Data-based Statistical Decision Model

Lecture 4 (Part V) - Data Wrangling

Sungkyu Jung

Data manipulation with `dplyr` (one table)

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` (https://github.com/hadley/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 by viewed by typing the following on the command line: `vignette("dplyr", package = "dplyr")` or by opening the `dplyr` file in the vignettes directory of the `dplyr` repo. The material for this section is extracted from Hadley Wickham's Introduction to dplyr Vignette (https://github.com/hadley/dplyr/blob/master/vignettes/dplyr.Rmd), *R for data science* (http://r4ds.had.co.nz/transform.html), and MDSR.

`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.

We will use `ggplot2::presidential` data frame.

```
library(dplyr)
library(lubridate)
library(ggplot2)
presidential
```

```
## # A tibble: 11 x 4
##    name       start      end        party
##    <chr>      <date>     <date>     <chr>
##  1 Eisenhower 1953-01-20 1961-01-20 Republican
##  2 Kennedy    1961-01-20 1963-11-22 Democratic
##  3 Johnson    1963-11-22 1969-01-20 Democratic
##  4 Nixon      1969-01-20 1974-08-09 Republican
##  5 Ford       1974-08-09 1977-01-20 Republican
##  6 Carter     1977-01-20 1981-01-20 Democratic
##  7 Reagan     1981-01-20 1989-01-20 Republican
##  8 Bush       1989-01-20 1993-01-20 Republican
##  9 Clinton    1993-01-20 2001-01-20 Democratic
## 10 Bush       2001-01-20 2009-01-20 Republican
## 11 Obama      2009-01-20 2017-01-20 Democratic
```

The variable names in `presidential` are self explanatory, but note that `presidential` 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`. `str` gives the inheritance path along with a summary of the data frame. For brevity we will use `class()` to give the inheritance path:

```
class(presidential)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

See how traditional data frame prints out.

```
MASS::Boston
as_tibble(MASS::Boston)
```

Single Table Verbs

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

- `filter()` : select rows (observations) in a data frame;
- `arrange()` : reorder rows in a data frame;
- `select()` : select columns (variables) in a data frame;
- `mutate()` : add new columns to a data frame;
- `summarise()` : collapses a data frame to a single row;

See texbook figures 4.1–4.5 for a graphical illustration of these operations.

These can all be used in conjunction with `group_by()` which changes the scope of each function from operating on the entire dataset to operating on it group-by-group. These six functions provide the verbs for a language of data manipulation.

All verbs work similarly:

1. The first argument is a data frame.

2. The subsequent arguments describe what to do with the data frame, using the variable names (without quotes).

3. The result is a new data frame.

Together these properties make it easy to chain together multiple simple steps to achieve a complex result.

A bit of side story

Wickham's approach is inspired by his desire to blur the boundaries between R and the ubiquitous relational database querying syntax SQL (https://www.datacamp.com/courses/intro-to-sql-for-data-science). In fact, the five verbs, when combined, exhibit a close similarity to SQL query statements (at least for data analysis purpose). Thus, mastering `dplyr` data wrangling verbs have become a gateway to analyzing big data, through relational database management system and beyond. The real power of `dplyr` is that it abstracts the data source, i.e., whether it is a data frame, a database, or Spark (http://spark.apache.org/).

In fact, the statistical package "SAS" have always had a powerful "data step" that does about the same thing, since 1970s.

`dplyr` also includes the powerful workflow operator called "pipes", found in e.g. Unix shell script. We will see the pipe in the first example below.

First example

select variables and filter rows

```
presidential
```

To retrieve only the names and party affiliations of these presidents, we would use `select()`. The first argument to the `select()` function is the data frame, followed by an arbitrarily long list of column names, separated by commas.

```
select(presidential, name, party)
```

To retrive only the Repulbican presidents, we use `filter()`. The first argument to `filter()` is a data frame, and subsequent arguments are *logical conditions* that are evaluated on any involved columns.

```
filter(presidential, party == "Republican")
```

Naturally, combining the `filter()` and `select()` commands enables one to drill down to very specific pieces of information. For example, we can find which Democratic presidents served since Watergate.

