CHAPTER 5: RELATIONAL DATA WITH DPLYR

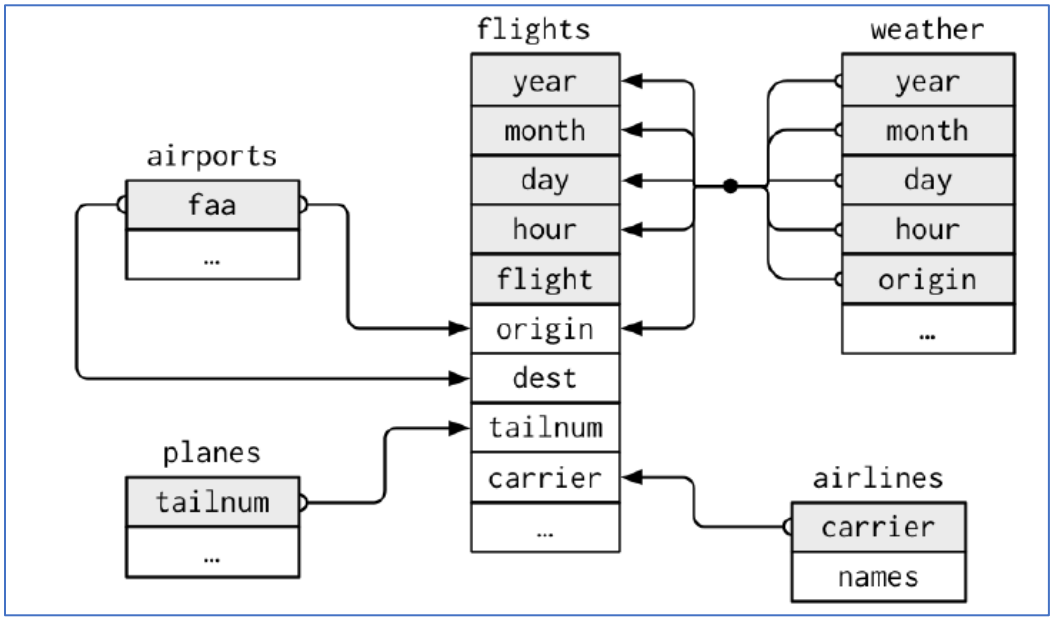
https://jrnold.github.io/r4ds-exercise-solutions/relational-data.html#filtering-joins

It's rare that a data analysis involves only a single table of data. Typically you have many tables of data, and you must combine them to answer the questions that you're interested in. Collectively, multiple tables of data are called relational data because it is the relations, not just the individual datasets, that are important.

There are three families of verbs designed to work with relational data:

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

Relationship b/w Diff datasets (Look for github entry on this)



KEYS

There are two types of keys

- A primary key uniquely identifies an observation in its own table. For example, planes$tailnum is a primary key because it uniquely identifies each plane in the planes table. In a table, primary key would be unique (only one unique key for each row)

- A foreign key uniquely identifies an observation in another table. For example, flights$tailnum is a foreign key because it appears in the flights table where it matches each flight to a unique plane.

Once you have identified a primary key, it's a good idea ensure there are duplicat values in them

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

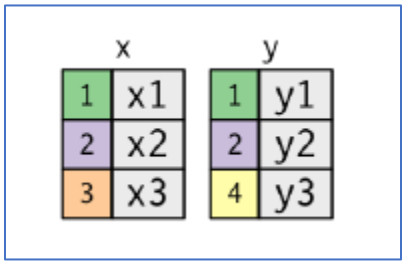
JOINS

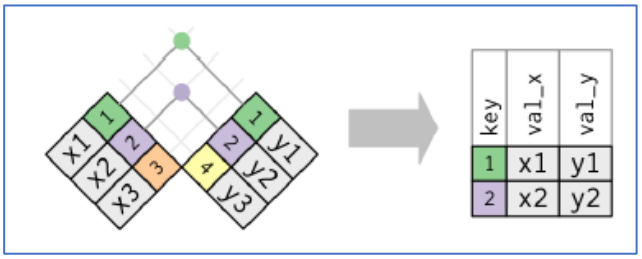
A join is a way of connecting each row in x to zero, one, or more rows in y. The following diagram shows each potential match as an intersection of a pair of lines

