First: `R` and `dplyr`

Why data manipulation?

- Supervised machine learning uses structured data in a very regular and explicit form called "denormalized":
  - Every row is an event or observation.
  - Each column is homogeneous facts or variables.
  - Every fact or variable is already landed in a column.

} Schematized

} Denormalized

- We need good tools to get from wild recorded forms or efficient *normalized forms* into the above form.

R and dplyr

"No matter how complex and polished the individual operations are, it is often the quality of the glue that most directly determines the power of the system."

— Hal Abelson

dplyr

A grammar of data manipulation

`select`	`left_join`	`bind_cols`
`filter`	`right_join`	`bind_rows`
`arrange`	`inner_join`	`union`
`mutate`	`full_join`	`intersect`
`summarise`	`semi_join`	`setdiff`
`group_by`	`anti_join`	`` `%>%` ``

`dplyr` formula components

Operators	`+, -, *, /, %%, ^`
Math functions	`abs, acos, cosh, sin, asinh, atan, atan2, atanh, ceiling, cos, cosh, cot, coth, exp, floor, log, log10, round, sign, sin, sinh, sqrt, tan, tanh`
Comparisons	`<, <=, !=, >=, >, ==, %in%`
Booleans	`&, &&, |, ||, !, xor`
Aggregations	`mean, n(), rank, rank_min, sum, min, max, sd, var`

example

```
> d <- data.frame(x= 1:4)
> d$y <- 2*d$x
> print(d)

  x y
1 1 2
2 2 4
3 3 6
4 4 8
```

```
> library("dplyr")
> d <- data_frame(x= 1:4)
> d <- mutate(d, y = 2*x)
> print(d)

# A tibble: 4 × 2
      x     y
  <int> <dbl>
1     1     2
2     2     4
3     3     6
4     4     8
```

Why `dplyr`?

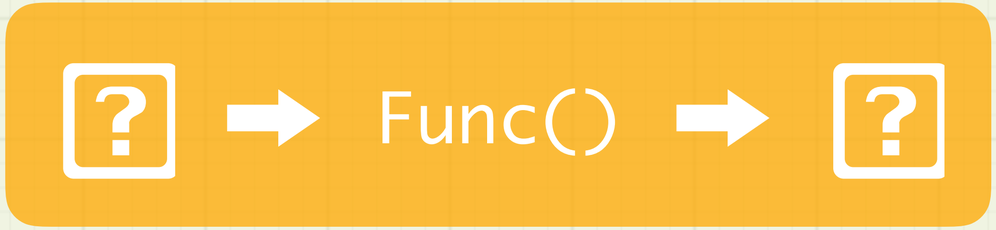
- `dplyr` is a collection of transforms you can decompose your task into.
- There are multiple `dplyr` "data service" implementations.
  - Tasks written as a sequence of `dplyr` operations can be moved from service to service.
    - Local `data.frame` / `tbl`
    - `Spark` / `Sparklyr`

Why review `dplyr`?

To make sure we are all *really familiar* with `dplyr` operations before trying to use them on `Spark`.

Single Table Verbs

Manipulate tabular data



`select` `filter` `summarise`

`mutate` `arrange` `group_by`

Two Table Verbs

Join together relational data

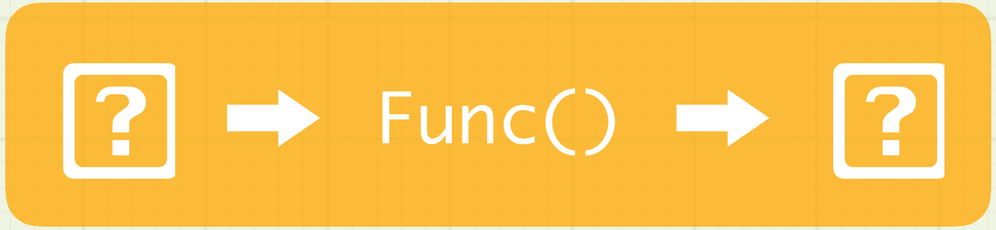
`left_join` `full_join` `union` `bind_cols`

`right_join` `semi_join` `intersect` `bind_rows`

`inner_join` `anti_join` `setdiff`

Single Table Verbs

Manipulate tabular data



`select` `filter` `summarise`

`mutate` `arrange` `group_by`

Two Table Verbs

Join together relational data

`left_join` `full_join` `union` `bind_cols`

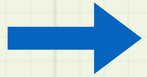
`right_join` `semi_join` `intersect` `bind_rows`