```
select(filter(presidential, start > 1973 & party == "Democratic"), name)
```

In the syntax demonstrated above, the `filter()` operation is nested inside the `select()` operation. Each of the five verbs takes and returns a data frame, which makes this type of nesting possible. These long expressions become very difficult to read. Instead, we recommend the use of the `%>%` (pipe) operator.

```
presidential %>%
  filter(start > 1973 & party == "Democratic") %>%
  select(name)
```

Notice how the expression `dataframe %>% filter(condition)` is equivalent to `filter(dataframe, condition)`.

The above *pipeline* reads

> Take `presidential` data frame, then filter the Democrate presidents whose start year is greater than 1973. Then select the variable `name`.

mutate variables to create new ones

Frequently, in the process of conducting our analysis, we will create, re-define, and rename some of our variables. The functions `mutate()` and `rename()` provide these capabilities.

While we have the raw data on when each of these presidents took and relinquished office, we don't actually have a numeric variable giving the length of each president's term.

```
mypresidents <- presidential %>%
  mutate(term_length = end - start)
head(mypresidents,2)
```

```
## # A tibble: 2 x 5
##   name       start      end        party      term_length
##   <chr>      <date>     <date>     <chr>      <time>
## 1 Eisenhower 1953-01-20 1961-01-20 Republican 2922 days
## 2 Kennedy    1961-01-20 1963-11-22 Democratic 1036 days
```

```
# textbook should have used mutate(term.length = interval(start, end) / dyears(1))
```

In this situation, it is generally considered good style to create a new object rather than clobbering the one that comes from an external source. To preserve the existing presidential data frame, we save the result of `mutate()` as a new object called mypresidents.

arrange rows

The function `sort()` will sort a vector, but not a data frame. The function that will sort a data frame is called `arrange()`.

To sort our presidential data frame by the length of each president's term, we specify that we want the column `term_length` in descending order.

```
mypresidents %>% arrange(desc(term_length))
```

A number of presidents completed either one or two full terms, and thus have the exact same term length (4 or 8 years, respectively). To break these ties, we can further sort by `start`.

```
mypresidents %>% arrange(desc(term_length), start)
```

summarize entire data set or for each group

Our last of the five verbs for single-table analysis is `summarize()`, which is nearly always used in conjunction with `group_by()`. The previous four verbs provided us with means to manipulate a data frame in powerful and flexible ways. But the extent of the analysis we can perform with these four verbs alone is limited. On the other hand, `summarize()` with `group_by()` enables us to make comparisons.

When used alone, `summarize()` collapses a data frame into a single row. We have to specify *how* we want to reduce an entire column of data into a single value.

```
mypresidents %>%
  summarize(
    N = n(),
    first_year = min(year(start)),
    last_year = max(year(end)),
    num_dems = sum(party == "Democratic"),
    years = sum(term_length) / 365.25,
    avg_term_length = mean(term_length)
    )
```

In this example, the function `n()` simply counts the number of rows. This is almost always useful information. The next variable determines the first year that one of these presidents assumed office. This is the smallest year in the `start` column. The variable `num_dems` simply counts the number of rows in which the value of the `party` variable was `"Democratic"`.

This begs the question of whether Democratic or Republican presidents served a longer average term during this time period. To figure this out, we can just execute `summarize()` again, but this time, instead of the first argument being the data frame mypresidents, we will specify that the rows of the `mypresidents` data frame should be grouped by the values of the `party` variable. In this manner, the same computations as above will be carried out for each party separately.

