data, where the rows are observations and the columns are variables (see Section 3.3). The discussion here will be limited to mutating joins.

All two-table verbs work similarly. The first two arguments are *x* and *y*, and provide the tables to combine. The output is always a new table with the same type as *x*.

Mutating joins Mutating joins allow you to combine variables from multiple tables. For example, take the *nycflights13* data. In one table we have flight information with an abbreviation for carrier, and in another we have a mapping between abbreviations and full names. You can use a join to add the carrier names to the flight data:

```
# Drop unimportant variables so it's easier to understand the join results.
flights2 <- flights %>%
  select(year:day, hour, origin, dest, tailnum, carrier)
airlines
```

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

```
flights2 %>%
  left_join(airlines)
```

```
## Joining, 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 Inc.
## 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 Inc.
## 9 2013      1      1      6 JFK   MCO   N593JB B6    JetBlue Airways
## 10 2013      1      1      6 LGA   ORD   N3ALAA AA    American Airlines Inc.
## # ... with 336,766 more rows
```

Controlling how the tables are matched In addition to `x` and `y`, each mutating join takes an argument by that controls which variables are used to match observations in the two tables. There are several ways to specify it.

- `NULL`, the default. `dplyr` will use all variables that appear in both tables, a natural join. For example, the `flights` and `weather` tables match on their common variables: `year`, `month`, `day`, `hour` and `origin`.

```
str(weather)
```

```
## tibble [26,115 x 15] (S3: tbl_df/tbl/data.frame)
## $ origin      : chr [1:26115] "EWR" "EWR" "EWR" "EWR" ...
## $ year        : int [1:26115] 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 ...
## $ month       : int [1:26115] 1 1 1 1 1 1 1 1 1 1 ...
## $ day         : int [1:26115] 1 1 1 1 1 1 1 1 1 1 ...
## $ hour        : int [1:26115] 1 2 3 4 5 6 7 8 9 10 ...
## $ temp        : num [1:26115] 39 39 39 39.9 39 ...
## $ dewp        : num [1:26115] 26.1 27 28 28 28 ...
## $ humid       : num [1:26115] 59.4 61.6 64.4 62.2 64.4 ...
## $ wind_dir    : num [1:26115] 270 250 240 250 260 240 240 250 260 260 ...
## $ wind_speed  : num [1:26115] 10.36 8.06 11.51 12.66 12.66 ...
## $ wind_gust   : num [1:26115] NA NA NA NA NA NA NA NA NA NA ...
## $ precip      : num [1:26115] 0 0 0 0 0 0 0 0 0 0 ...
## $ pressure    : num [1:26115] 1012 1012 1012 1012 1012 ...
## $ visib       : num [1:26115] 10 10 10 10 10 10 10 10 10 10 ...
## $ time_hour   : POSIXct[1:26115], format: "2013-01-01 01:00:00" "2013-01-01 02:00:00" ...
```

```
flights2 %>%
```

```
  left_join(weather)
```

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

```
## # 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 <dtm>
```

- A character vector, `by = "x"`. Like a natural join, but uses only some of the common variables. For example, `flights` and `planes` have year columns, but they mean different things so we only want to join by `tailnum`.

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

```
## # A tibble: 336,776 x 16
##   year.x month   day hour origin dest tailnum carrier year.y type
##   <int> <int> <int> <dbl> <chr> <chr> <chr> <chr> <int> <chr>
## 1 2013     1     1     5 EWR   IAH  N14228 UA      1999 Fixe~
## 2 2013     1     1     5 LGA   IAH  N24211 UA      1998 Fixe~
## 3 2013     1     1     5 JFK   MIA  N619AA AA      1990 Fixe~
## 4 2013     1     1     5 JFK   BQN  N804JB B6      2012 Fixe~
## 5 2013     1     1     6 LGA   ATL  N668DN DL      1991 Fixe~
## 6 2013     1     1     5 EWR   ORD  N39463 UA      2012 Fixe~
## 7 2013     1     1     6 EWR   FLL  N516JB B6      2000 Fixe~
## 8 2013     1     1     6 LGA   IAD  N829AS EV      1998 Fixe~
## 9 2013     1     1     6 JFK   MCO  N593JB B6      2004 Fixe~
## 10 2013     1     1     6 LGA   ORD  N3ALAA AA      NA <NA>
## # ... with 336,766 more rows, and 6 more variables: manufacturer <chr>,
## #   model <chr>, engines <int>, seats <int>, speed <int>, engine <chr>
```

Note that the year columns in the output are disambiguated with a suffix.

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

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: 336,776 x 15
##   year month   day hour origin dest tailnum carrier name lat lon alt
##   <int> <int> <int> <dbl> <chr> <chr> <chr> <chr> <chr> <dbl> <dbl> <dbl>
## 1 2013     1     1     5 EWR   IAH  N14228 UA      Geor~ 30.0 -95.3 97
## 2 2013     1     1     5 LGA   IAH  N24211 UA      Geor~ 30.0 -95.3 97
## 3 2013     1     1     5 JFK   MIA  N619AA AA      Miam~ 25.8 -80.3 8
## 4 2013     1     1     5 JFK   BQN  N804JB B6      <NA> NA NA NA
## 5 2013     1     1     6 LGA   ATL  N668DN DL      Hart~ 33.6 -84.4 1026
## 6 2013     1     1     5 EWR   ORD  N39463 UA      Chic~ 42.0 -87.9 668
## 7 2013     1     1     6 EWR   FLL  N516JB B6      Fort~ 26.1 -80.2 9
## 8 2013     1     1     6 LGA   IAD  N829AS EV      Wash~ 38.9 -77.5 313
## 9 2013     1     1     6 JFK   MCO  N593JB B6      Orla~ 28.4 -81.3 96
## 10 2013     1     1     6 LGA   ORD  N3ALAA AA      Chic~ 42.0 -87.9 668
## # ... with 336,766 more rows, and 3 more variables: tz <dbl>, dst <chr>,
## #   tzone <chr>
```

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