`inner_join` `anti_join` `setdiff`

select()

storms

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21



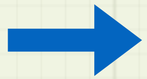
storm	pressure
Alberto	1007
Alex	1009
Allison	1005
Ana	1013
Arlene	1010
Arthur	1010

```
select(storms, storm, pressure)
```

* These data sets are in the EDAWR package

mutate()

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21



storm	wind	pressure	date	ratio
Alberto	110	1007	2000-08-12	9.15
Alex	45	1009	1998-07-30	22.42
Allison	65	1005	1995-06-04	15.46
Ana	40	1013	1997-07-01	25.32
Arlene	50	1010	1999-06-13	20.20
Arthur	45	1010	1996-06-21	22.44

```
mutate(storms, ratio = pressure / wind)
```

* These data sets are in the EDAWR package

logical tests in R

`?Comparison`

<	Less than
>	Greater than
==	Equal to
<=	Less than or equal to
>=	Greater than or equal to
!=	Not equal to
%in%	Group membership
is.na	Is NA
!is.na	Is not NA

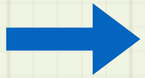
`?base::Logic`

&	boolean and
|	boolean or
xor	exactly or
!	not
any	any true
all	all true

filter()

storms

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21



storm	wind	pressure	date
Alberto	110	1007	2000-08-12

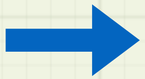
```
filter(storms, wind == max(wind))
```

* These data sets are in the EDAWR package

filter()

storms

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21



storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Allison	65	1005	1995-06-04
Arlene	50	1010	1999-06-13

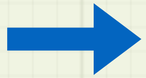
```
filter(storms, wind >= 50)
```

* These data sets are in the EDAWR package

filter()

storms

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21



storm	wind	pressure	date
Alex	45	1009	1998-07-30
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21

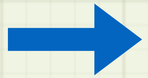
```
filter(storms, wind < 60, wind >= 40)
```

* These data sets are in the EDAWR package

arrange()

storms

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21



storm	wind	pressure	date
Ana	40	1013	1997-07-01
Alex	45	1009	1998-07-30
Arthur	45	1010	1996-06-21
Arlene	50	1010	1999-06-13
Allison	65	1005	1995-06-04
Alberto	110	1007	2000-08-12

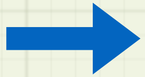
```
arrange(storms, wind)
```

* These data sets are in the EDAWR package

arrange()

storms

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21



storm	wind	pressure	date
Ana	40	1013	1997-07-01
Alex	45	1009	1998-07-30
Arthur	45	1010	1996-06-21
Arlene	50	1010	1999-06-13
Allison	65	1005	1995-06-04
Alberto	110	1007	2000-08-12

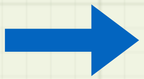
```
arrange(storms, wind)
```

* These data sets are in the EDAWR package

arrange()

storms

storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Alex	45	1009	1998-07-30
Allison	65	1005	1995-06-04
Ana	40	1013	1997-07-01
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21



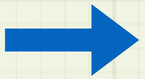
storm	wind	pressure	date
Alberto	110	1007	2000-08-12
Allison	65	1005	1995-06-04
Arlene	50	1010	1999-06-13
Arthur	45	1010	1996-06-21
Alex	45	1009	1998-07-30
Ana	40	1013	1997-07-01

```
arrange(storms, desc(wind))
```

* These data sets are in the EDAWR package

summarise()

city	particle size	amount ($\mu g/m^3$)
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56



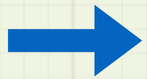
median
22.5

```
summarise(pollution, median = median(amount))
```

* These data sets are in the EDAWR package

summarise()

city	particle size	amount ($\mu g/m^3$)
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56



mean	sum	n
42	252	6

```
summarise(pollution, mean = mean(amount), sum = sum(amount), n = n())
```

* These data sets are in the EDAWR package

city	particle size	amount ($\mu g/m^3$)
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56

