
The other day

```
iex(1)>
```

```
iex(1)> defmodule RepeatN do
...(1)>   def repeat_n(_function, 0) do
...(1)>     # noop
...(1)>   end
...(1)>   def repeat_n(function, 1) do
...(1)>     function.()
...(1)>   end
...(1)>   def repeat_n(function, count) do
...(1)>     function.()
...(1)>     repeat_n(function, count - 1)
...(1)>   end
...(1)> end
{:module, RepeatN, ...}
```

```
iex(1)> defmodule RepeatN do
...(1)>   def repeat_n(_function, 0) do
...(1)>     # noop
...(1)>   end
...(1)>   def repeat_n(function, 1) do
...(1)>     function.()
...(1)>   end
...(1)>   def repeat_n(function, count) do
...(1)>     function.()
...(1)>     repeat_n(function, count - 1)
...(1)>   end
...(1)> end
{:module, RepeatN, ...}
iex(2)> :timer.tc fn -> RepeatN.repeat_n(fn -> 0 end, 100) end
{210, 0}
```

```
iex(1)> defmodule RepeatN do
...(1)>   def repeat_n(_function, 0) do
...(1)>     # noop
...(1)>   end
...(1)>   def repeat_n(function, 1) do
...(1)>     function.()
...(1)>   end
...(1)>   def repeat_n(function, count) do
...(1)>     function.()
...(1)>     repeat_n(function, count - 1)
...(1)>   end
...(1)> end
{:module, RepeatN, ...}
iex(2)> :timer.tc fn -> RepeatN.repeat_n(fn -> 0 end, 100) end
{210, 0}
iex(3)> list = Enum.to_list(1..100)
[...]
iex(4)> :timer.tc fn -> Enum.each(list, fn(_) -> 0 end) end
{165, :ok}
```

```
iex(1)> defmodule RepeatN do
...(1)>   def repeat_n(_function, 0) do
...(1)>     # noop
...(1)>   end
...(1)>   def repeat_n(function, 1) do
...(1)>     function.()
...(1)>   end
...(1)>   def repeat_n(function, count) do
...(1)>     function.()
...(1)>     repeat_n(function, count - 1)
...(1)>   end
...(1)> end
{:module, RepeatN, ...}
iex(2)> :timer.tc fn -> RepeatN.repeat_n(fn -> 0 end, 100) end
{210, 0}
iex(3)> list = Enum.to_list(1..100)
[...]
iex(4)> :timer.tc fn -> Enum.each(list, fn(_) -> 0 end) end
{165, :ok}
iex(5)> :timer.tc fn -> Enum.each(list, fn(_) -> 0 end) end
{170, :ok}
iex(6)> :timer.tc fn -> Enum.each(list, fn(_) -> 0 end) end
{184, :ok}
```


Success!

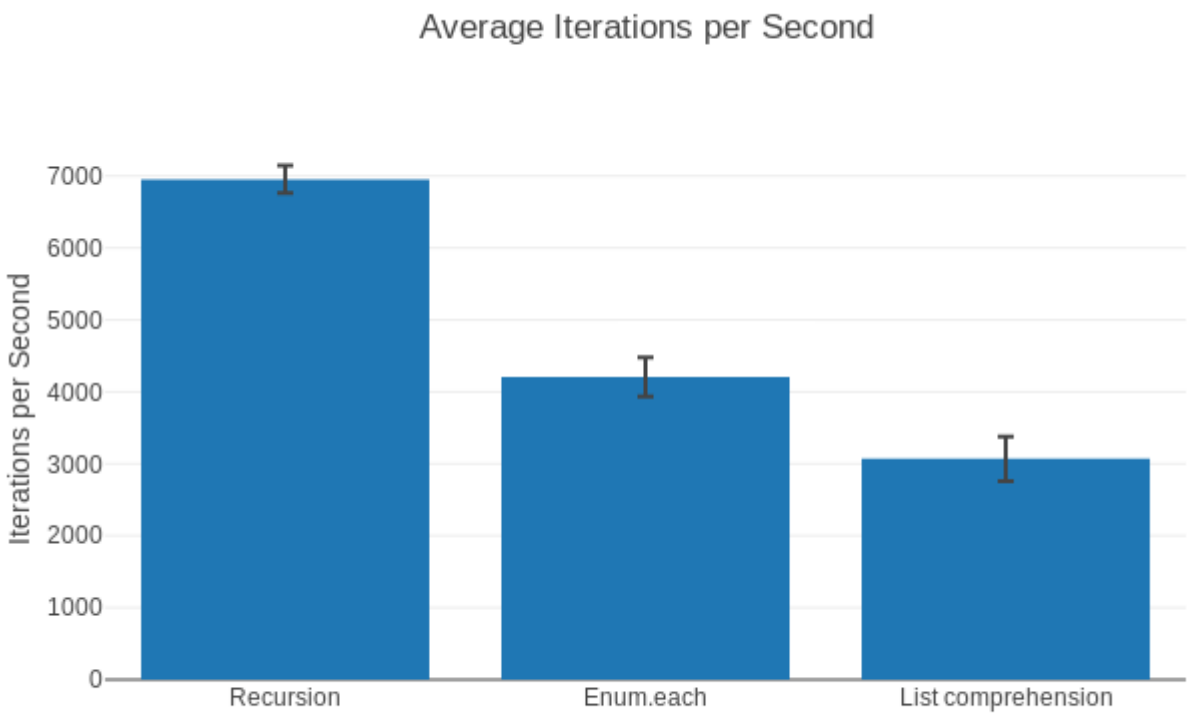
The End?


I HAVE NO
IDEA WHAT
I'M DOING

Numerous Failures!

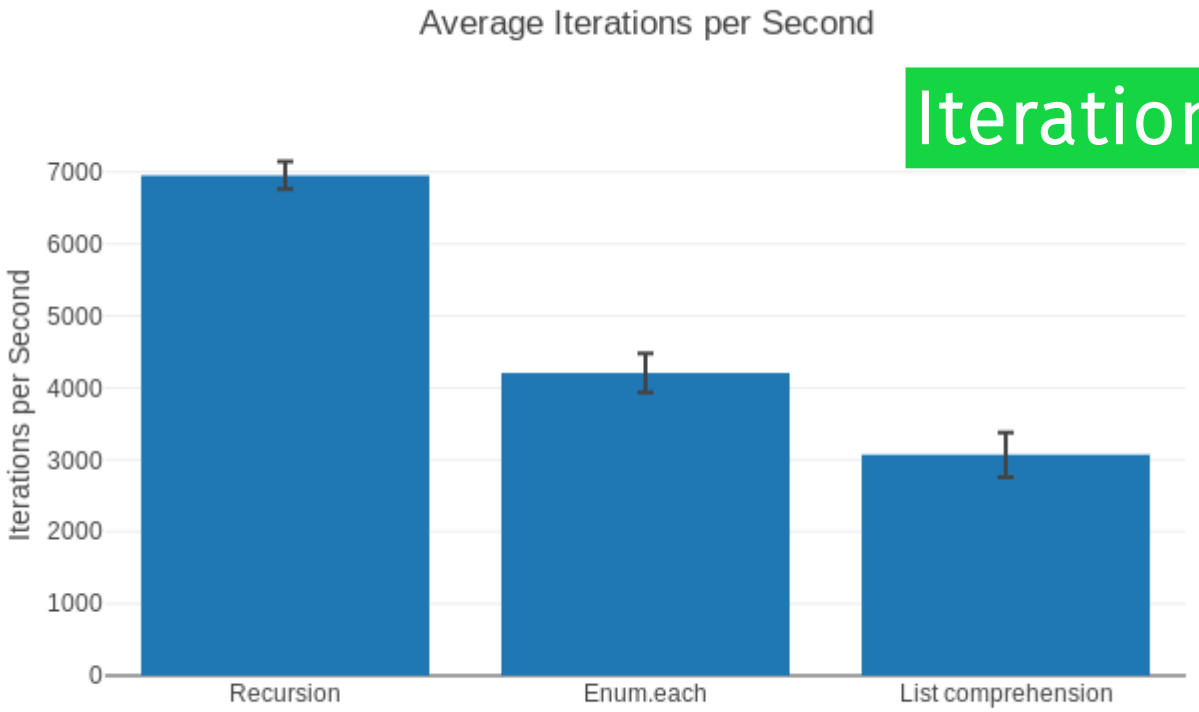
- Way too few samples
- Realistic data/multiple inputs?
- No warmup
- Non production environment
- Does creating the list matter?
- Is repeating really the bottle neck?
- Repeatability?
- Setup information
- Running on battery
- Lots of applications running

```elixir
n      = 10_000
range = 1..n
list  = Enum.to_list range
fun   = fn -> 0 end

Benchee.run %{
  "Enum.each" =>
    fn -> Enum.each(list, fn(_) -> fun.() end) end,
  "List comprehension" =>
    fn -> for _ <- list, do: fun.() end,
  "Recursion" =>
    fn -> RepeatN.repeat_n(fun, n) end
}
```




```
Name                           ips        average  deviation         median         99th %
Recursion                   6.95 K      143.80 µs     ±2.76%      142 µs      156 µs
Enum.each                   4.21 K      237.70 µs     ±6.47%      230 µs      278 µs
List comprehension          3.07 K      325.88 µs    ±10.02%      333 µs      373 µs

Comparison:
Recursion                   6.95 K
Enum.each                   4.21 K - 1.65x slower
List comprehension          3.07 K - 2.27x slower
```

Iterations per Second



Name	ips	average	deviation	median	99th %
Recursion	6.95 K	143.80 µs	±2.76%	142 µs	156 µs
Enum.each	4.21 K	237.70 µs	±6.47%	230 µs	278 µs
List comprehension	3.07 K	325.88 µs	±10.02%	333 µs	373 µs

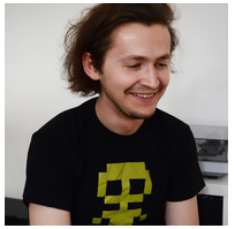
Comparison:

Recursion	6.95 K	
Enum.each	4.21 K	1.65x slower
List comprehension	3.07 K	2.27x slower

Stop guessing and Start Measuring
Benchmarking in Practice

Tobias Pfeiffer
@PragTob
pragtob.info
github.com/PragTob/benchee

LIEFERY

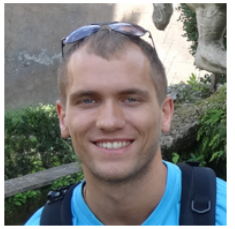


ANDREY KUZMIN

maintainer of WebGL in Elm, organizer of Elm Berlin meetup

BARBARA CHASSOUL

Core member of nonsense, working on Luerl and Luajit



TOMASZ HEIMOWSKI

Functional Programming Adept

MAXIMILIAN ALGEHED

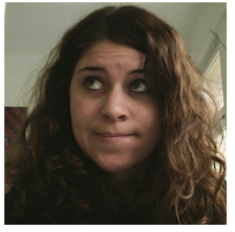
Quick, where is the spec?!