```
mypresidents %>%
  group_by(party) %>%
  summarize(
    N = n(),
    avg_term_length = mean(term_length),
    std_term_length = sd(term_length)
    )
# Compare the intermediate data.frame group_by(mypresidents,party) with mypresidents
```

The pipe

The pipe, %>%, comes from the `magrittr` package by Stefan Milton Bache. Packages in the `tidyverse` load %>% for you automatically.

Keyboard shortcut to type `%>%` is

- Cmd + Shift + M (Mac)
- Ctrl + Shift + M (Windows)

Supplement

Comparisons for `filter()`

The first argument of the function `filter()` is the data set (usually supplied through pipes).

The second argument of `filter()` is a logical vector: i.e. a vector consisting of `TRUE` and `FALSE`. Only rows where the conditon evalutes to `TRUE` are kept.

The logical vector is created by comparing one or more variables.

- Basic logical operators are `>`, `>=`, `<`, `<=`, `!=` (not equal), and `==` (equal).
- For set comparison, use `x %in% Y`, which is true is `x` is an element of the set `Y`.
- When you combine two or more comparions, use Boolean operators: `&` (and), `|` (or), `!` (not),

Suppose that `x` is a variable with four observations. What is the resulting logical vector?

```
x <- c(2,1,3,0)
x == 0
!(x == 0)
x == 0 | x == 1
x %in% c(0,1)
```

Handling missing values

One important feature of R that can make comparison tricky are missing values, or `NA`s ("not availables").

`NA` represents an unknown value so missing values are "contagious": almost any operation involving an unknown value will also be unknown.

All of the following operations return `NA`

```
x <- NA
x > 5
x + 10
x == NA
x == x
```

To check whether elements of `x` is `NA` , use `is.na(x)` .

For example, to filter *out* all observations with missing values:

```
<DATA_FRAME> %>% filter(!is.na(<VARIABLE>)
```

Select many variables

The `presidential` data set has only four variables, so selecting variables makes a little sense. To select variables for data sets with a large number of variables, there are a few handy options.

To demonstrate, load `nycflights13::flights` data set. There are 19 variables (not terribly large).

```
library(nycflights13)
flights
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

- Selecting a few varibles is easy: `select(flights, year, month, day)`
- To select all variables from variable `year` to variable `arr_time` : `select(flights, year:arr_time)` (so you don't have to type all variable names)
- To select all variables except `year, month, day` : `select(flights, -(year:day))`
- There are a number of helper functions you can use within `select()` :
  - `starts_with("abc")` : matches names that begin with “abc”.
  - `ends_with("xyz")` : matches names that end with “xyz”.
  - `contains("ijk")` : matches names that contain “ijk”.
  - `num_range("x", 1:3)` matches x1, x2 and x3.

For example:

```
flights %>% select(ends_with("time"))
flights %>% select(contains("time"))
```

Recap

- a grammar of data manipulation
    - tibble: for better management of small data sets
    - five verbs: filter, arrange, select, mutate and summarize
    - pipe `%>%` : “then”

A grammar of data manipulation: two tables

Relational data manipulation using `dplyr`

In the previous lectures, we illustrated how the five verbs can be chained to perform operations on a single table. This single table is reminiscent of a single well-organized spreadsheet. But in the same way that a workbook can contain multiple spreadsheets, we will often work with multiple tables.

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. The most common place to find relational data is in a *relational database management system* (or RDBMS), a term that encompasses almost all modern databases.

“Big Data” often involves storing really big pieces of information, fast processing of data and computation-intensive statistical learning. It requires large storage, large memory and parallel computing. In almost all instances, it involves a database, because:

- you have so much data that it does not fit in memory and you have to use a database.

The real power of `dplyr` is that it abstracts the data source, i.e., whether it is a data frame, a database, or a Spark database (a “Lightning-fast cluster computing” platform) or multidimensional arrays.

1. Databases: Currently `dplyr` supports the three most popular open source databases (`sqlite` , `mysql` and `postgresql`), and Google’s `bigquery` .

2. Spark: The `sparklyr` package is the basis for data manipulation and machine learning based on a data frame workflow. This approach has limitations, e.g., with graph algorithms, but it covers most use cases. The `rsparkling` package with its support for `h2o` delves even deeper into machine learning, e.g., deep learning. An alternative approach, officially supported by Spark, is the `SparkR` package.

3. Data cubes: `tbl_cube()` provides an experimental interface to multidimensional arrays or data cubes. Potentially this could be used for deep learning algorithms, e.g., see TensorFlow (https://www.tensorflow.org).