- Inner Joins
- Outer Joins

Inner Join:- An inner join matches pairs of observations whenever their keys are equal The output of an inner join is a new data frame that contains the key, the x values, and the y values. We use by to tell dplyr which variable is the key:

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





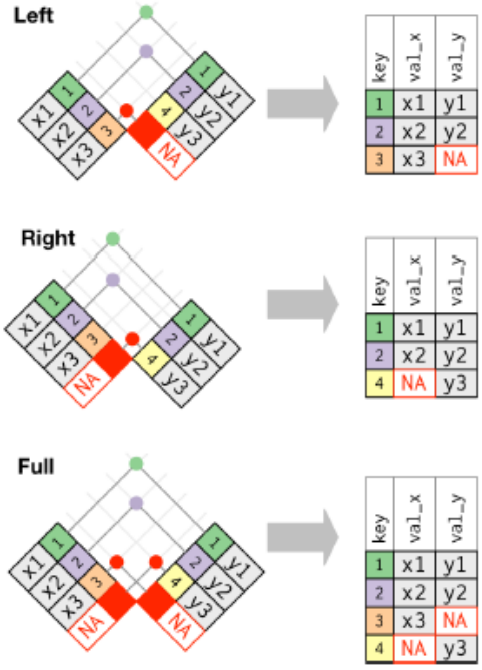
The most important property of an inner join is that unmatched rows are not included in the result. Here in the result X3 & Y3 arent included in the result.

Outer Join:- An inner join keeps observations that appear in both tables. An outer join keeps observations that appear in at least one of the tables.

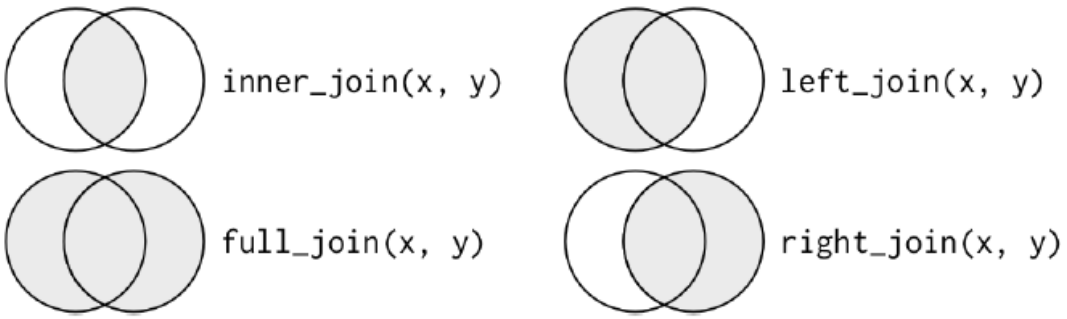
There are three types of outer joins:

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

Graphically, that looks like:

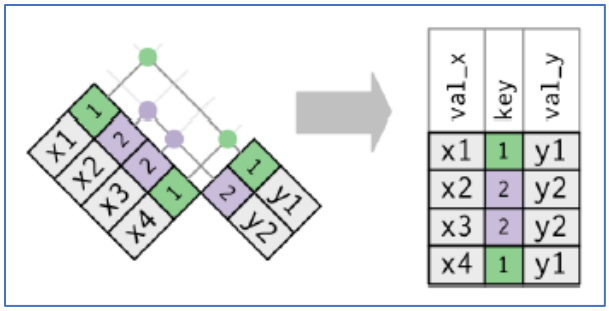


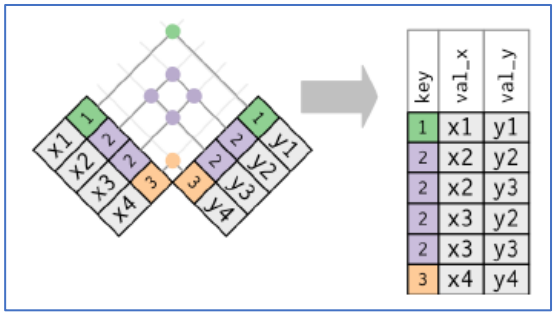
Another way to depict the different types of joins is with a Venn diagram:



Duplicate Keys:- So far all the diagrams have assumed that the keys are unique. But that's not always the case. This section explains what happens when the keys are not unique. There are two possibilities:

- One table has duplicate keys. This is useful when you want to add in additional information as there is typically a one-to-many relationship. 1st figure
- Both tables have duplicate keys. This is usually an error because in neither table do the keys uniquely identify an observation. When you join duplicated keys, you get all possible combinations, the Cartesian product. 2nd figure





Other Implementations

base::merge() can perform all four types of mutating join:

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

SQL is the inspiration for dplyr's conventions, so the translation is straightforward:

dplyr	SQL
`inner_join(x, y, by = "z")`	`SELECT * FROM x INNER JOIN y USING (z)`
`left_join(x, y, by = "z")`	`SELECT * FROM x LEFT OUTER JOIN y USING (z)`
`right_join(x, y, by = "z")`	`SELECT * FROM x RIGHT OUTER JOIN y USING (z)`
`full_join(x, y, by = "z")`	`SELECT * FROM x FULL OUTER JOIN y USING (z)`

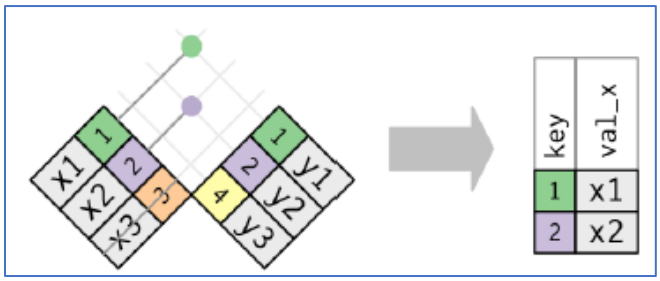
FILTERING JOINS

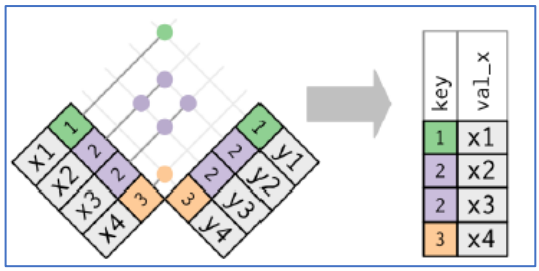
Filtering joins match observations in the same way as mutating joins, but affect the observations, not the variables. There are two types:

- 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.

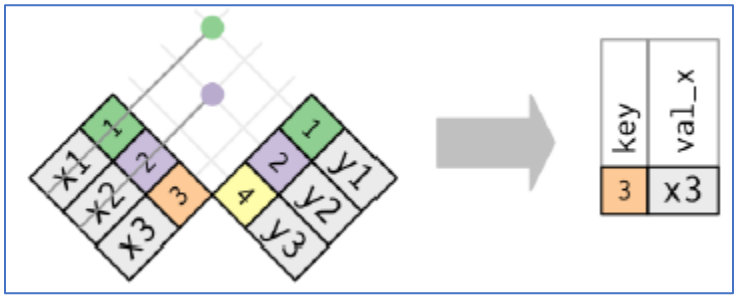
Semi-joins are useful for matching filtered summary tables back to the original rows (1st figure).

Only the existence of a match is important; it doesn't matter which observation is matched. This means that filtering joins never duplicate rows like mutating joins do (2nd figure).





The inverse of a semi-join is an anti-join. An anti-join keeps the rows that don't have a match. Anti-joins are useful for diagnosing join mismatches.



Set Operations

The final type of two-table verb are the set operations. Generally, I use these the least frequently, but they are occasionally useful when you want to break a single complex filter into simpler pieces.

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.