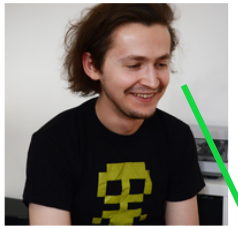
TOBIAS PFEIFFER

Benchmarker by Passion

ADAM LINDBERG

Peer Stritzinger GmbH



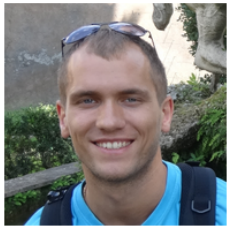


ANDREY KUZMIN

maintainer of WebGL in Elm, organizer of Elm Berlin meetup

BARBARA CHASSOUL

Core member of nonsense, working on Luerl and Luajit



TOMASZ HEIMOWSKI

Functional Programming Adept

MAXIMILIAN ALGEHED

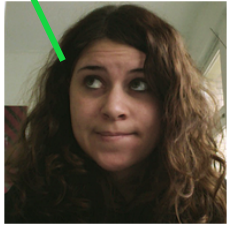
Quick, where is the spec?!

TOBIAS PFEIFFER

Benchmarker by Passion

ADAM LINDBERG

Peer Stritzinger GmbH



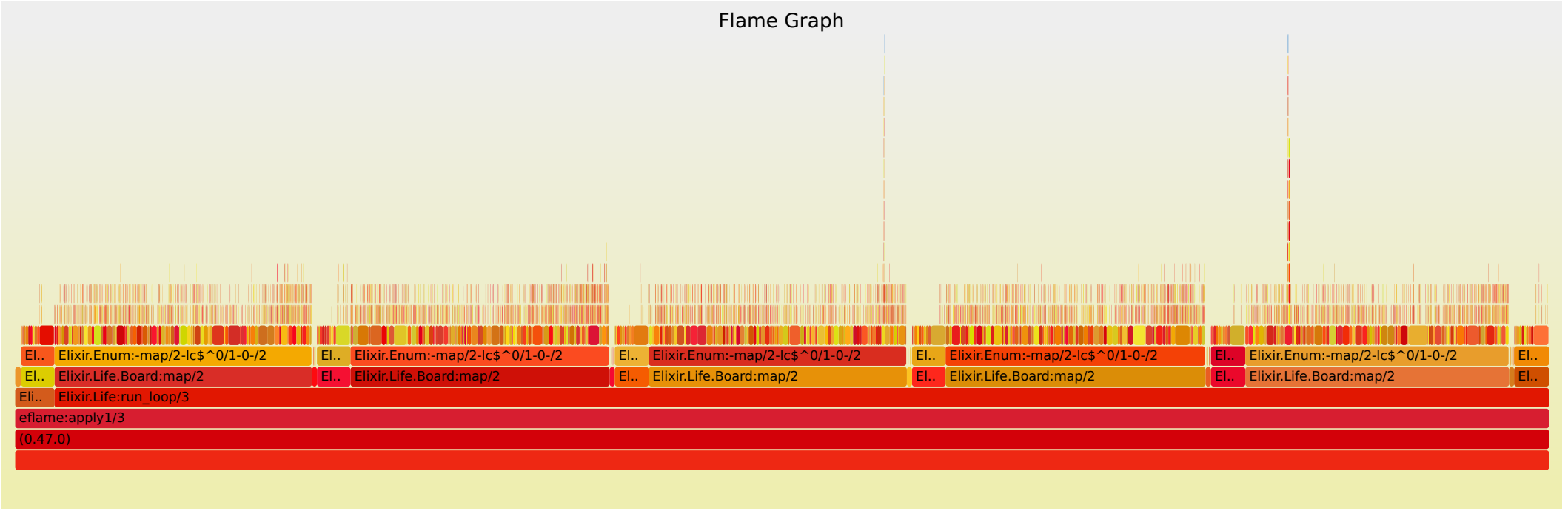
Concept vs Tool Usage

What’s Benchmarking?

Profiling

VS

Benchmarking



http://learningelixir.joekain.com/profiling-elixir-2/

What to benchmark?

What to measure?

Runtime?

Memory?

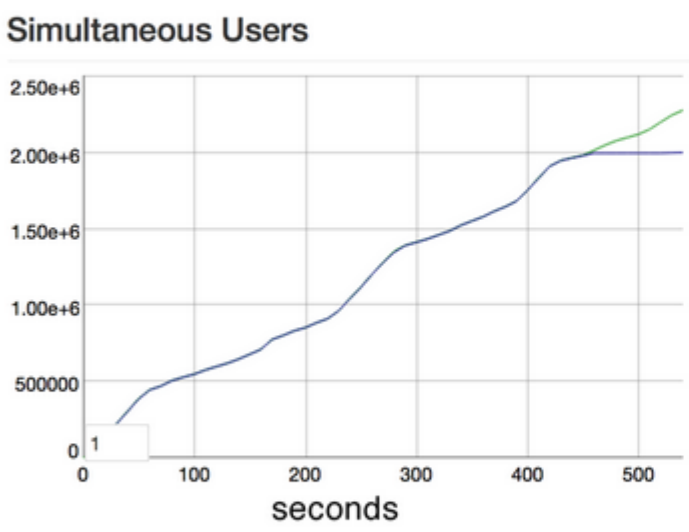
Throughput?

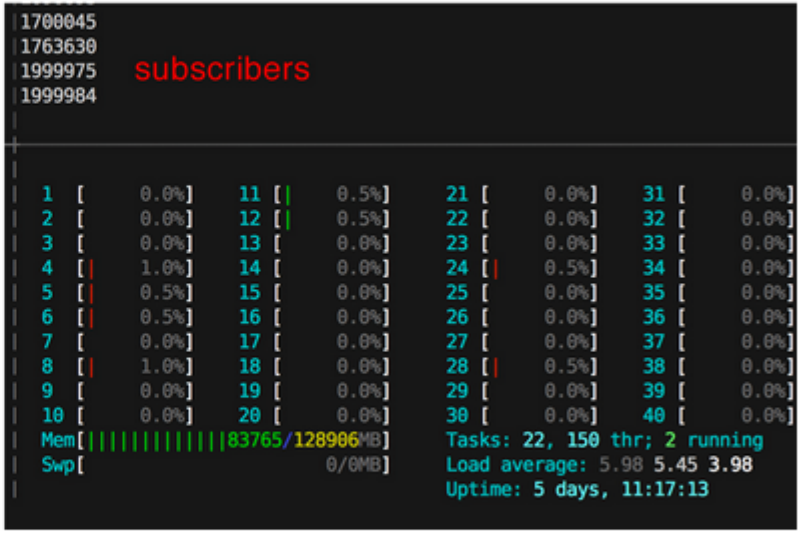
Custom?

The famous post

The Road to 2 Million Websocket Connections in Phoenix

By Gary Rennie · about a year ago · v1.0.0





If you have been paying attention on Twitter recently, you have likely seen some increasing numbers regarding the number of simultaneous connections the Phoenix web framework can handle. This post documents some of the techniques used to perform the benchmarks.

What to measure?

Runtime!

Memory?

Throughput?

Custom?

How long will this take?

Enum.sort/1 performance

Name	Iterations per Second	Average	Deviation	median	minimum	maximum	Sample size
100k	42.13	23.74 ms	±5.20%	23.42 ms	21.94 ms	27.09 ms	211
10k	601.90	1.66 ms	±6.10%	1.60 ms	1.59 ms	2.62 ms	3006
1M	2.75	363.81 ms	±7.54%	352.32 ms	334.70 ms	416.30 ms	14
5M	0.50	1994.18 ms	±1.37%	1986.60 ms	1965.13 ms	2030.82 ms	3

Enum.sort/1 performance

Name	Iterations per Second	Average	Deviation	median	minimum	maximum	Sample size
100k	42.13	23.74 ms	±5.20%	23.42 ms	21.94 ms	27.09 ms	211
10k	601.90	1.66 ms	±6.10%	1.60 ms	1.59 ms	2.62 ms	3006
1M	2.75	363.81 ms	±7.54%	352.32 ms	334.70 ms	416.30 ms	14
5M	0.50	1994.18 ms	±1.37%	1986.60 ms	1965.13 ms	2030.82 ms	3

Did we make it faster?

What's fastest?

