How to work efficiently with large datasets

BPLIM Workshop on Empirical Research with Large Datasets

Mauricio M. Cáceres Bravo

20 December 2022

Department of Economics
Brown University

Introduction

What is different about "Big Data"

The main practical difference between small and large large datasets is scale, but why does scale matter?

1. ***Speed:*** Operations can be prohibitively slow.
   Example: An operation runs in a second with 10,000 rows of data; with 100 million rows it would take hours.
2. ***Memory:*** It can be hard to fit in RAM.
   Example: Numeric data is 4 or 8 bytes. With 10,000 rows even 1000 variables take under 100MiB, but with 100 million a *single* variable is 400-800MiB.
3. Some operations can break with big data.

What is different about “Big Data”

The main practical difference between small and large large datasets is scale, but why does scale matter?

1. ***Speed:*** Operations can be prohibitively slow.
   Example: An operation runs in a second with 10,000 rows of data; with 100 million rows it would take hours.
2. ***Memory:*** It can be hard to fit in RAM.
   Example: Numeric data is 4 or 8 bytes. With 10,000 rows even 1000 variables take under 100MiB, but with 100 million a *single* variable is 400-800MiB.
3. Some operations can break with big data.

What is different about “Big Data”

The main practical difference between small and large large datasets is scale, but why does scale matter?

1. ***Speed:*** Operations can be prohibitively slow.
   Example: An operation runs in a second with 10,000 rows of data; with 100 million rows it would take hours.
2. ***Memory:*** It can be hard to fit in RAM.
   Example: Numeric data is 4 or 8 bytes. With 10,000 rows even 1000 variables take under 100MiB, but with 100 million a *single* variable is 400-800MiB.
3. Some operations can break with big data.

What is different about "Big Data"

The main practical difference between small and large large datasets is scale, but why does scale matter?

1. ***Speed:*** Operations can be prohibitively slow.
   Example: An operation runs in a second with 10,000 rows of data; with 100 million rows it would take hours.
2. ***Memory:*** It can be hard to fit in RAM.
   Example: Numeric data is 4 or 8 bytes. With 10,000 rows even 1000 variables take under 100MiB, but with 100 million a *single* variable is 400-800MiB.
3. Some operations can break with big data.

Today

Discuss big data primarily in the context of Stata:

1. Introduction
2. What can break?
3. Memory Management
4. Efficient Code (main focus)

What can break with big data?

Example: Storing IDs

```
clear
qui set obs `=2^24+5'
gen id = _n
format %21.0gc id
list in -10/l
disp "`:type id'" // float
```

```
          +------------+
          |         id |
          |------------|
16777212. | 16,777,212 |
16777213. | 16,777,213 |
16777214. | 16,777,214 |
16777215. | 16,777,215 |
16777216. | 16,777,216 |
          |------------|
16777217. | 16,777,216 |
16777218. | 16,777,218 |
16777219. | 16,777,220 |
16777220. | 16,777,220 |
16777221. | 16,777,220 |
          +------------+
```

Example: Storing IDs

You *could* set double as the default type, but that can consume a lot of memory! Efficient solution:

```
gen `c(obs_t)' id = _n
```

`c(obs_t)` is the smallest ***integer*** type that can store the number of observations correctly.

```
clear
qui set obs 1
disp "`c(obs_t)'" // byte
qui set obs `=maxbyte()+1'
disp "`c(obs_t)'" // int
qui set obs `=maxint()+1'
disp "`c(obs_t)'" // long
qui set obs `=maxlong()+1'
disp "`c(obs_t)'" // double
```

Memory Management

Minimize Memory Use

- Store variables in the smallest sensible type.
  - `double`: Largest number type (8 bytes).
  - `float`: Stata default type (4 bytes).
  - `long`: Largest integer type (4 bytes).
  - `int`: Small integer type (2 bytes).
  - `byte`: Smallest integer type (1 bytes).

  (See **`help`** `data_types` for more.)
- **`compress`**: Recast each variable (numeric and string) to its smallest possible type without data loss.
- Organize your data!
  - No need to store every variable in single file.
  - Encode strings as numbers and save their values separately (for few levels can use value labels).

