Introducing dplyr

http://bit.ly/intro-dplyr5

Hadley Wickham
@hadleywickham
Chief Scientist, RStudio

March 2014

http://bit.ly/intro-dplyr5

1. Data analysis bottlenecks
2. Introducing dplyr
3. Escaping SQL: a metaprogramming case study

Bottlenecks

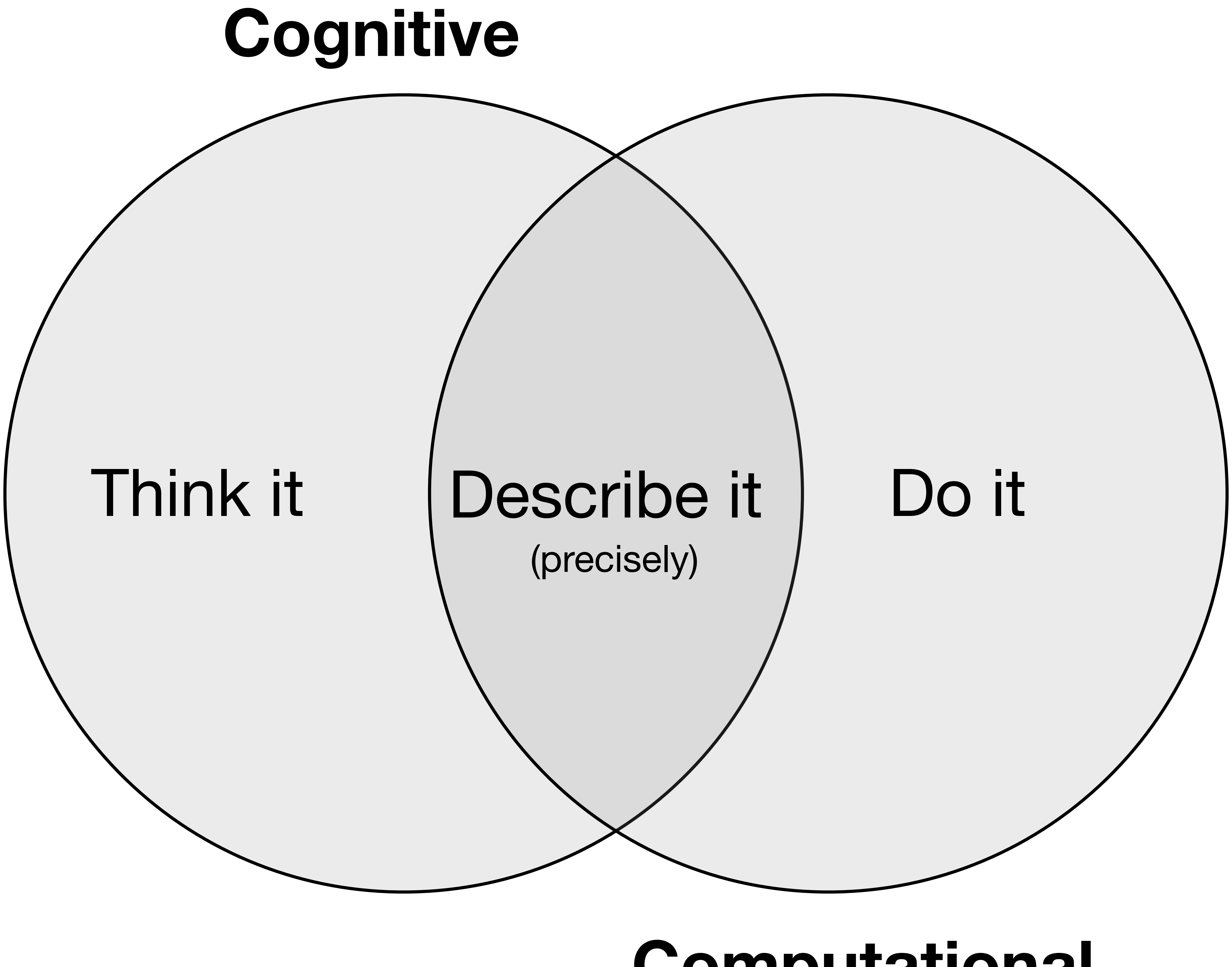
Cost

=

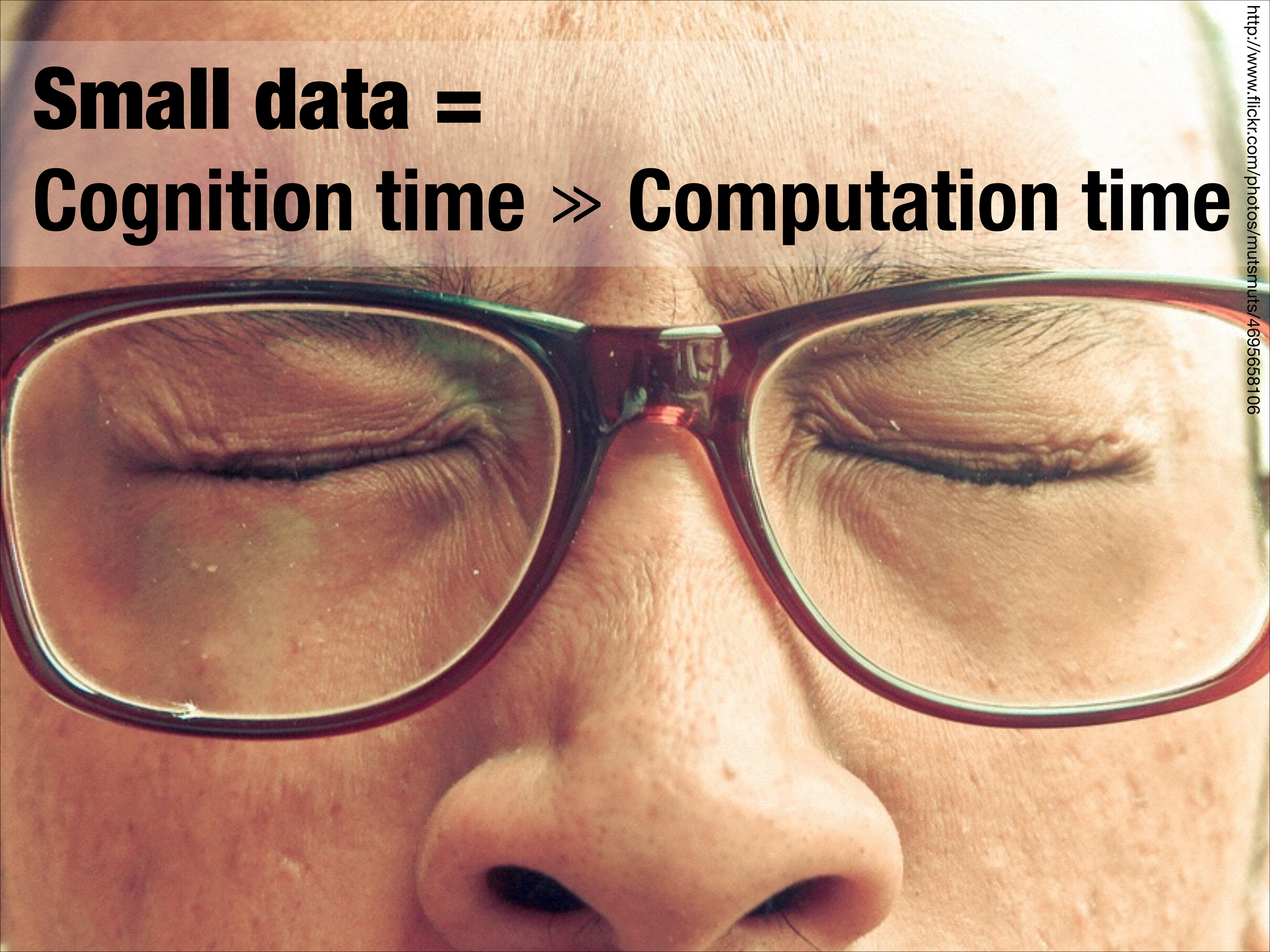
Cognition

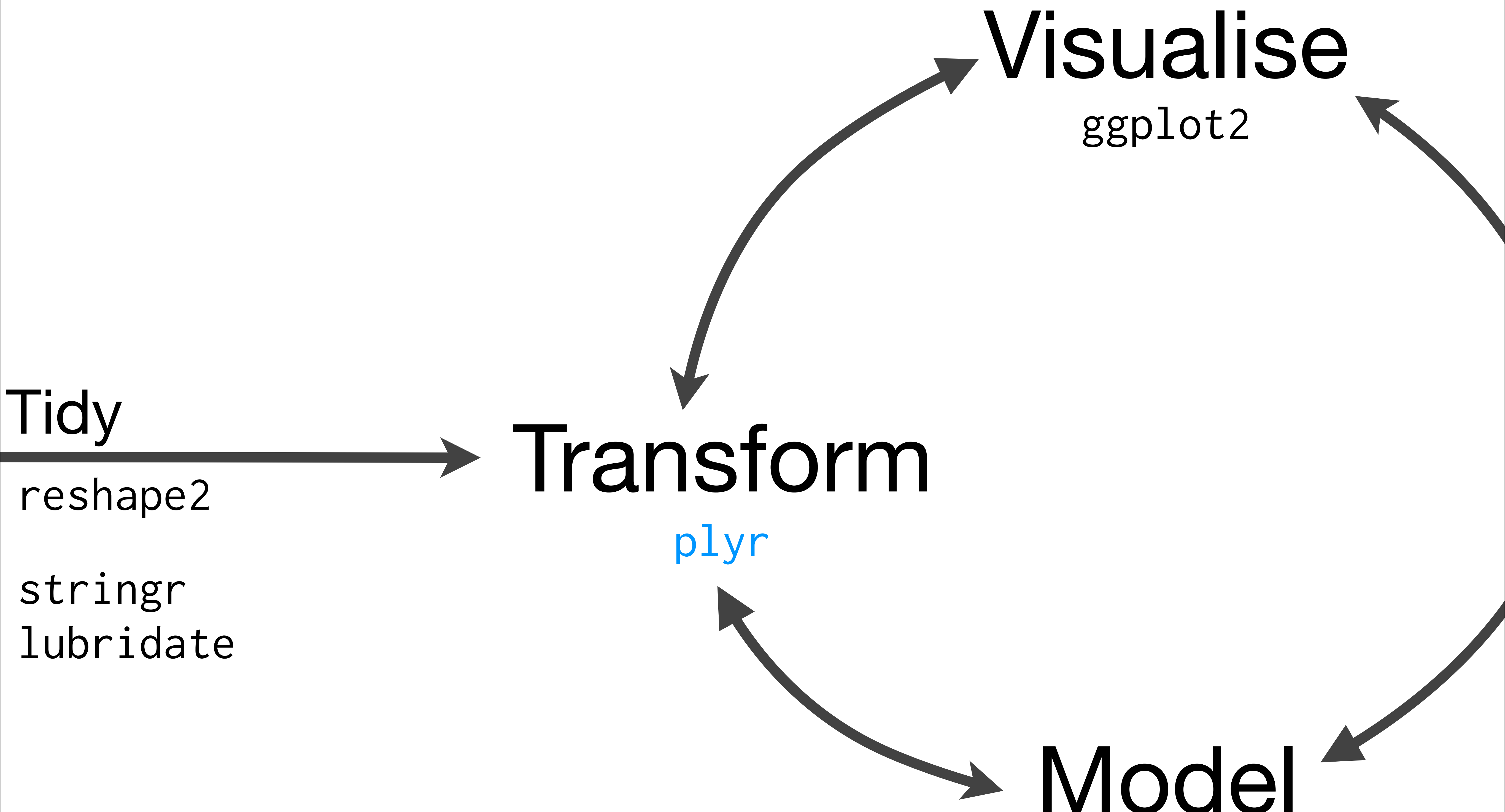