Working with two tables

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. We will primarily discuss mutating joins, which are used most often.

Introduction with `nycflights13` data

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

The package contains Airline on-time data for all flights departing NYC in 2013 (in `flights`). Also includes useful 'metadata' on `airlines`, `airports`, `weather`, and `planes`.

Take a look at (a part of) `flights` data.

```
flights %>%
  filter(month == 1 & day == 1, abs(dep_delay) > 30) %>%
  select(dep_time,arr_time,carrier:dest)
```

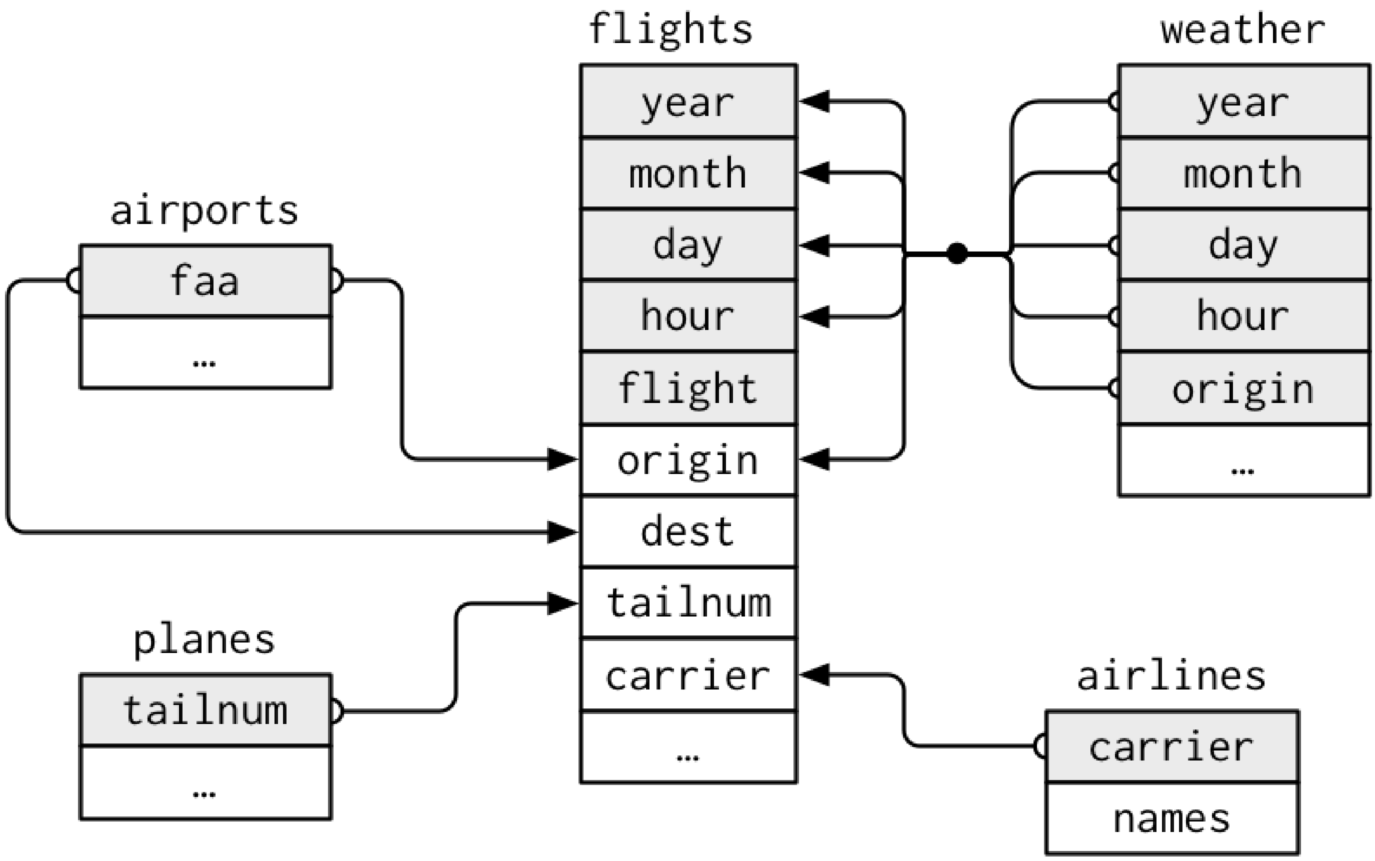
It is difficult to read the tabel because it includes lots of "codes". To decipher, we need a codebook, or metadata:

- `airlines` lets you look up the full carrier name from its abbreviated code
- `airports` gives information about each airport, identified by the `faa` airport code
- `planes` gives information about each plane, identified by its `tailnum`
- `weather` gives the weather at each NYC airport for each hour

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

Looking up the `airlines` codebook, we find that carrier `AA` stands for American Airlines Inc. This is possible because a unique identifier, the variable `carrier` appearsin both data tables. Unique identifiers are called **keys**.

One way to show the relationships between the different tables is with a drawing:



Different pairs of tables have different keys. For `nycflights13` :

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

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:

```
library("nycflights13")
# Drop unimportant variables so it's easier to understand the join results.
flights2 <- flights %>% select(year:day, hour, origin, dest,
                               tailnum, carrier)
airlines

flights2 %>%
  left_join(airlines)
```

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 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.

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

- 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")
```

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

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

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

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

`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)
```

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

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

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

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

Observations

While mutating joins are primarily used to add new variables, they can also generate new observations. If a match is not unique, a join will add all possible combinations (the Cartesian product) of the matching observations:

```
df1 <- data_frame(x = c(1, 1, 2), y = 1:3)
df2 <- data_frame(x = c(1, 1, 2), z = c("a", "b", "a"))

df1 %>% left_join(df2)
```

Filtering joins

Filtering joins match observations in the same way as mutating joins, but affect the observations, not the variables. There are two 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.

These are most useful for diagnosing join mismatches. For example, there are many flights in the `nycflights13` dataset that don't have a matching tail number in the planes table:

```
library("nycflights13")
flights %>%
  anti_join(planes, by = "tailnum") %>%
  count(tailnum, sort = TRUE)
```

If you're worried about what observations your joins will match, start with a `semi_join()` or `anti_join()`. `semi_join()` and `anti_join()` never duplicate; they only remove observations.

```
df1 <- data_frame(x = c(1, 1, 3, 4), y = 1:4)
df2 <- data_frame(x = c(1, 1, 2), z = c("a", "b", "a"))

# Four rows to start with:
df1 %>% nrow()
# And we get four rows after the join
df1 %>% inner_join(df2, by = "x") %>% nrow()
df1 %>% inner_join(df2, by = "x")
# But only two rows actually match
df1 %>% semi_join(df2, by = "x") %>% nrow()
df1 %>% semi_join(df2, by = "x")
```

Set operations

The final type of two-table verb is set operations. These expect the x and y inputs to have the same variables, and treat the observations like sets:

- `intersect(x, y)`: return only observations in both `x` and `y`
- `union(x, y)`: return unique observations in `x` and `y`
- `setdiff(x, y)`: return observations in `x`, but not in `y`.

Given this simple data:

```
df1 <- data_frame(x = 1:2, y = c(1L, 1L))
df2 <- data_frame(x = 1:2, y = 1:2)
```

The four possibilities are:

```
intersect(df1, df2)
# Note that we get 3 rows, not 4
union(df1, df2)
setdiff(df1, df2)
setdiff(df2, df1)
```

Databases

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`
- `semi_join()` : `SELECT * FROM x WHERE EXISTS (SELECT 1 FROM y WHERE x.a = y.a)`
- `anti_join()` : `SELECT * FROM x WHERE NOT EXISTS (SELECT 1 FROM y WHERE x.a = y.a)`
- `intersect(x, y)` : `SELECT * FROM x INTERSECT SELECT * FROM y`
- `union(x, y)` : `SELECT * FROM x UNION SELECT * FROM y`
- `setdiff(x, y)` : `SELECT * FROM x EXCEPT SELECT * FROM y`

`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.