Minimize Memory Use

- Store variables in the smallest sensible type.
    - `double`: Largest number type (8 bytes).
    - `float`: Stata default type (4 bytes).
    - `long`: Largest integer type (4 bytes).
    - `int`: Small integer type (2 bytes).
    - `byte`: Smallest integer type (1 bytes).

  (See **`help`** `data_types` for more.)
- **`compress`**: Recast each variable (numeric and string) to its smallest possible type without data loss.
- Organize your data!
    - No need to store every variable in single file.
    - Encode strings as numbers and save their values separately (for few levels can use value labels).

Minimize Memory Use

- Store variables in the smallest sensible type.
  - `double`: Largest number type (8 bytes).
  - `float`: Stata default type (4 bytes).
  - `long`: Largest integer type (4 bytes).
  - `int`: Small integer type (2 bytes).
  - `byte`: Smallest integer type (1 bytes).

  (See **`help`** `data_types` for more.)
- **`compress`**: Recast each variable (numeric and string) to its smallest possible type without data loss.
- Organize your data!
  - No need to store every variable in single file.
  - Encode strings as numbers and save their values separately (for few levels can use value labels).

Cautionary Tale: Swapping

RAM is both limited and essential for an operating system:

- If your program consumes too much, the OS will eventually refuse to give it more memory.
- In extreme cases, the program will die.
- More commonly, it will start to swap: The OS will move your program's data to disk and grind execution to a halt.

Solutions

- Minimize your program's memory footprint.
- Get more memory! But easier said than done.
- Keep only the essential variables in memory.
- Chunk your program execution. (NB: Stata stores data by row, so this can be efficient both in terms of memory and speed.)

Cautionary Tale: Swapping

RAM is both limited and essential for an operating system:

- If your program consumes too much, the OS will eventually refuse to give it more memory.
- In extreme cases, the program will die.
- More commonly, it will start to swap: The OS will move your program's data to disk and grind execution to a halt.

Solutions

- Minimize your program's memory footprint.
- Get more memory! But easier said than done.
- Keep only the essential variables in memory.
- Chunk your program execution. (NB: Stata stores data by row, so this can be efficient both in terms of memory and speed.)

Efficient Code

What does it mean to code efficiently?

Code that executes quickly is part of coding efficiently, but:

1. Trade-off: Writing code fast vs writing fast code.
2. Planning ahead: Run similar tasks all at once.
3. Look for available solutions: Someone else might have solved your problem.
4. Improve your algorithms: Making a bad algorithm very fast is often slower than coding an efficient algorithm.
5. Pick the best tool for the job.

What does it mean to code efficiently?

Code that executes quickly is part of coding efficiently, but:

1. Trade-off: Writing code fast vs writing fast code.
2. Planning ahead: Run similar tasks all at once.
3. Look for available solutions: Someone else might have solved your problem.
4. Improve your algorithms: Making a bad algorithm very fast is often slower than coding an efficient algorithm.
5. Pick the best tool for the job.

What does it mean to code efficiently?

Code that executes quickly is part of coding efficiently, but:

1. Trade-off: Writing code fast vs writing fast code.
2. Planning ahead: Run similar tasks all at once.
3. Look for available solutions: Someone else might have solved your problem.
4. Improve your algorithms: Making a bad algorithm very fast is often slower than coding an efficient algorithm.
5. Pick the best tool for the job.

What does it mean to code efficiently?

Code that executes quickly is part of coding efficiently, but:

1. Trade-off: Writing code fast vs writing fast code.
2. Planning ahead: Run similar tasks all at once.
3. Look for available solutions: Someone else might have solved your problem.
4. Improve your algorithms: Making a bad algorithm very fast is often slower than coding an efficient algorithm.
5. Pick the best tool for the job.

What does it mean to code efficiently?