+

Computation


Cognitive
Think it
Describe it
(precisely)
Do it
Computational

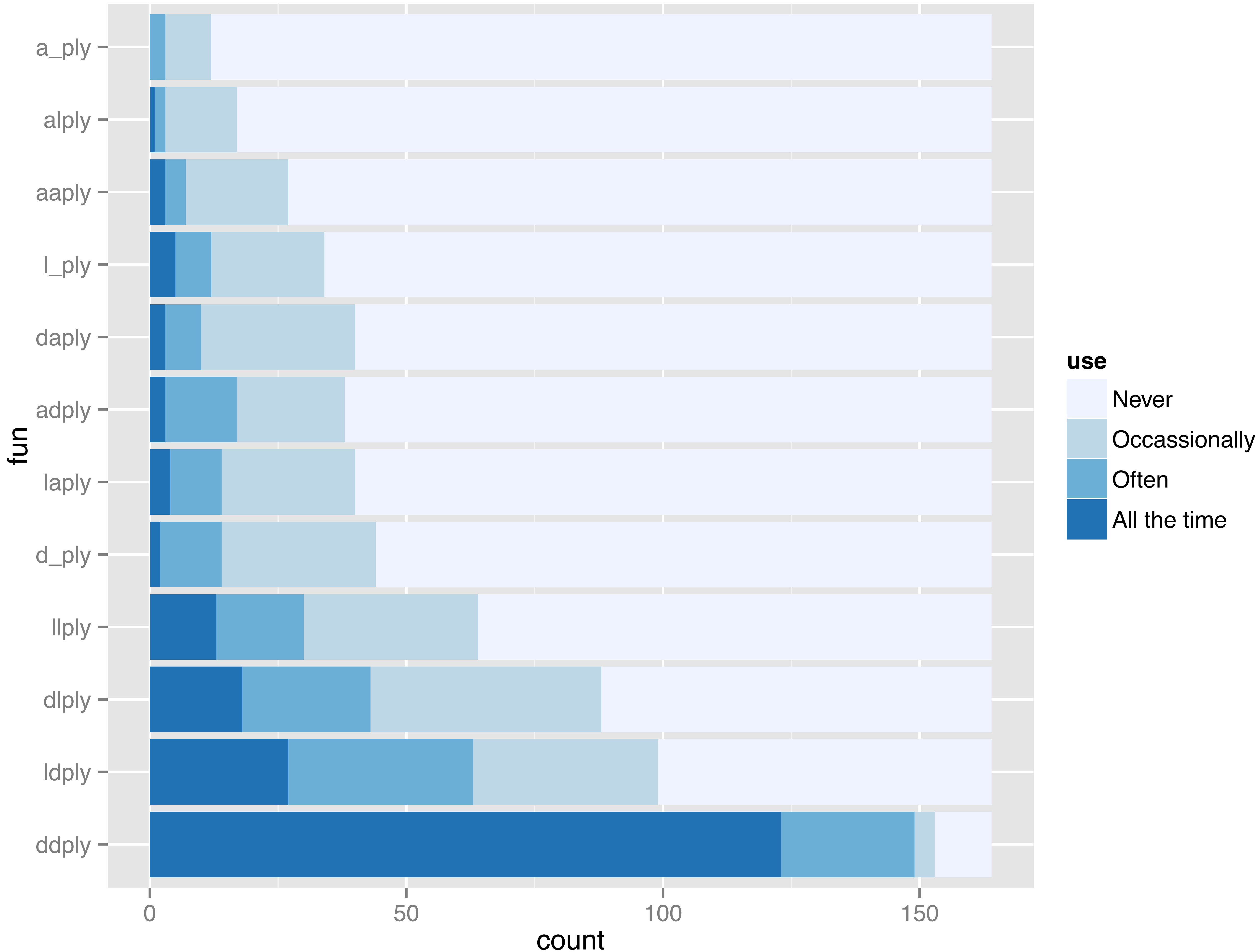
Small data =
Cognition time ≫ Computation time