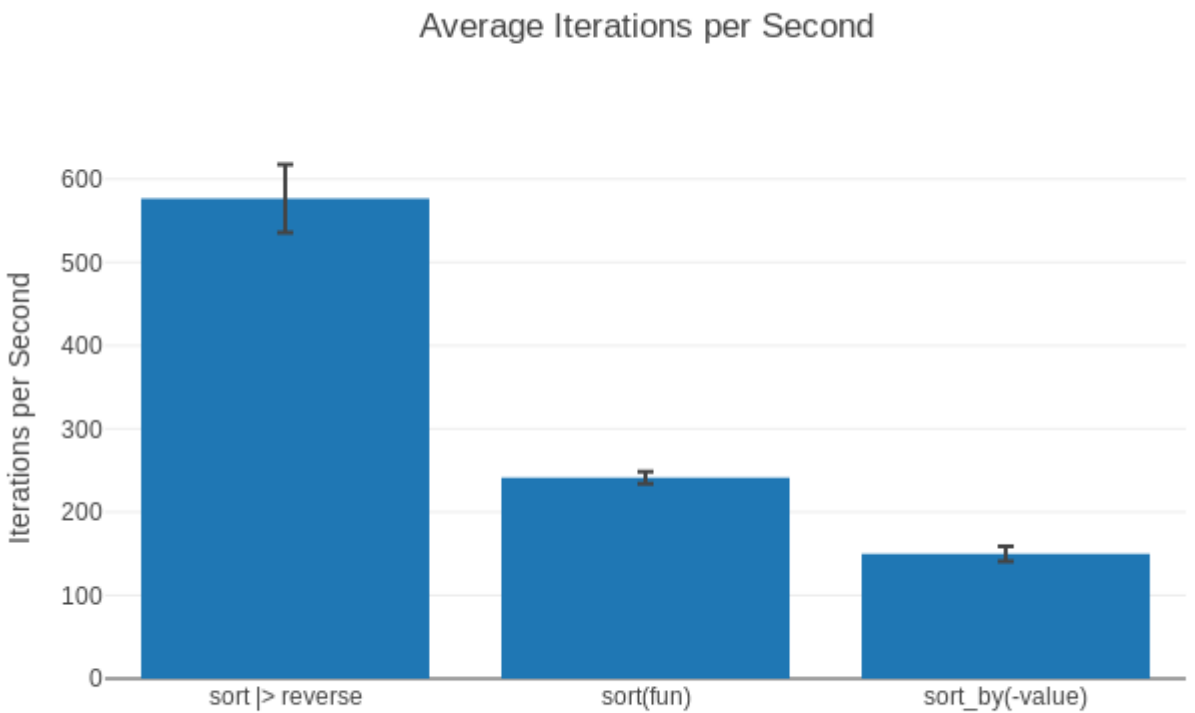
What's the fastest way to sort a list of numbers largest to smallest?

```elixir
list = 1..10_000 |> Enum.to_list |> Enum.shuffle

Benchee.run %{
  "sort(fun)" =>
    fn -> Enum.sort(list, &(&1 > &2)) end,
  "sort |> reverse" =>
    fn -> list |> Enum.sort |> Enum.reverse end,
  "sort_by(-value)" =>
    fn -> Enum.sort_by(list, fn(val) -> -val end) end
}
```

```elixir
list = 1..10_000 |> Enum.to_list |> Enum.shuffle

Benchee.run %{
  "sort(fun)" =>
    fn -> Enum.sort(list, &(&1 > &2)) end,
  "sort |> reverse" =>
    fn -> list |> Enum.sort |> Enum.reverse end,
  "sort_by(-value)" =>
    fn -> Enum.sort_by(list, fn(val) -> -val end) end
}
```

```elixir
list = 1..10_000 |> Enum.to_list |> Enum.shuffle

Benchee.run %{
  "sort(fun)" =>
    fn -> Enum.sort(list, &(&1 > &2)) end,
  "sort |> reverse" =>
    fn -> list |> Enum.sort |> Enum.reverse end,
  "sort_by(-value)" =>
    fn -> Enum.sort_by(list, fn(val) -> -val end) end
}
```

```elixir
list = 1..10_000 |> Enum.to_list |> Enum.shuffle

Benchee.run %{
  "sort(fun)" =>
    fn -> Enum.sort(list, &(&1 > &2)) end,
  "sort |> reverse" =>
    fn -> list |> Enum.sort |> Enum.reverse end,
  "sort_by(-value)" =>
    fn -> Enum.sort_by(list, fn(val) -> -val end) end
}
```




```
Name                           ips        average  deviation         median         99th %
sort |> reverse            576.52        1.74 ms     ±7.11%        1.70 ms        2.25 ms
sort(fun)                  241.21        4.15 ms     ±3.04%        4.15 ms        4.60 ms
sort_by(-value)            149.96        6.67 ms     ±6.06%        6.50 ms        7.83 ms

Comparison:
sort |> reverse            576.52
sort(fun)                  241.21 - 2.39x slower
sort_by(-value)            149.96 - 3.84x slower
```

“Isn’t that the
root of all evil?”

"More likely, not reading the sources is the source of all evil."

Me, just now

Yup it's there

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

Donald Knuth, 1974

(Computing Surveys, Vol 6, No 4, December 1974)

The very next sentence

"Yet we should not pass up our opportunities in that critical 3%."

Donald Knuth, 1974

(Computing Surveys, Vol 6, No 4, December 1974)

Prior Paragraph

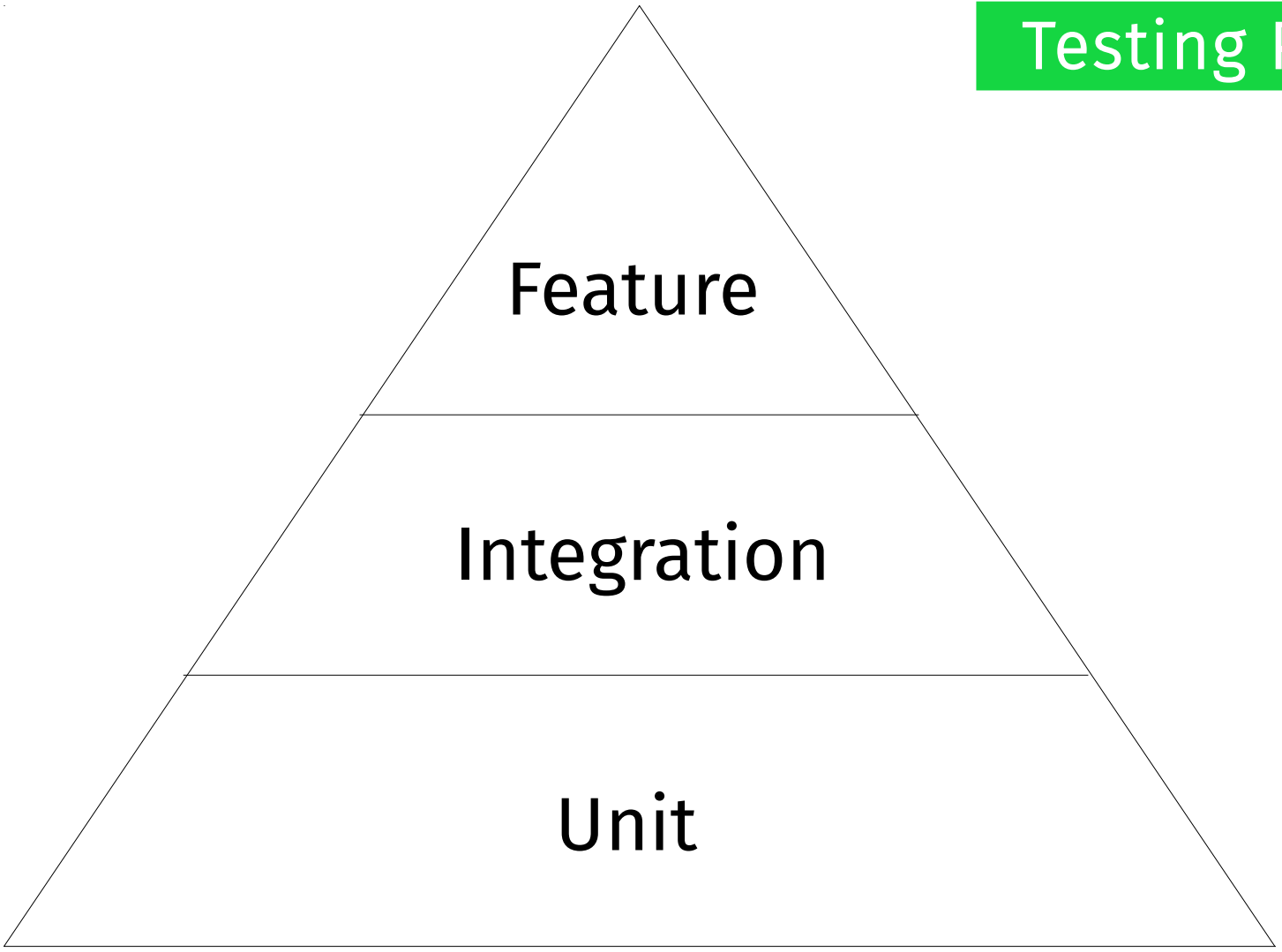
"(...) a 12 % improvement, easily obtained, is never considered marginal"