Code that executes quickly is part of coding efficiently, but:

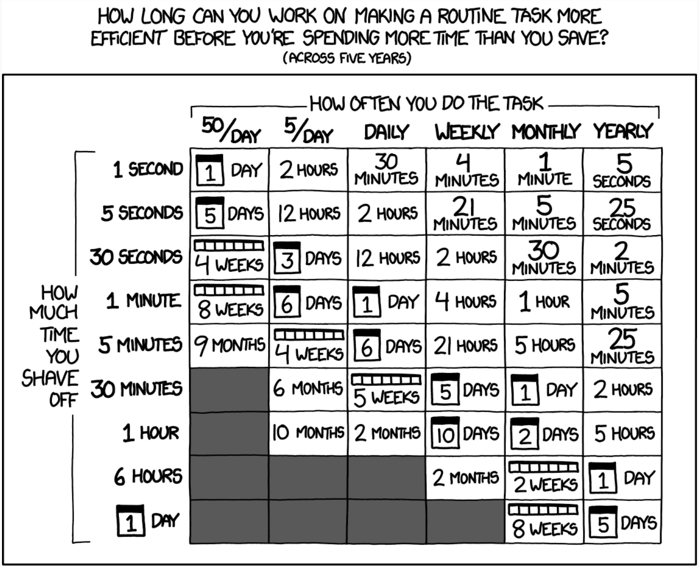
1. Trade-off: Writing code fast vs writing fast code.
2. Planning ahead: Run similar tasks all at once.
3. Look for available solutions: Someone else might have solved your problem.
4. Improve your algorithms: Making a bad algorithm very fast is often slower than coding an efficient algorithm.
5. Pick the best tool for the job.

What does it mean to code efficiently?

Code that executes quickly is part of coding efficiently, but:

1. Trade-off: Writing code fast vs writing fast code.
2. Planning ahead: Run similar tasks all at once.
3. Look for available solutions: Someone else might have solved your problem.
4. Improve your algorithms: Making a bad algorithm very fast is often slower than coding an efficient algorithm.
5. Pick the best tool for the job.

Trade-off: Writing faster code is slower



HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

HOW MUCH TIME YOU SHAVE OFF	HOW OFTEN YOU DO THE TASK					
	50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
30 MINUTES		6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
1 HOUR		10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
6 HOURS				2 MONTHS	2 WEEKS	1 DAY
1 DAY					8 WEEKS	5 DAYS

Source: XKCD 1205 ↗.

Planning Ahead: Similar Operations

- If you will be doing many operations by group, sorting the data will make each operation much faster. (NB: Gtools functions are also faster on sorted data.)
- Pre-computing variables that will be re-used instead of creating them on the fly.

Planning Ahead: Very Long Operations

Sometimes a program that takes a long time to run is inevitable:

- Run overnight or over a break. (So program does not compete for computing time or your own time.)
- Include checkpoints:
  - Do not write a single function to do all your work.
  - Group tasks into programs, and save your data along the way.
  - Print messages along your program to tell you where you are (can check log while program executes).

Planning Ahead: Very Long Operations

```stata
program part1
   display "part 1, task 1"
   // ...
   display "part 1, task 2"
   // ...
end

program part2
   display "part 2, task 1"
   // ...
end

part1
save part1.dta
display "finished part 1"

part2
save part2.dta
display "finished part 2"
```

Look for Available Solutions

Some popular user-written Stata packages:

- `reghdfe` (and `ivreghdfe`, `ppmlhdfe`): High-dimensional fixed effects for regression modes.
- `parallel`: Parallelize code execution.
- `gtools`: Fast by-able data management and summary statistics (authored by yours truly).

Gtools Overview

Original impetus for `gtools`:

- **`collapse`**, **`egen`**, and **`merge`** were main bottlenecks on program operating on 400M rows of data (30M groups).
- Up to Stata 16, `gcollapse` was several times faster (and in some cases 100s of times faster). Stata 17 massively improved **`collapse`**; now seldom slower (specially in MP).
- Over the years `gtools` expanded, and most commands remain much faster even in Stata 17.

Gtools Overview

Original impetus for `gtools`:

- **`collapse`**, **`egen`**, and **`merge`** were main bottlenecks on program operating on 400M rows of data (30M groups).
- Up to Stata 16, `gcollapse` was several times faster (and in some cases 100s of times faster). Stata 17 massively improved **`collapse`**; now seldom slower (specially in MP).
- Over the years `gtools` expanded, and most commands remain much faster even in Stata 17.