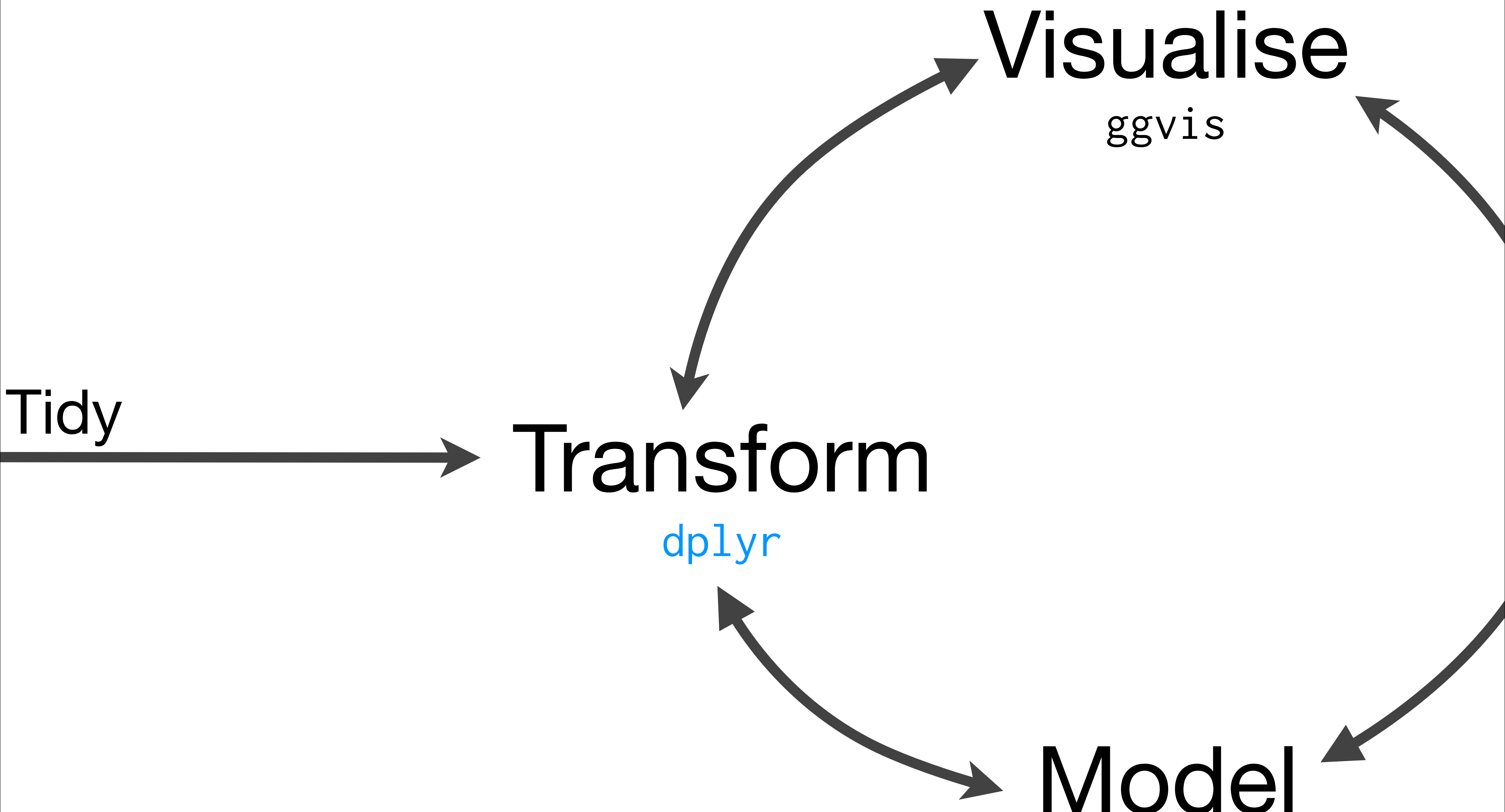

Visualise
ggplot2
Tidy
reshape2
stringr
lubridate
Transform
plyr
Model

	array	data frame	list	nothing
array	aaply	adply	alply	a_ply
data frame	daply	ddply	dlply	d_ply
list	laply	ldply	llply	l_ply
n replicates	raply	rdply	rlply	r_ply
function arguments	maply	mdply	mlply	m_ply

	array	data frame	list	nothing
array	aaply	adply	alply	a_ply
data frame	daply	ddply	dlply	d_ply
list	laply	ldply	llply	l_ply
n replicates	raply	rdply	rlply	r_ply
function arguments	maply	mdply	mlply	m_ply


a_ply
alply
aaply
l_ply
daply
adply
laply
d_ply
llply
dlply
ldply
ddply
fun
0
50
100
150
count
use
Never
Occassionally
Often
All the time