Donald Knuth, 1974

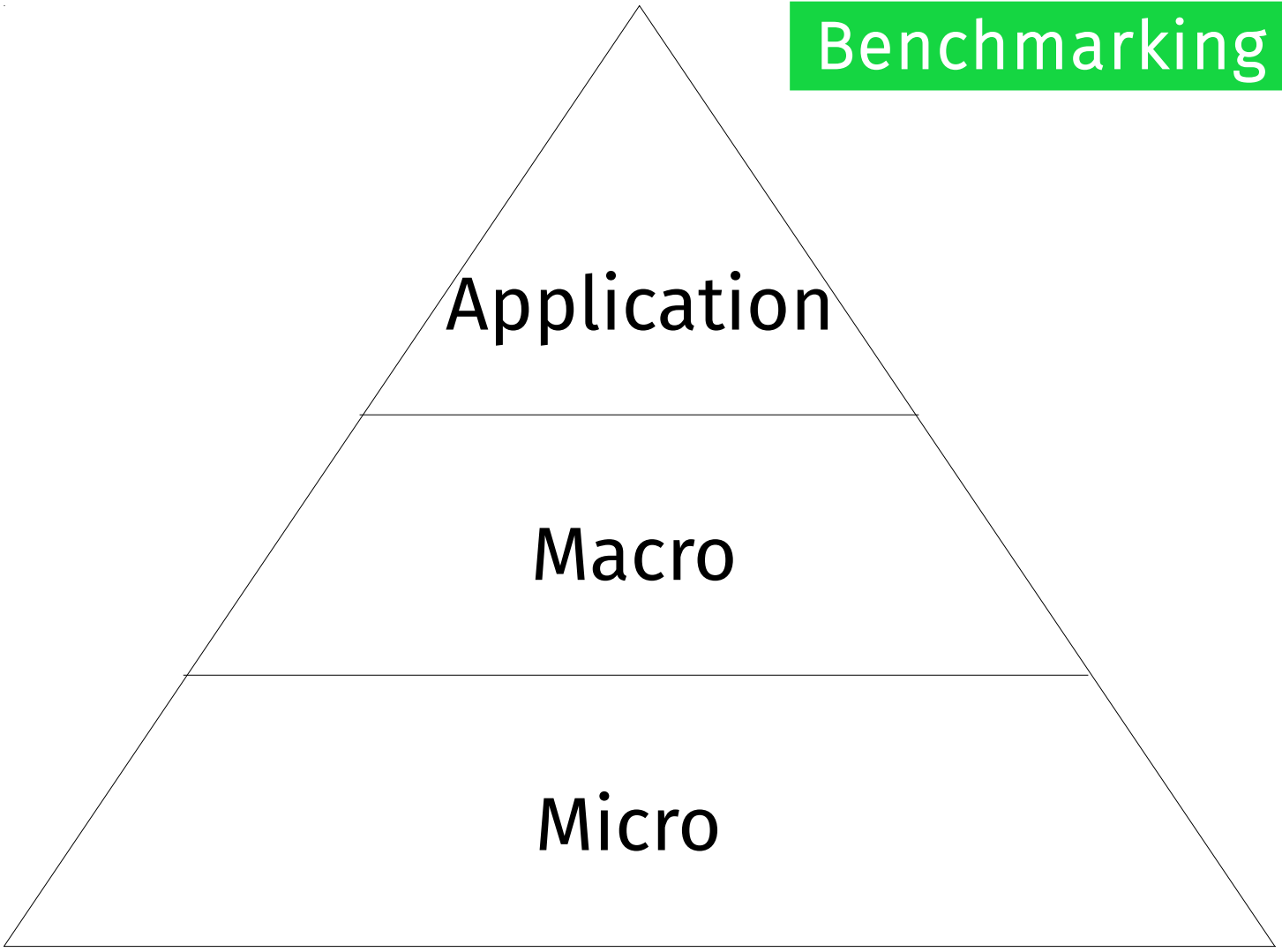
(Computing Surveys, Vol 6, No 4, December 1974)