Gtools Showcase: Speed vs Core Stata

For more details, visit gtools.readthedocs.io

Gtools	Stata	Speedup	Extra features
`gcollapse`	`collapse`	−0.5 to 2	`merge`, various functions
`greshape`	`reshape`	4 to 20	Various
`gegen`	`egen`	4 to 20	Various, incl weights
`gquantiles`	`xtile`	10 to 30	Various, incl `by()`
	`pctile`	3 to 40	Ibid.
	`_pctile`	3 to 40	Ibid.
`gstats tab`	`tabstat`	5 to 50	Various
`gcontract`	`contract`	3 to 7	
`gisid`	`isid`	4 to 30	`if`, `in`
`glevelsof`	`levelsof`	2 to 10	Various
`gduplicates`	`duplicates`	3 to 15	

Gtools Showcase: Speed vs User Commands

Gtools	Similar (SSC/SJ)	Speedup	Extra features?
`gstats winsor`	`winsor2`	10 to 40	Weights
`gunique`	`unique`	4 to 25	
`gdistinct`	`distinct`	4 to 25	
`gstats range`	`rangestat`	10 to 20	Weights

Gtools commands without equivalent:

- `gtop`: Print most frequent (modal) levels of `varlist`.
- `gstats transform`: Various transformations (e.g. `cumsum`, `moving`, `shift`, `wrank`).
- `gstats hdfe`: Residualize variables (absorb fixed effects).

Gtools Spotlight: gtop

Display frequency table with most common (modal) groups defined by a varlist. Example:

```
sysuse auto, clear
gtop rep78, ntop(3)

             rep78 |    N  Cum   Pct (%)   Cum Pct (%)
 -------------------------------------------------------
                 3 |   30   30      40.5          40.5
                 4 |   18   48      24.3          64.9
                 5 |   11   59      14.9          79.7
 -------------------------------------------------------
  Other (3 groups) |   15   74      20.3         100.0
```

Takes many variables, weights, if/in; most useful when exploring data. Docs: gtools.readthedocs.io/en/latest/usage/gtoplevelsof ↗ .

Gtools Spotlight: **greshape**

Same syntax as `reshape`, with some extras:

- Option `dropmiss`: When reshaping long, drop observations if every reshaped variable is missing for that row.
- Option `match(regex)`: When reshaping long, match stubs to variables using regular expressions.
- Option `j()` (alias `keys()`) accepts a `varlist` when reshaping wide.
- Supports `greshape gather` and `greshape spread`, which are analogues to the `tidyr` functions.

Docs: gtools.readthedocs.io/en/latest/usage/greshape ↗ .

Gtools Spotlight: `gquantiles`

Faster percentiles (`xtile`, `pctile`); also by-able.

```
set seed 1729
clear
set obs 10000000
set type double
gen group = int(runiform() * 100)
gen x = runiform()

// Analogous to pctile and xtile, but by group.
gquantiles pctile = x, by(group) strict pctile nq(10) genp(perc)
gquantiles xtile1 = x, by(group) strict xtile  nq(10)

// Analogous to xtile's cutpoints option, but by group. cutquantiles
// interprets percentile instead of number cutoffs.
gquantiles xtile2 = x, by(group) strict xtile  cutpoints(pctile)  cutby
gquantiles xtile3 = x, by(group) strict xtile  cutquantiles(perc) cutby

assert xtile1 == xtile2
assert xtile1 == xtile3
```

Docs: gtools.readthedocs.io/en/latest/usage/gquantiles .

Gtools Spotlight: gstats tab

Very similar to tabstat:

```
sysuse auto, clear
gstats tab price, by(foreign)

  foreign |    n        sum       mean        min        max         sd
-----------------------------------------------------------------------
 Domestic |   52     315766   6072.423       3291      15906   3097.104
  Foreign |   22     140463   6384.682       3748      12990   2621.915
-----------------------------------------------------------------------
```

- Supports additional functions (same functions as gcollapse and gegen).
- Can save output in mata: Faster alternative to gcollapse when number of groups is small.

Docs: gtools.readthedocs.io/en/latest/usage/gstats_summarize ↗

.