Big data =
Computation time ≫ Cognition time


Visualise
ggvis
Tidy
Transform
dplyr
Model

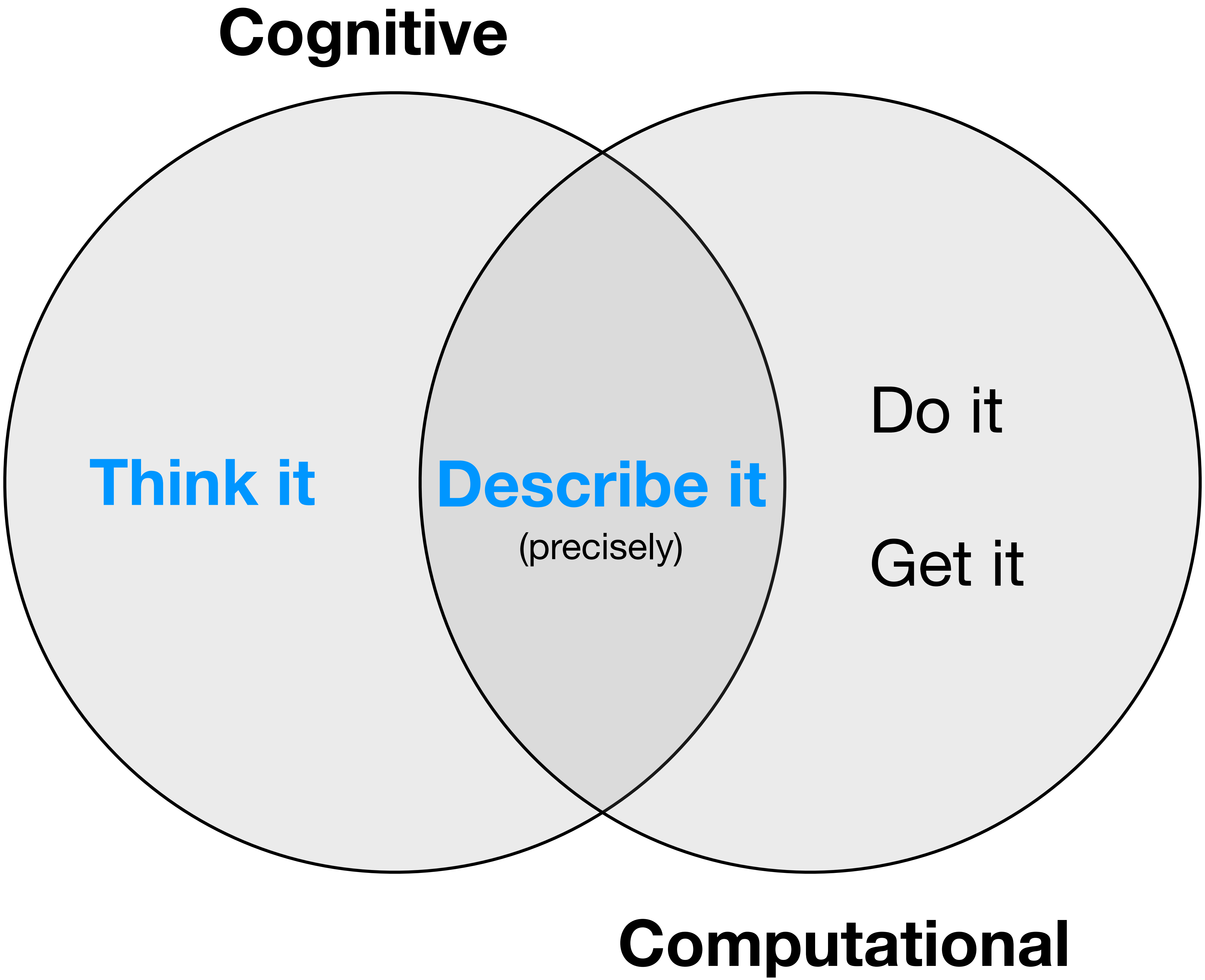
dplyr

```
library(dplyr)
logs <- readRDS("logs.rds") # http://cran-logs.rstudio.com/

print(logs)
#> Source: local data frame [23,454,437 x 10]
#>
#>           date     time    si                 r_arch        r_os   package
#> 1  2013-01-01 00:18:22  551371    2.15.2 x86_64 darwin9.8.0     knitr
#> 2  2013-01-01 00:43:47  220277    2.15.2 x86_64     mingw32 R.devices
#> 3  2013-01-01 00:43:51 3505851    2.15.2 x86_64     mingw32     PSCBS
#> 4  2013-01-01 00:43:53  761107    2.15.2 x86_64     mingw32      R.oo
#> 5  2013-01-01 00:31:15  187381    2.15.2   i686   linux-gnu     akima
#> 6  2013-01-01 00:59:46 2388932    2.15.2 x86_64     mingw32 spacetime
#> 7  2013-01-01 00:31:31   34662    2.15.1 x86_64   linux-gnu    mnormt
#>                     00:30:55  873639    2.15.2 x86_64     mingw32      MASS
#>                     00:43:26  607000        NA     NA          NA     tsDyn
#>                     00:19:25  402583    2.15.2 x86_64 darwin9.8.0   mvtnorm
#>..         ...      ...     ...         ...    ...         ...       ...
#> Variables not shown: version (chr), country (chr), ip_id (int)

print(object.size(logs), units = "GB")
#> 1.6 Gb
```