Different types of benchmarks

Testing Pyramid



Benchmarking Pyramid

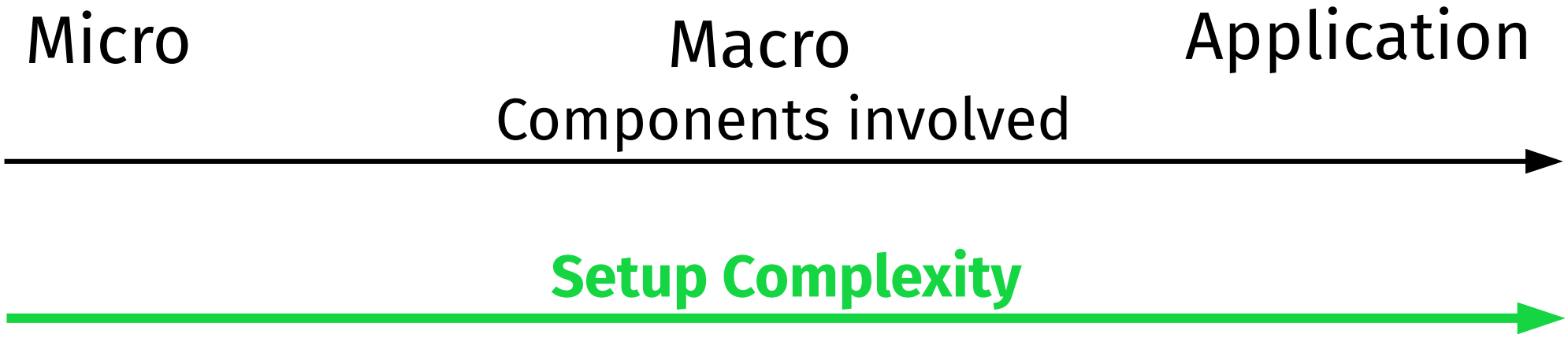


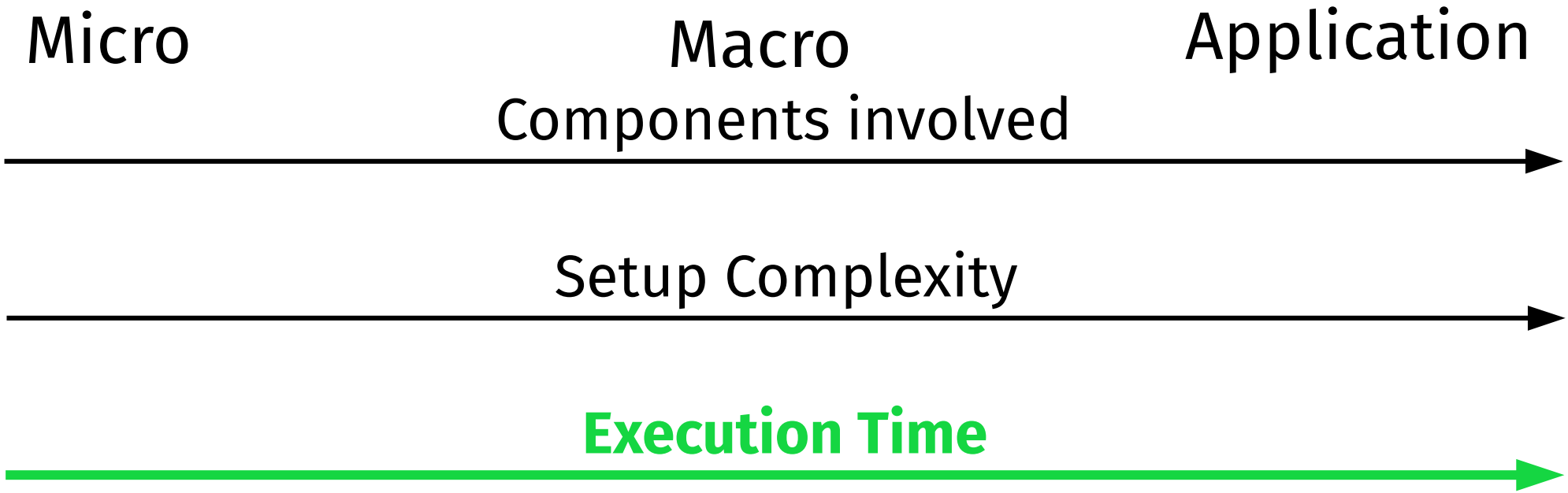
Micro

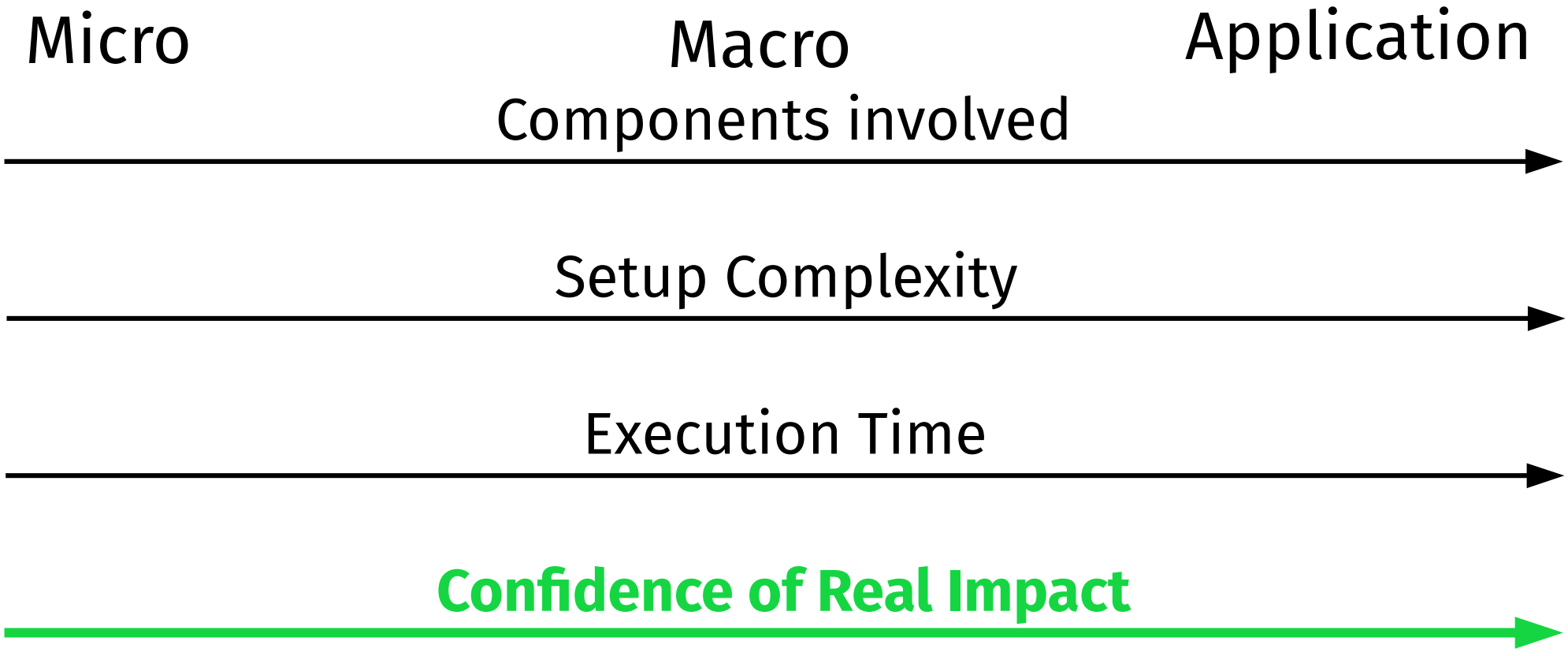
Macro

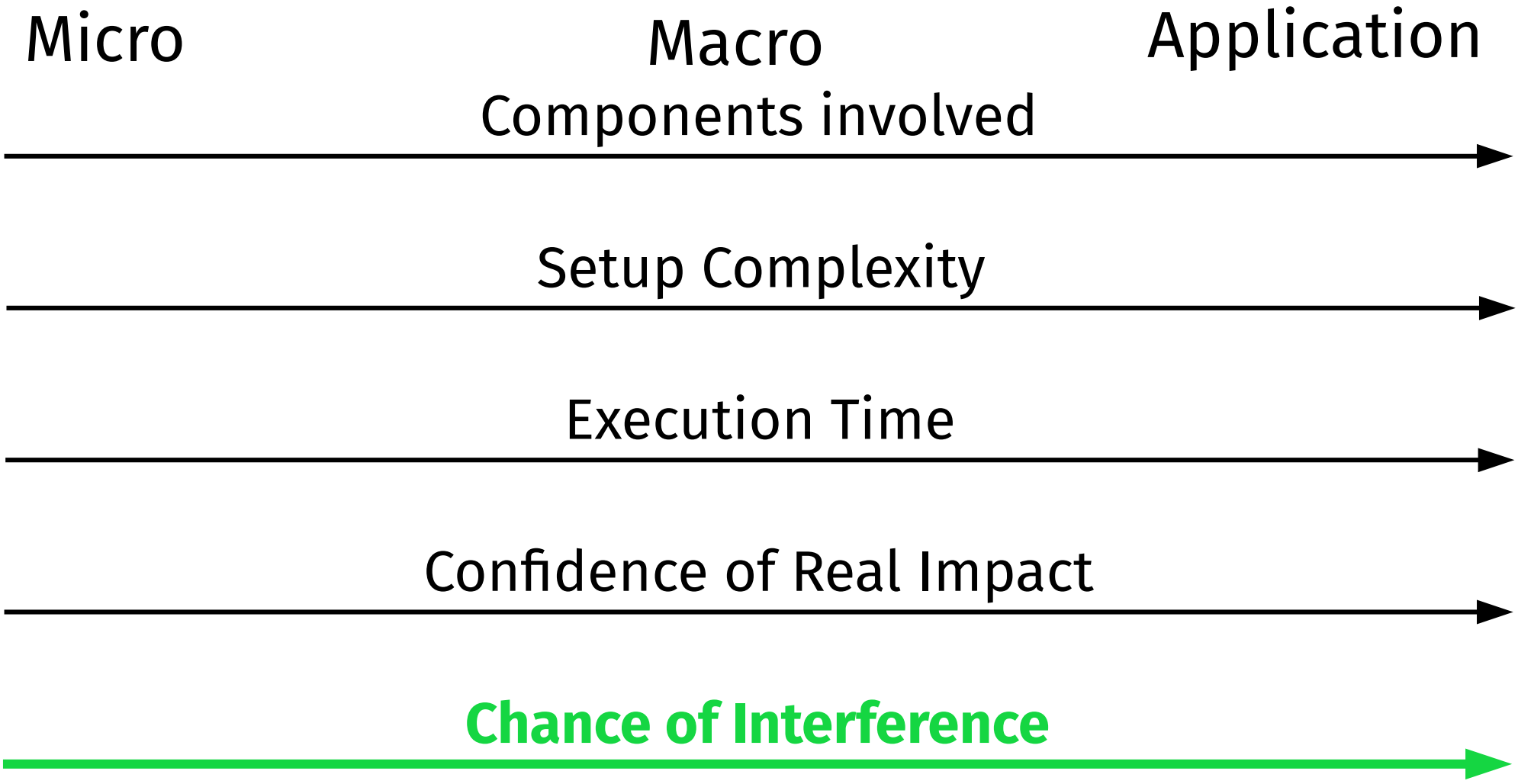
Application

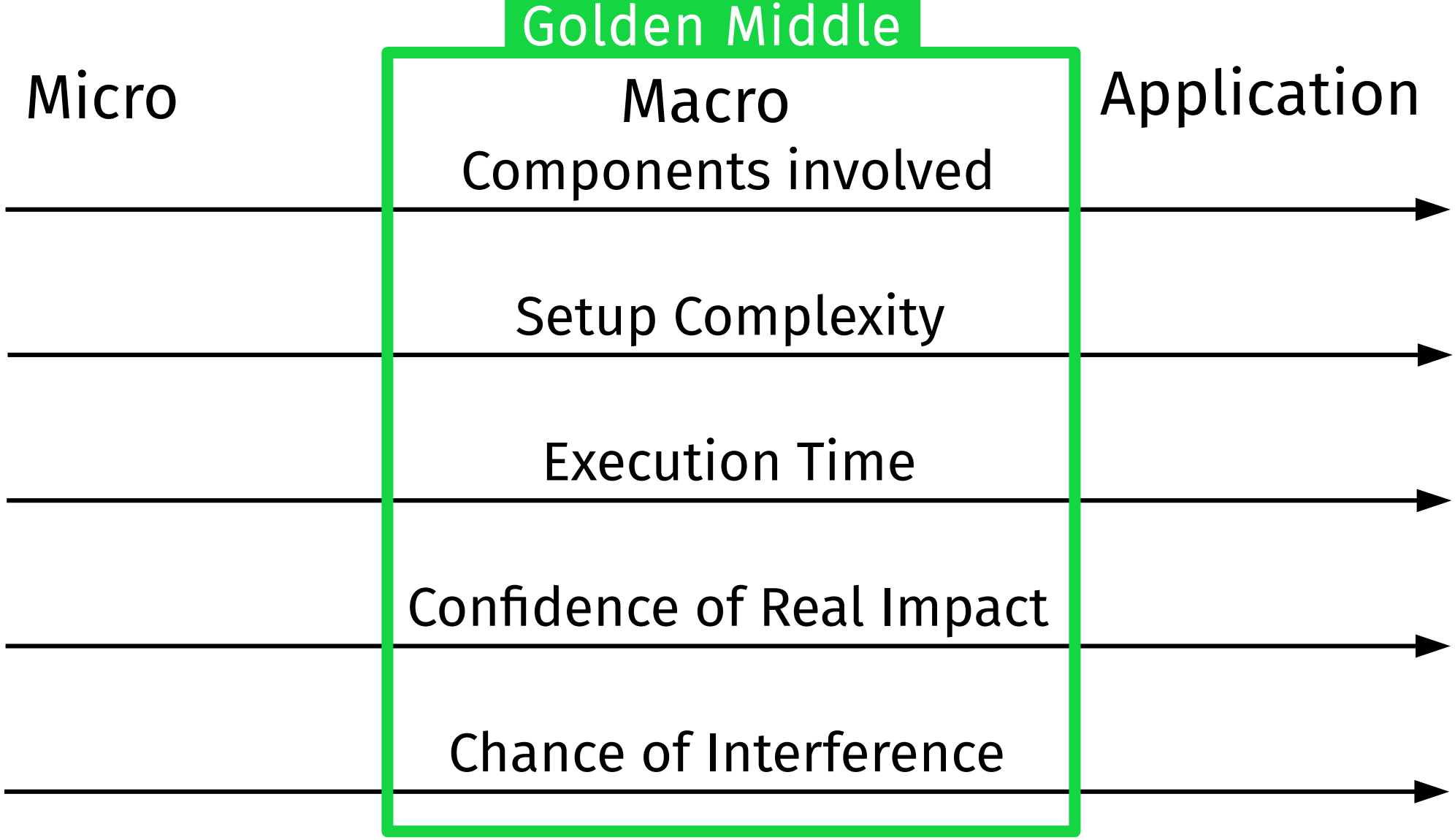

Micro
Macro
Application
Components involved


Micro
Macro
Application
Components involved
Setup Complexity


Micro
Macro
Application
Components involved
Setup Complexity
Execution Time


Micro
Macro
Application
Components involved
Setup Complexity
Execution Time
Confidence of Real Impact


Micro
Macro
Application
Components involved
Setup Complexity
Execution Time
Confidence of Real Impact
Chance of Interference


Golden Middle
Micro
Macro
Application
Components involved
Setup Complexity
Execution Time
Confidence of Real Impact
Chance of Interference