Gtools Spotlight: Option `-replace-`

Many `gtools` commands accept the option `replace` in order to replace existing targets; in most cases, this saves time and memory by avoiding re-generating them. Examples:

```
sysuse auto, clear
gegen mean_price = mean(price), by(foreign)
gegen sd_price   = sd(price),   by(foreign)

gcollapse (mean) mean_price=price ///
          (sd)   sd_price=price,  ///
          by(rep78) merge replace

gquantiles bins_price=price, by(foreign) nq(2)
gquantiles bins_price=price, by(foreign) nq(5) replace
```

In general, every `gtools` command that generates a target accepts `replace`. ***Warning:*** This does ***not*** upgrade variable types, so use with caution!

Improve your Algorithms

- A parallelized and extremely efficient loop is slower than the equivalent vector operation.
- The fastest regression with a full set of fixed-effect indicators is slower than `reghdfe`.
- Code tailored to your specific task is faster than general-purpose code.

Improve your Algorithms

- A parallelized and extremely efficient loop is slower than the equivalent vector operation.
- The fastest regression with a full set of fixed-effect indicators is slower than `reghdfe`.
- Code tailored to your specific task is faster than general-purpose code.

Improve your Algorithms

- A parallelized and extremely efficient loop is slower than the equivalent vector operation.
- The fastest regression with a full set of fixed-effect indicators is slower than `reghdfe`.
- Code tailored to your specific task is faster than general-purpose code.

Improve your Algorithms

- A parallelized and extremely efficient loop is slower than the equivalent vector operation.
- The fastest regression with a full set of fixed-effect indicators is slower than `reghdfe`.
- Code tailored to your specific task is faster than general-purpose code.

Algorithm Showdown Example: Merge

Often it is possible to merge on single, integer IDs (vs arbitrary sets of variables).

```
merge 1:1 id using data.dta
// vs
merge 1:1 var_double var_str8 var_long // ...
```

Sorting and matching on a single variable is much faster than sorting and matching many variables of different types, regardless of how efficient you make the sort or the merge itself!

The best tool for the job

No program is the best at everything!

- Stata: Easy to use, but can be slow in places.
- C: Extremely fast (e.g. underpins gtools) but hard to learn.
- Mata: Stata's embedded matrix language, can be used to speed-up many simple tasks.
- Frames: Available from Stata 16, allows the user to have multiple datasets in memory.
- Python: Built-in interface from Stata 16; allows direct interaction with Python.

The best tool for the job

No program is the best at everything!

- Stata: Easy to use, but can be slow in places.
- C: Extremely fast (e.g. underpins gtools) but hard to learn.
- Mata: Stata's embedded matrix language, can be used to speed-up many simple tasks.
- Frames: Available from Stata 16, allows the user to have multiple datasets in memory.
- Python: Built-in interface from Stata 16; allows direct interaction with Python.

The best tool for the job

No program is the best at everything!

- Stata: Easy to use, but can be slow in places.
- C: Extremely fast (e.g. underpins gtools) but hard to learn.
- Mata: Stata's embedded matrix language, can be used to speed-up many simple tasks.
- Frames: Available from Stata 16, allows the user to have multiple datasets in memory.
- Python: Built-in interface from Stata 16; allows direct interaction with Python.

The best tool for the job

No program is the best at everything!

- Stata: Easy to use, but can be slow in places.
- C: Extremely fast (e.g. underpins gtools) but hard to learn.
- Mata: Stata's embedded matrix language, can be used to speed-up many simple tasks.
- Frames: Available from Stata 16, allows the user to have multiple datasets in memory.
- Python: Built-in interface from Stata 16; allows direct interaction with Python.

The best tool for the job

No program is the best at everything!

- Stata: Easy to use, but can be slow in places.
- C: Extremely fast (e.g. underpins gtools) but hard to learn.
- Mata: Stata's embedded matrix language, can be used to speed-up many simple tasks.
- Frames: Available from Stata 16, allows the user to have multiple datasets in memory.
- Python: Built-in interface from Stata 16; allows direct interaction with Python.

The best tool for the job

No program is the best at everything!

- Stata: Easy to use, but can be slow in places.
- C: Extremely fast (e.g. underpins gtools) but hard to learn.
- Mata: Stata's embedded matrix language, can be used to speed-up many simple tasks.
- Frames: Available from Stata 16, allows the user to have multiple datasets in memory.
- Python: Built-in interface from Stata 16; allows direct interaction with Python.

Thank you!