Commas helpful

No, I don't want to see 10,000 rows!

Not "big" data, but still big


Cognitive
Think it
Describe it
(precisely)
Do it
Get it
Computational

Key insight

There are only a few data analysis verbs **and** they're the same regardless of where your data lives

Single table verbs

+ group by

- **select**: subset variables
- **filter**: subset rows
- **mutate**: add new columns
- **summarise**: reduce to a single row
- **arrange**: re-order the rows

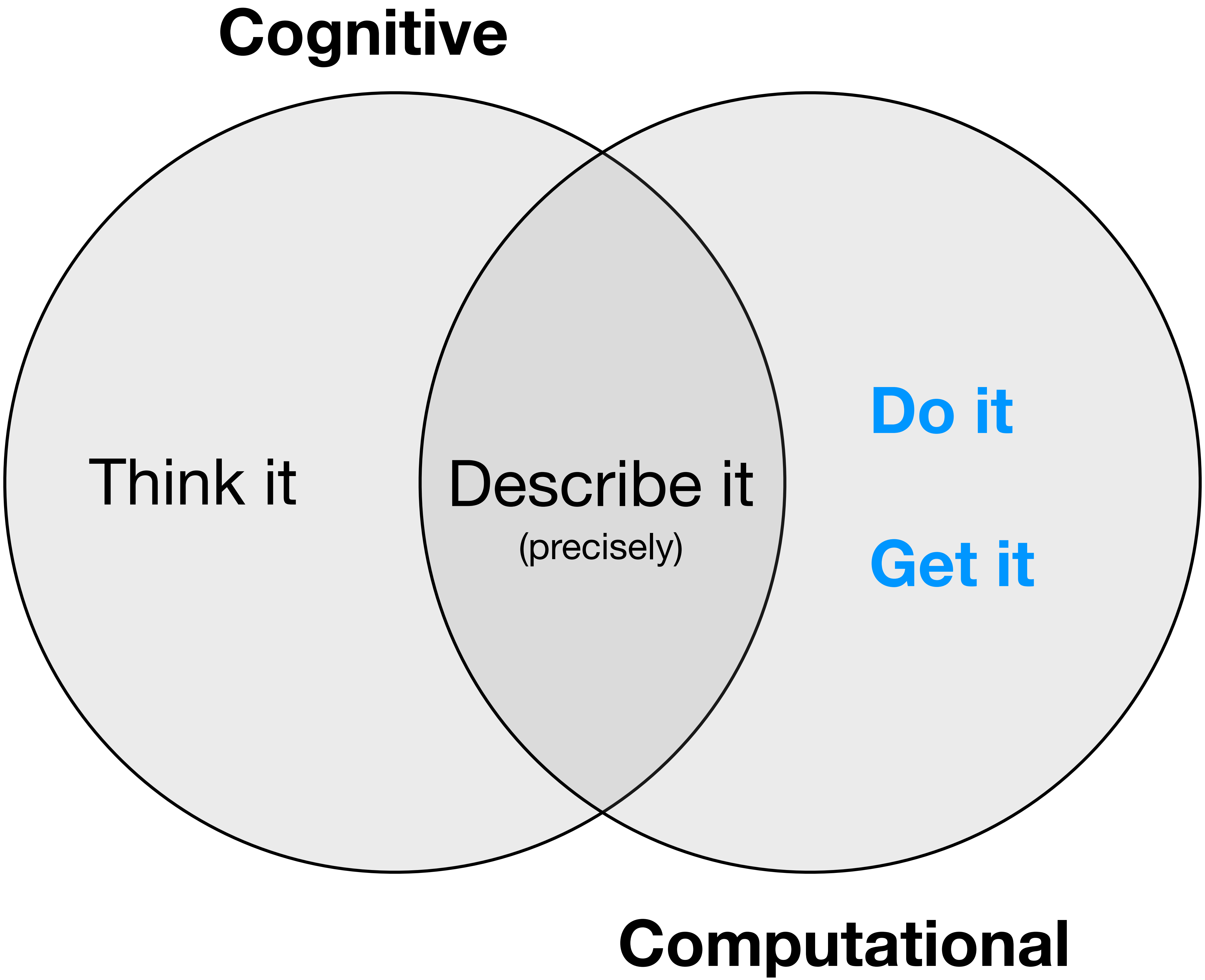
```r
# What packages are most downloaded
packages <- group_by(logs, package)
counts <- summarise(packages, n = n())
head(arrange(counts, desc(n)), 20)

# Takes ~2s (mostly to build index)
```

```
# All functions are pure (no side-effects) -> easy to
# reason about. But function composition is hard to read.
# Solution: x %.% f(y) -> f(x, y)

logs %.%
  group_by(package) %.%
  summarise(n = n()) %.%
  arrange(desc(n)) %.%
  head(20)
```