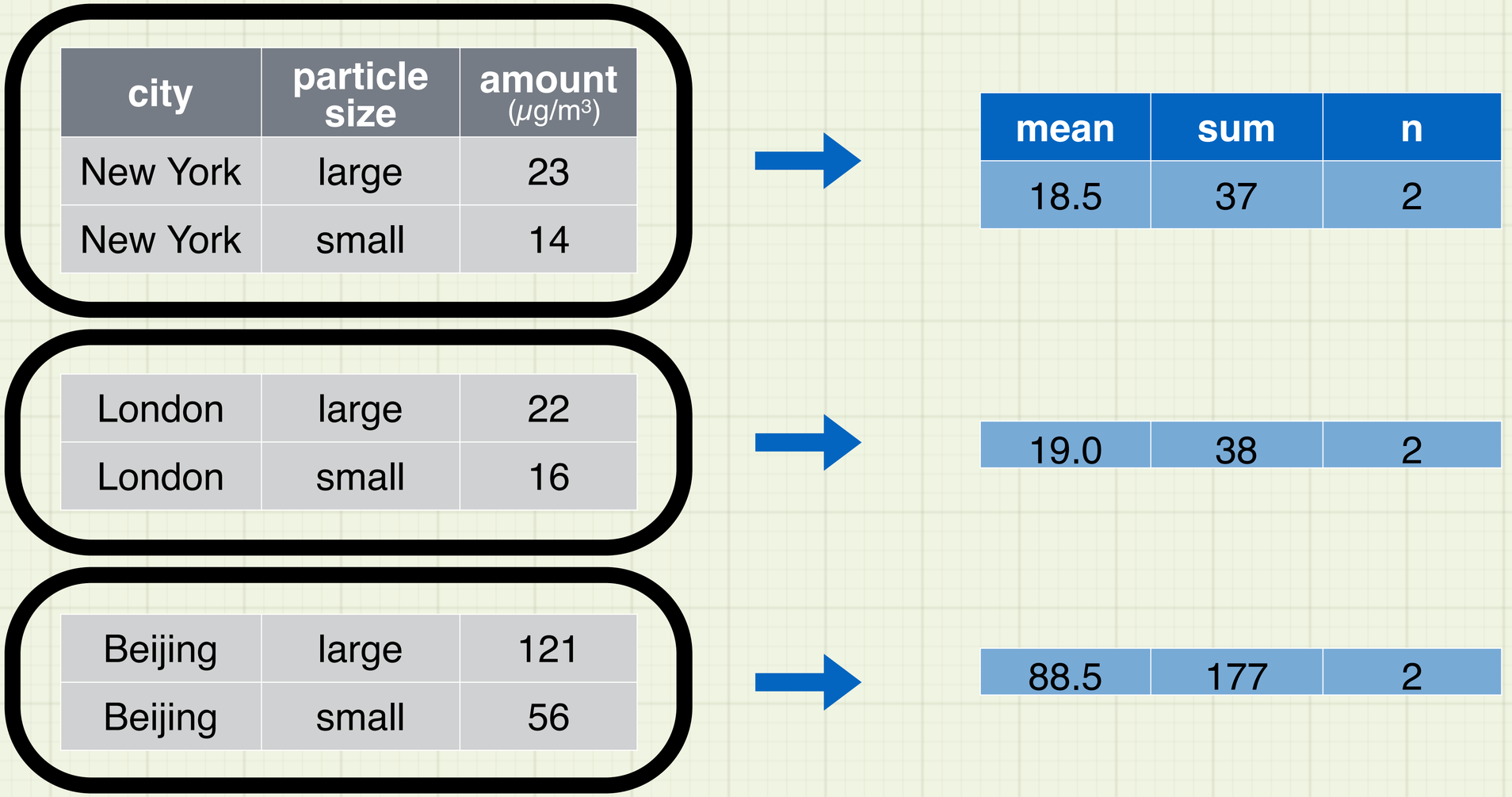
mean	sum	n
42	252	6

* These data sets are in the EDAWR package

city	particle size	amount ($\mu g/m^3$)
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56

mean	sum	n
42	252	6

* These data sets are in the EDAWR package

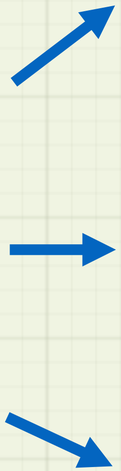


group_by() + summarise()

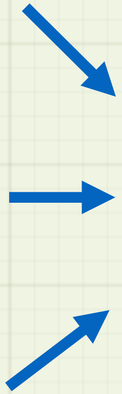
* These data sets are in the EDAWR package

group_by()

city	particle size	amount ($\mu g/m^3$)
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56



city	particle size	amount ($\mu g/m^3$)
New York	large	23
New York	small	14
London	large	22
London	small	16
Beijing	large	121
Beijing	small	56



mean	sum	n
18.5	37	2
19.0	38	2
88.5	177	2

```
p <- group_by(pollution, city)
summarise(p, mean = mean(amount), sum = sum(amount), n = n())
```