Good Benchmarking

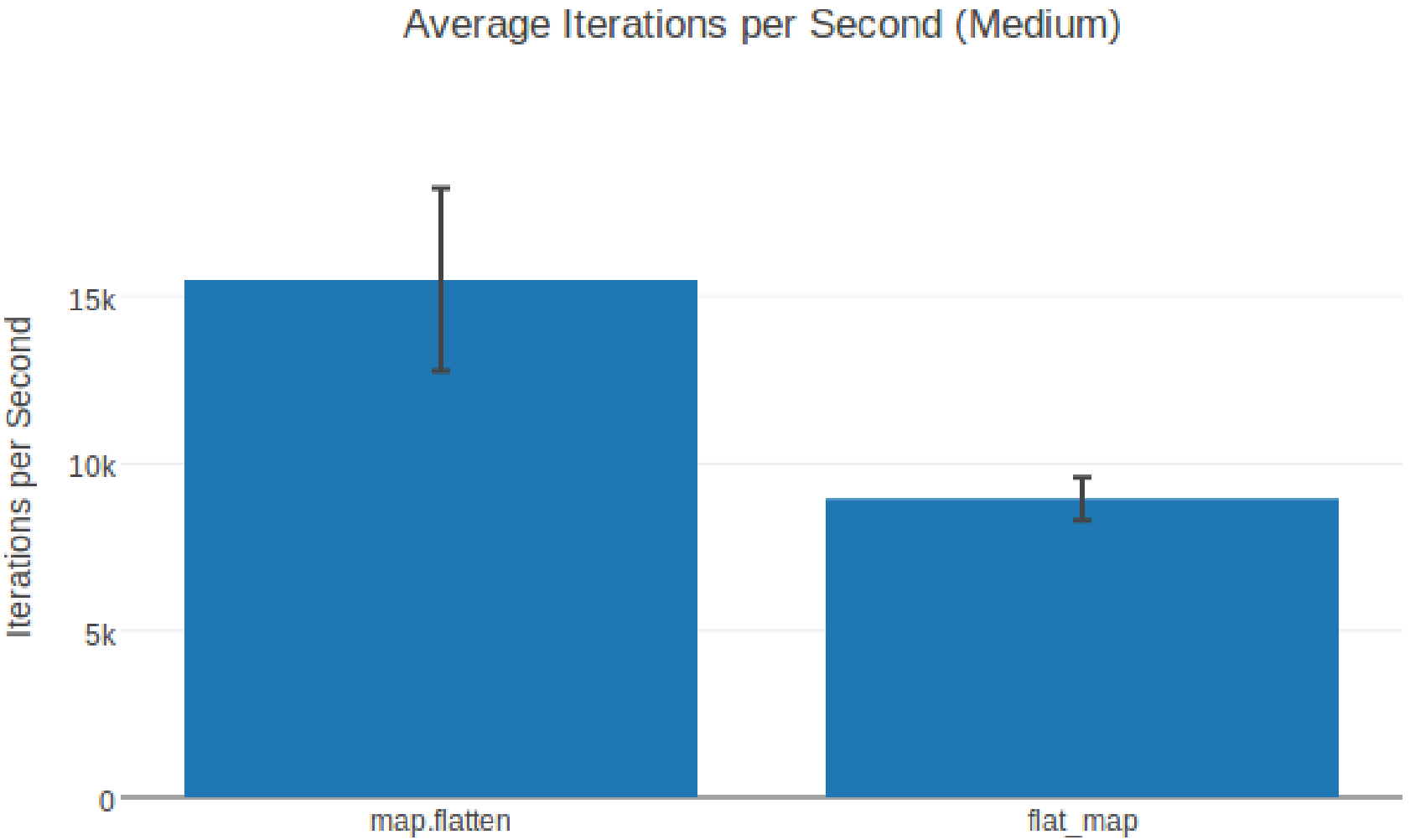
What are you benchmarking for?

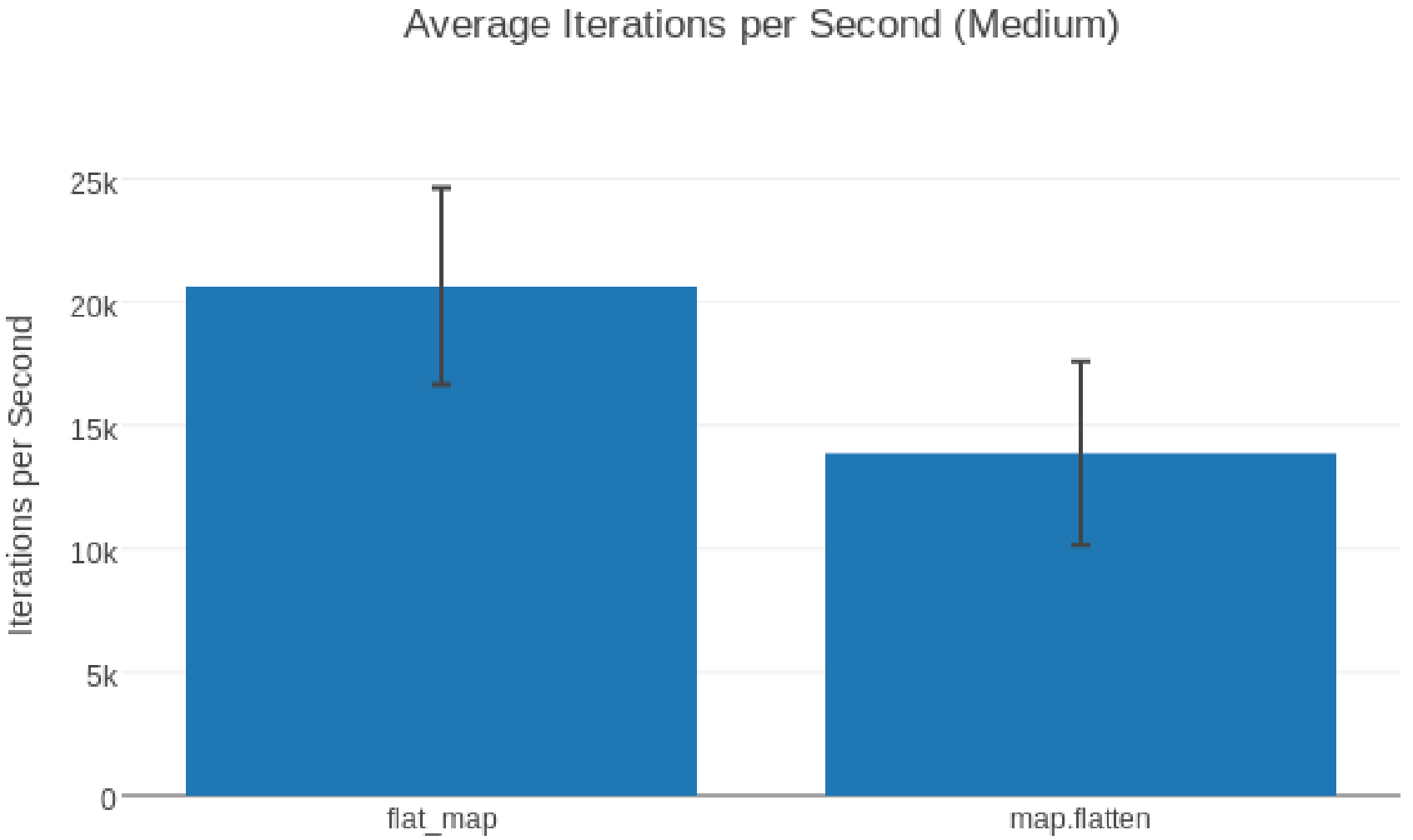
System Specification

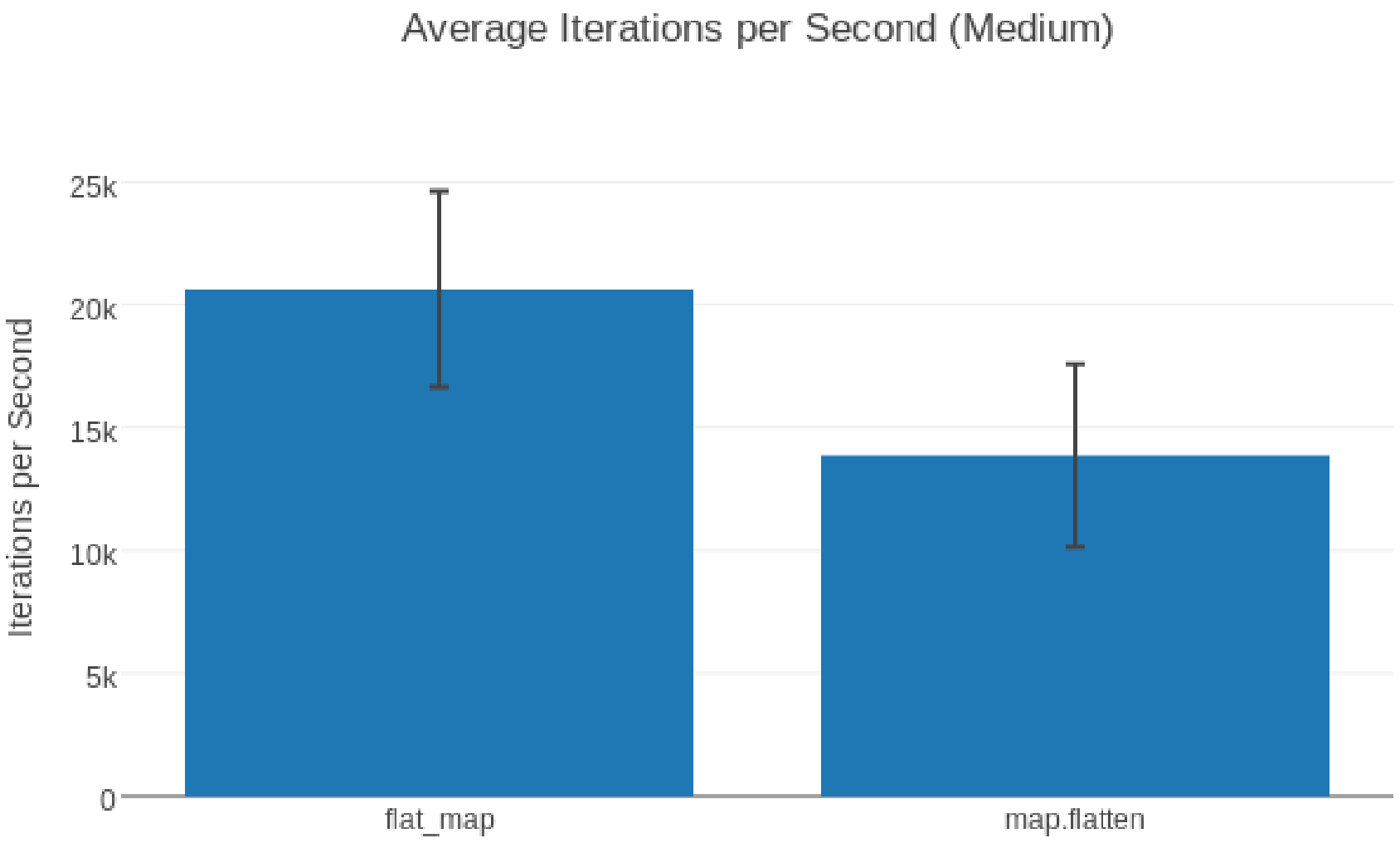
- Elixir 1.6.1
- Erlang 20.2
- i5-7200U – 2 x 2.5GHz (Up to 3.10GHz)
- 8GB RAM
- Linux Mint 18.3 - 64 bit (Ubuntu 16.04 base)