```
## # A tibble: 336,776 x 15
##   year month   day hour origin dest  tailnum carrier name   lat lon alt
##   <int> <int> <int> <dbl> <chr> <chr> <chr>   <chr>   <chr> <dbl> <dbl> <dbl>
## 1  2013     1     1     5 EWR  IAH  N14228  UA     Newa~ 40.7 -74.2  18
## 2  2013     1     1     5 LGA  IAH  N24211  UA     La G~ 40.8 -73.9  22
## 3  2013     1     1     5 JFK  MIA  N619AA  AA     John~ 40.6 -73.8  13
## 4  2013     1     1     5 JFK  BQN  N804JB  B6     John~ 40.6 -73.8  13
## 5  2013     1     1     6 LGA  ATL  N668DN  DL     La G~ 40.8 -73.9  22
## 6  2013     1     1     5 EWR  ORD  N39463  UA     Newa~ 40.7 -74.2  18
## 7  2013     1     1     6 EWR  FLL  N516JB  B6     Newa~ 40.7 -74.2  18
## 8  2013     1     1     6 LGA  IAD  N829AS  EV     La G~ 40.8 -73.9  22
## 9  2013     1     1     6 JFK  MCO  N593JB  B6     John~ 40.6 -73.8  13
## 10 2013     1     1     6 LGA  ORD  N3ALAA  AA     La G~ 40.8 -73.9  22
## # ... with 336,766 more rows, and 3 more variables: tz <dbl>, dst <chr>,
## #   tzone <chr>
```

Types of join There are four types of mutating join, which differ in their behavior when a match is not found. We'll illustrate each with a simple example:

```
(df1 <- data_frame(x = c(1, 2), y = 2:1))
```

```
## Warning: 'data_frame()' is deprecated as of tibble 1.1.0.
## Please use 'tibble()' instead.
## This warning is displayed once every 8 hours.
## Call 'lifecycle::last_warnings()' to see where this warning was generated.
```

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

```
(df2 <- data_frame(x = c(1, 3), a = 10, b = "a"))
```

```
## # A tibble: 2 x 3
##       x     a b
##   <dbl> <dbl> <chr>
## 1     1    10 a
## 2     3    10 a
```

`inner_join(x, y)` only includes observations that match in both `x` and `y`.

```
df1 %>% inner_join(df2) # %>% knitr::kable()
```

```
## Joining, by = "x"
```

```
## # A tibble: 1 x 4
##       x     y     a b
##   <dbl> <int> <dbl> <chr>
## 1     1     2    10 a
```

`left_join(x, y)` includes all observations in `x`, regardless of whether they match or not. This is the most commonly used join because it ensures that you don't lose observations from your primary table.

```
df1 %>% left_join(df2)
```

```
## Joining, by = "x"
```

```
## # A tibble: 2 x 4
```

```
##      x      y      a b
##   <dbl> <int> <dbl> <chr>
## 1     1     2    10 a
## 2     2     1     NA <NA>
```

`right_join(x, y)` includes all observations in `y`. It's equivalent to `left_join(y, x)`, but the columns will be ordered differently.

```
df1 %>% right_join(df2)
```

```
## Joining, by = "x"
## # A tibble: 2 x 4
##      x      y      a b
##   <dbl> <int> <dbl> <chr>
## 1     1     2    10 a
## 2     3     NA    10 a
```

```
df2 %>% left_join(df1)
```

```
## Joining, by = "x"
## # A tibble: 2 x 4
##      x      a b      y
##   <dbl> <dbl> <chr> <int>
## 1     1    10 a      2
## 2     3    10 a     NA
```

`full_join()` includes all observations from `x` and `y`.

```
df1 %>% full_join(df2)
```

```
## Joining, by = "x"
## # A tibble: 3 x 4
##      x      y      a b
##   <dbl> <int> <dbl> <chr>
## 1     1     2    10 a
## 2     2     1     NA <NA>
## 3     3     NA    10 a
```

The left, right and full joins are collectively known as outer joins. When a row doesn't match in an outer join, the new variables are filled in with missing values.

Each two-table verb has a straightforward SQL equivalent. The correspondences between R and SQL are:

- `inner_join()`: `SELECT * FROM x JOIN y ON x.a = y.a`
- `left_join()`: `SELECT * FROM x LEFT JOIN y ON x.a = y.a`
- `right_join()`: `SELECT * FROM x RIGHT JOIN y ON x.a = y.a`
- `full_join()`: `SELECT * FROM x FULL JOIN y ON x.a = y.a`

`x` and `y` don't have to be tables in the same database. If you specify `copy = TRUE`, `dplyr` will copy the `y` table into the same location as the `x` variable. This is useful if you've downloaded a summarized dataset and determined a subset for which you now want the full data.

You should review the coercion rules, e.g., factors are preserved only if the levels match exactly and if their levels are different the factors are coerced to character.

At this time, `dplyr` does not provide any functions for working with three or more tables.

See the complete set of vignettes on the `dplyr` repo for other examples.