Single Table Verbs

Manipulate tabular data

`select` `filter` `summarise`

`mutate` `arrange` `group_by`

Two Table Verbs

Join together relational data

`left_join` `full_join` `union` `bind_cols`

`right_join` `semi_join` `intersect` `bind_rows`

`inner_join` `anti_join` `setdiff`

Win-Vector LLC

Joins

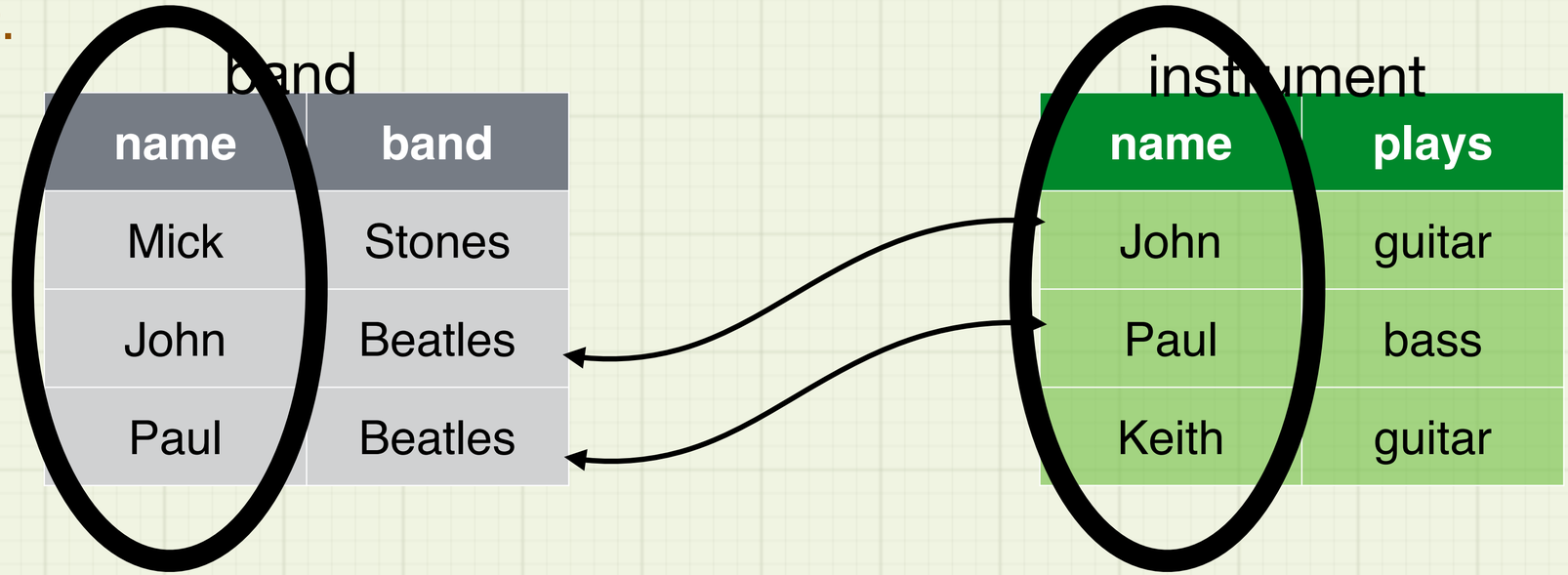
- The core of relational data processing.
- Most important data transforms can be written in terms of a sequence of joins:
  - intersection
  - cross-product
  - lookup
  - lapply / list comprehensions
- Master these and you have mastered data manipulation

Joins: the math

- Joins are implemented *as if:*
  - Each row in each table is paired with every other row in the other table and once more with an extra “no match” row.
    - Two tables with `m` rows and `n` rows respectively could generate as many as (m+1)*(n+1) notional rows.
    - Rows contain columns from both tables. Duplicate column names are disambiguated by appending extra names to the columns.
  - The result is winnowed down to only rows matching the join conditions, and only columns named in the statement.
- Join implementations are *much* more efficient than the above specification.
  - The database implementation examines to join conditions to only generate rows the user wants. Filtering is implicit, unwanted rows and duplicate columns are not generated.

joins: first example

- Task: For each band member look up what, if any instrument they play.
- The right tool:
  - “left join by name” (also called “left join on name”).
    - “left” means keep records from left table
    - “by name” means names must match
- This join can be implemented in time proportional to the smallest of the two tables!
  - *Very* fast.