Multi-table verbs

- **left join**: all x + matching y
- **inner join**: matching x + y
- **semi join**: all x with match in y
- **anti join**: all x without match in y

```r
top20 <- logs %.%
  group_by(package) %.%
  tally(sort = T) %.%
  head(20)

# What if we want to look at how many downloads
# per day for the top 20 packages?
logs20 <- logs %.% semi_join(top20)
```



Cognitive
Think it
Describe it
(precisely)
Do it
Get it
Computational

Local data frames

- High-performance C++. Avoid copies. Avoid R function call overhead with custom interpreter for simple R expressions.
- Thanks to Romain Francois
- (Currently working on automatic parallelisation)

Key insight

Move the computation to the data

dplyr sources

- Local data frame
- Local data table
- Local data cube (experimental)
- RDMS: Postgres, MySQL, SQLite, Oracle, SQL Server, MonetDB
- BigQuery

Translate R to SQL

High-level data manip verbs correspond to high-level component of SQL grammar.

Automatically translate small expressions from R to SQL.

Translation can't be perfect; aiming for semantic equivalency.

```
downloads <- logs_db %.%
  group_by(package) %.%
  summarise(n = n()) %.%
  arrange(desc(n))
downloads$query
# <Query> SELECT "package", "n"
# FROM (
#   SELECT "package", count(*) AS "n"
#   FROM "logs"
#   GROUP BY "package"
# ) AS "_W1"
# ORDER BY "n" DESC
# <PostgreSQLConnection:(8627,0)>

explain(downloads)
downloads
```

Translate R to SQL

High-level data manip verbs correspond to high-level component of SQL grammar.

Automatically translate small expressions from R to SQL.

Translation can't be perfect; aiming for semantic equivalency.

http://adv-r.had.co.nz/dsl.html

```r
translate_sql(Month > 1, hflights)
# <SQL> "Month" > 1.0
translate_sql(Month > 1L, hflights)
# <SQL> "Month" > 1

translate_sql(Dest == "IAD" || Dest == "DCA",
  hflights)
# <SQL> "Dest" = 'IAD' OR "Dest" = 'DCA'

dc <- c("IAD", "DCA")
translate_sql(Dest %in% dc, hflights)
# <SQL> "Dest" IN ('IAD', 'DCA')
```

```
translate_sql(Month > 1, hflights)
# <SQL> "Month" > 1.0
translate_sql(Month > 1L, hflights)
# <SQL> "Month" > 1

translate_sql(Dest == "IAD" || Dest == "DCA",
  hflights)
# <SQL> "Dest" = 'IAD' OR "Dest" = 'DCA'

dc <- c("IAD", "DCA")
translate_sql(Dest %in% dc, hflights)
# <SQL> "Dest" IN ('IAD', 'DCA')
```

Variable names escaped

```
translate_sql(Month > 1, hflights)
# <SQL> "Month" > 1.0
translate_sql(Month > 1L, hflights)
# <SQL> "Month" > 1
```

Correct variable types

```
translate_sql(Dest == "IAD" || Dest == "DCA",
  hflights)
# <SQL> "Dest" = 'IAD' OR "Dest" = 'DCA'

dc <- c("IAD", "DCA")
translate_sql(Dest %in% dc, hflights)
# <SQL> "Dest" IN ('IAD', 'DCA')
```

```
translate_sql(Month > 1, hflights)
# <SQL> "Month" > 1.0
translate_sql(Month > 1L, hflights)
# <SQL> "Month" > 1

translate_sql(Dest == "IAD" || Dest == "DCA",
  hflights)
# <SQL> "Dest" = 'IAD' OR "Dest" = 'DCA'

dc <- c("IAD", "DCA")
translate_sql(Dest %in% dc, hflights)
# <SQL> "Dest" IN ('IAD', 'DCA')
```

SQL strings use '