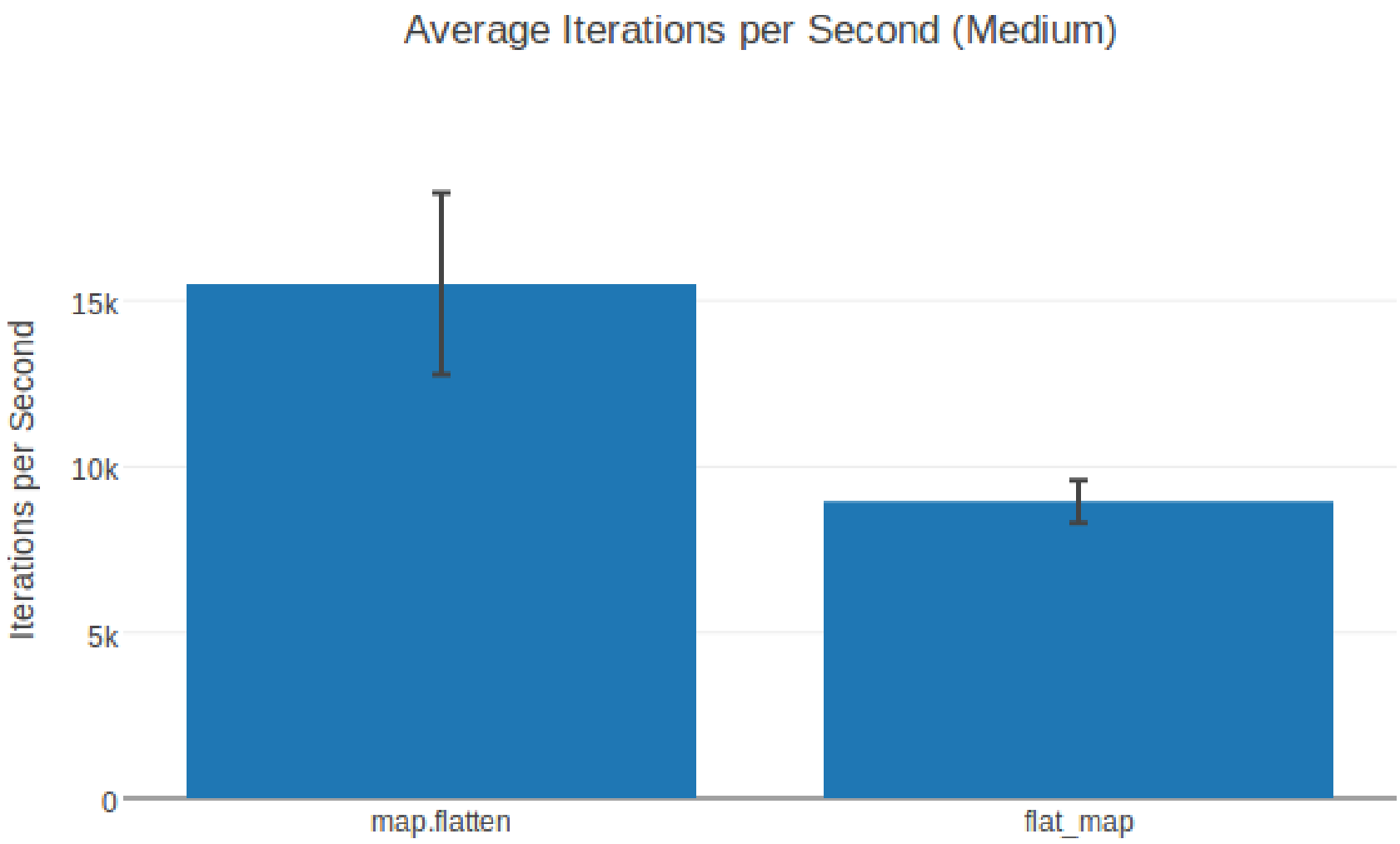
System Specification

```
tobi@comfy elixir_playground $ mix run bench/something.exs
Operating System: Linux
CPU Information: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
Number of Available Cores: 4
Available memory: 7.67 GB
Elixir 1.6.1
Erlang 20.2
Benchmark suite executing with the following configuration:
warmup: 2 s
time: 5 s
parallel: 1
inputs: none specified
Estimated total run time: 21 s
```


Average Iterations per Second (Medium)
Iterations per Second
15k
10k
5k
0
map.flatten
flat_map


Average Iterations per Second (Medium)
Iterations per Second
25k
20k
15k
10k
5k
0
flat_map
map.flatten


Average Iterations per Second (Medium)
1.4!
Iterations per Second
25k
20k
15k
10k
5k
0
flat_map
map.flatten


Average Iterations per Second (Medium)
1.3
Iterations per Second
15k
10k
5k
0
map.flatten
flat_map

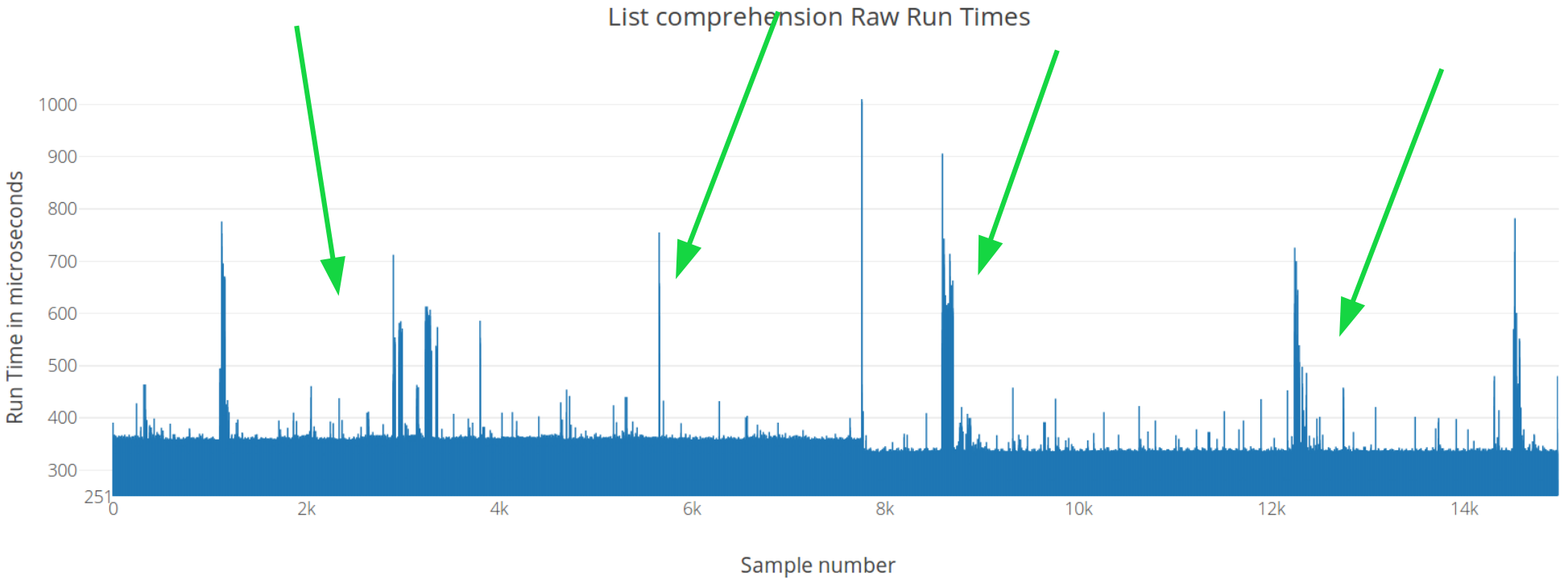

Correct & Meaningful Setup


Interference free Environment

Logging & Friends

```
[info] GET /
[debug] Processing by Rumbl.PageController.index/2
  Parameters: %{}
  Pipelines: [:browser]
[info] Sent 200 in 46ms
[info] GET /sessions/new
[debug] Processing by Rumbl.SessionController.new/2
  Parameters: %{}
  Pipelines: [:browser]
[info] Sent 200 in 5ms
[info] GET /users/new
[debug] Processing by Rumbl.UserController.new/2
  Parameters: %{}
  Pipelines: [:browser]
[info] Sent 200 in 7ms
[info] POST /users
[debug] Processing by Rumbl.UserController.create/2
  Parameters: %{"_csrf_token" => "NUEUdRMNAiBfIHEeNwZkfA05PgAOJgAAf0ACXJqCjl7YojW+trdjdg==", "_utf8" => "✓", "user"
=> %{"name" => "asdasd", "password" => "[FILTERED]", "username" => "Homer"}}
  Pipelines: [:browser]
[debug] QUERY OK db=0.1ms
begin []
[debug] QUERY OK db=0.9ms
INSERT INTO "users" ("name","password_hash","username","inserted_at","updated_at") VALUES ($1,$2,$3,$4,$5) RETURNING
"id" ["asdasd", "$2b$12$.qY/kpo0Dec7vMK1ClJoC.Lw77c3oGllX7uieZILMlFh2hFpJ3F.C", "Homer", {{2016, 12, 2}, {14, 10, 28, 0}},
{{2016, 12, 2}, {14, 10, 28, 0}}]
```


Garbage Collection


List comprehension Raw Run Times
Run Time in microseconds
1000
900
800
700
600
500
400
300
251
0
2k
4k
6k
8k
10k
12k
14k
Sample number


Warmup

Probably not what you want

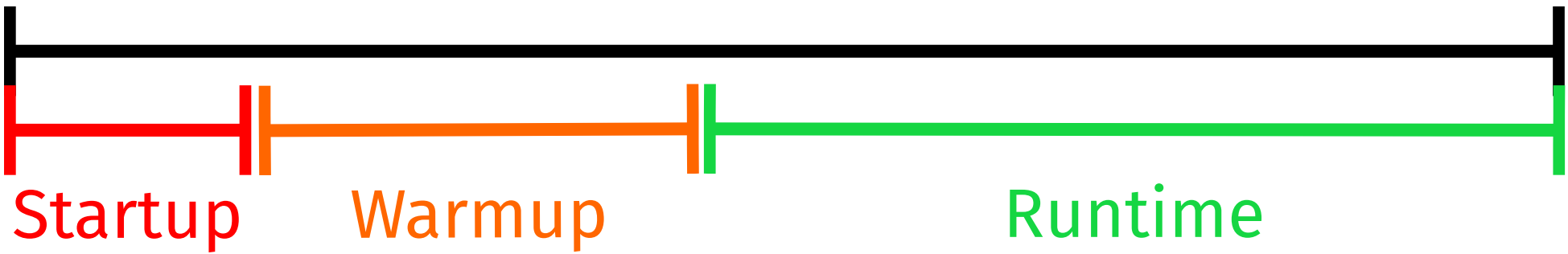
```
$ time my_program
```

What’s important for you?