left_join(): result

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band	plays
Mick	Stones	<NA>
John	Beatles	guitar
Paul	Beatles	bass

```
left_join(band, instrument, by = "name")
```

left_join(): theory

band

name	band
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

Smaller example, so we can illustrate all the notional steps.

```
left_join(band, instrument, by = "name")
```

left_join(): theory

band

name	band
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

Augment each table with a no-match or empty row.

```
left_join(band, instrument, by = "name")
```

left_join(): theory

band

name	band
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band	name	plays
Paul	Beatles	John	guitar
Paul	Beatles	Paul	bass
Paul	Beatles	Keith	guitar
Paul	Beatles		
		John	guitar
		Paul	bass
		Keith	guitar

Form the cross product.

```
left_join(band, instrument, by = "name")
```

left_join(): theory

band

name	band
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band	name	plays
Paul	Beatles	John	guitar
Paul	**Beatles**	**Paul**	**bass**
Paul	Beatles	Keith	guitar
Paul	Beatles		
		John	guitar
		Paul	bass
		Keith	guitar

Cross out rows that don't match specified conditions.

killed by "left" specification

killed by "by = "name"" specification

```
left_join(band, instrument, by = "name")
```

ProTip

- *Always* inspect your intermediate results after joins.
- In particular ***count rows*** and groups of rows to make sure you haven't missed a join condition.
  - Missing a join condition can cause some rows to be duplicated.

right_join()

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band	plays
John	Beatles	guitar
Paul	Beatles	bass
Keith	<NA>	guitar

```
right_join(band, instrument, by = "name")
```

inner_join()

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

=

name	band	plays
John	Beatles	guitar
Paul	Beatles	bass

```
inner_join(band, instrument, by = "name")
```

full_join()

band

name	band
Mick	Stones
John	Beatles
Paul	Beatles

+

instrument

name	plays
John	guitar
Paul	bass
Keith	guitar

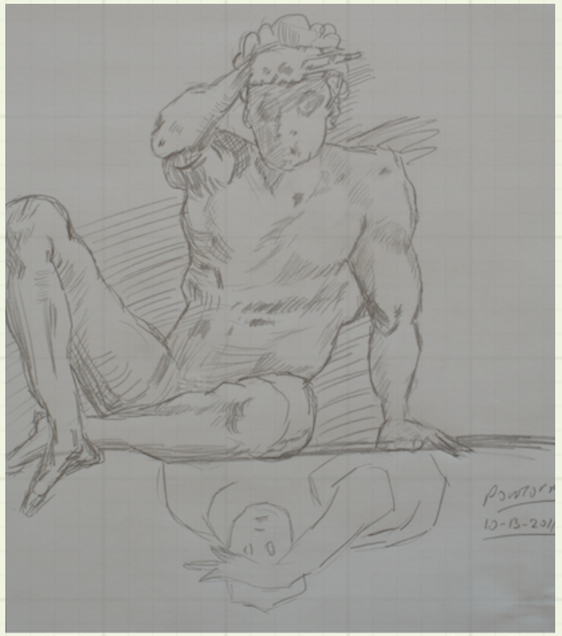
=

name	band	plays
Mick	Stones	<NA>
John	Beatles	guitar
Paul	Beatles	bass
Keith	<NA>	guitar

```
full_join(band, instrument, by = "name")
```

Relational Thinking

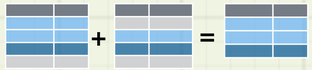
- To think relationally (in terms of joins) you must simultaneously hold three conflicting ideas in your head:
  - join sequences can be made comprehensible
  - joins are powerful
  - joins can be fast.



"Theory versus practice" (after Pontromo).

Recap: Two table verbs

Join together observations with **left_join()**, **right_join()**, **inner_join()**, and **full_join()**



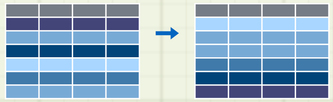
Filter one data set based on another with **semi_join()** and **anti_join()**

Bind data sets together with **bind_rows()** and **bind_cols()**

Do set operations on rows with dplyr's **union()**, **intersect()**, and **setdiff()**

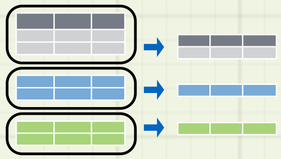
Recap: dplyr one table verbs

Extract columns and rows with **select()** and **filter()**



Arrange rows with **arrange()**.

Make new columns with **mutate()**.



Make groupwise summaries with **group_by()** and **summarise()**.

Next:
`dplyr` exercises