OR not ||

= not ==

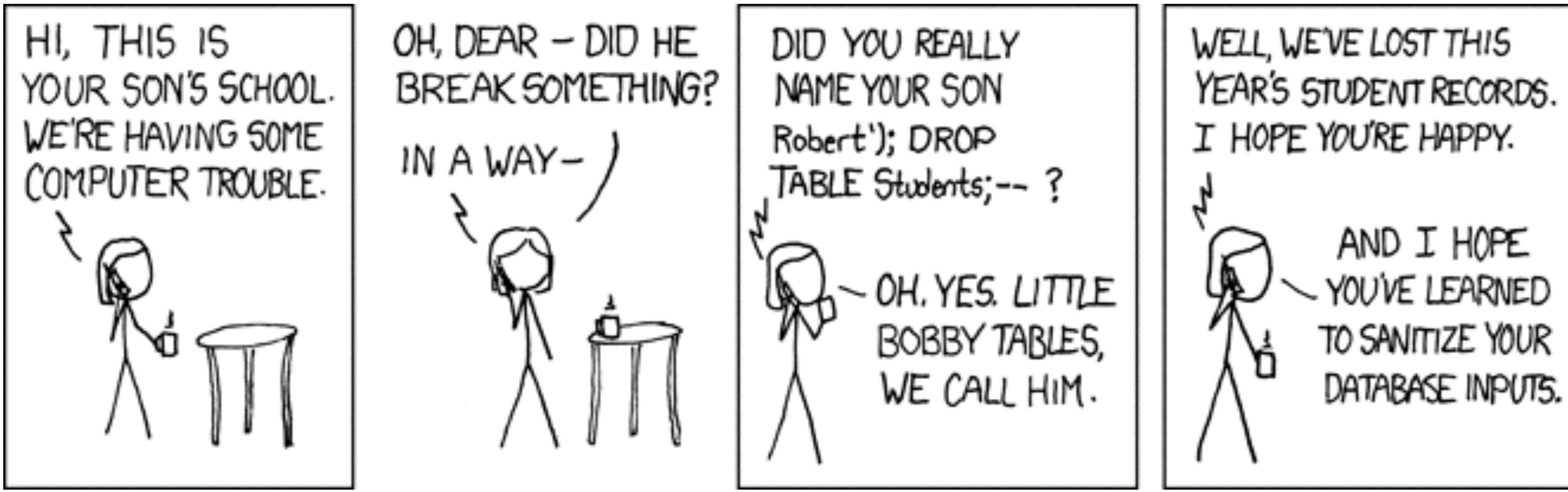
```
translate_sql(Month > 1, hflights)
# <SQL> "Month" > 1.0
translate_sql(Month > 1L, hflights)
# <SQL> "Month" > 1

translate_sql(Dest == "IAD" || Dest == "DCA",
  hflights)
# <SQL> "Dest" = 'IAD' OR "Dest" = 'DCA'

dc <- c("IAD", "DCA")
translate_sql(Dest %in% dc, hflights)
# <SQL> "Dest" IN ('IAD', 'DCA')
```

contents of local variables inserted

http://adv-r.had.co.nz/dsl.html

Metaprogramming

http://adv-r.had.co.nz/Computing-on-the-language.html



http://xkcd.com/327/

```
# The essentials of some sql escaping code:

sql <- function(x) structure(x, class = "sql")
print.sql <- function(x) cat("<SQL>", x, "\n")

escape <- function(x) UseMethod("escape")
escape.sql <- function(x) x
escape.character <- function(x) {
  x <- gsub("'", "''", x, fixed = TRUE)
  sql(paste0("'", x, "'"))
}
```

```
x <- sql("SELECT * FROM Students")
escape(x)
escape(escape(x))

name <- "Robert'); DROP TABLE Students; --"
escape(name)
escape(escape(name))

# How can we make this so easy that we do it
# by default?
```

```
# Want something like paste:
build_sql(”SELECT * FROM students WHERE name = ”,
  name, ”;”)

# name should be escaped, but string shouldn’t be.
#
# Let’s do a little metaprogramming (aka computing
# on the language to escape variables, but not
# strings
```

```
# Take my word that this is the easiest way to
# capture unevaluated expressions
dots <- function(...) {
  eval(substitute(alist(...)))
}

# There are three types of thing that dots can
# return:

# 1. a call
dots(f(x))
# 2. a symbol
dots(x)
# 3. an atomic vector of length 1
dots(1)

dots("SELECT * FROM students WHERE name = ",
  name, ";")
```

```
# Quick quiz: why will dots() never return
# an atomic vector of length > 1?

# or in other words, what will these functions
# return?

dots(1:10)
dots(c("a", "b"))
```

```r
build_sql <- function(...) {
   env <- parent.frame()

   args <- dots(...)
   pieces <- lapply(args, function(arg) {
     if (is.character(arg)) return(arg)

     escape(eval(arg, env))
   })

   paste0(unlist(pieces), collapse = "")
}

build_sql("SELECT * FROM students WHERE name = ",
  name, ";")
```

Conclusions

Future work

http://bit.ly/intro-dplyr5

More top-level verbs: `sample()`, `colwise()`, `cross_join()`

More types of grouping: `rowwise()`, `bootstrap()`, `rollup()`, `binned()`

Better support for arbitrary functions with `do()`.

Google for

“dplyr”

http://bit.ly/intro-dplyr5