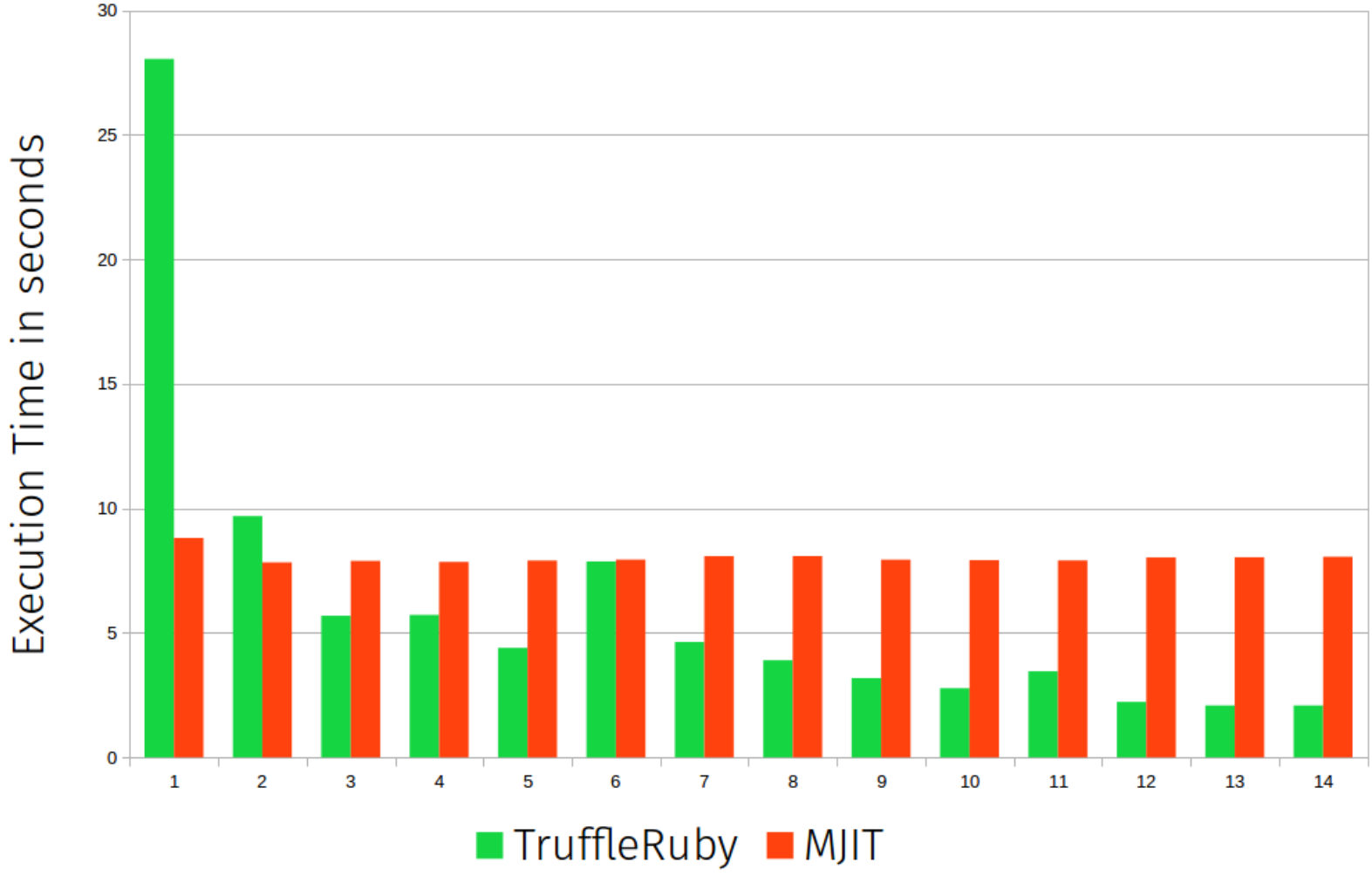
What’s important for you?

What's important for you?

What's important for you?



28s → 2s

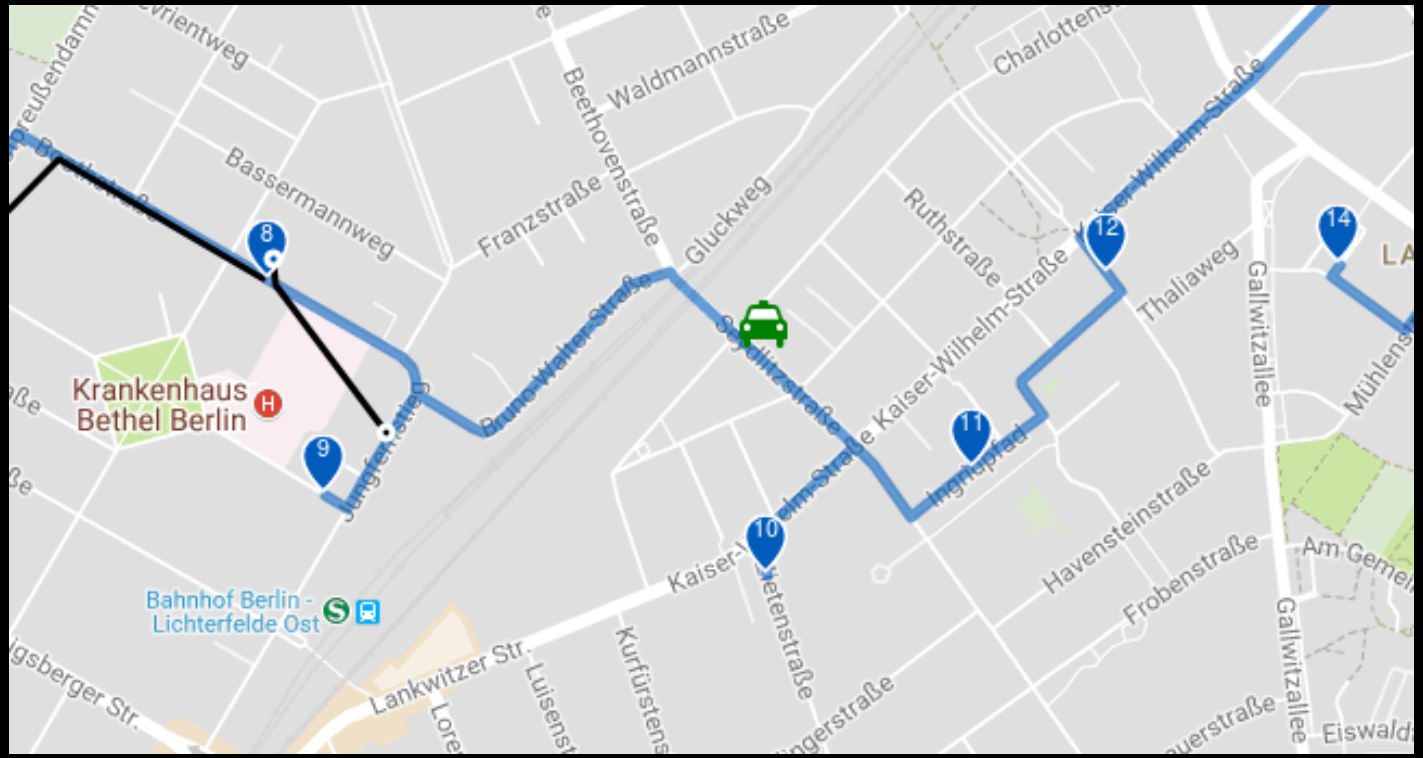

Execution Time in seconds
30
25
20
15
10
5
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
TruffleRuby
MJIT


Inputs matter!


Story Time

Boom


Boom
Elixir.DBConnection.ConnectionError


Krankenhaus Bethel Berlin
Bahnhof Berlin - Lichterfelde Ost
Bassermannweg
Franzstraße
Beethovenstraße
Waldmannstraße
Gluckweg
Bruno-Walter-Straße
Jungfernsteig
Schützenstraße
Kaiser-Wilhelm-Straße
Ruthstraße
Thaliaweg
Gallwitzallee
Havensteinstraße
Frobenstraße
Lankwitzer Str.
Kurfürstenstraße
Luisenstr.
Eiswaldt
8
9
10
11
12
14

```elixir
Benchee.run %{
  "DB View" => fn ->
    LatestCourierLocation
    |> CourierLocation.with_courier_ids(courier_id)
    |> Repo.one(timeout: :infinity)
  end,
  "with_courier_ids" => fn ->
    CourierLocation.with_courier_ids(courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one(timeout: :infinity)
  end,
  "full custom" => fn ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one(timeout: :infinity)
  end
}
```

Database View

```elixir
Benchee.run %{
  "DB View" => fn ->
    LatestCourierLocation
    |> CourierLocation.with_courier_ids(courier_id)
    |> Repo.one(timeout: :infinity)
  end,
  "with_courier_ids" => fn ->
    CourierLocation.with_courier_ids(courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one(timeout: :infinity)
  end,
  "full custom" => fn ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one(timeout: :infinity)
  end
}
```

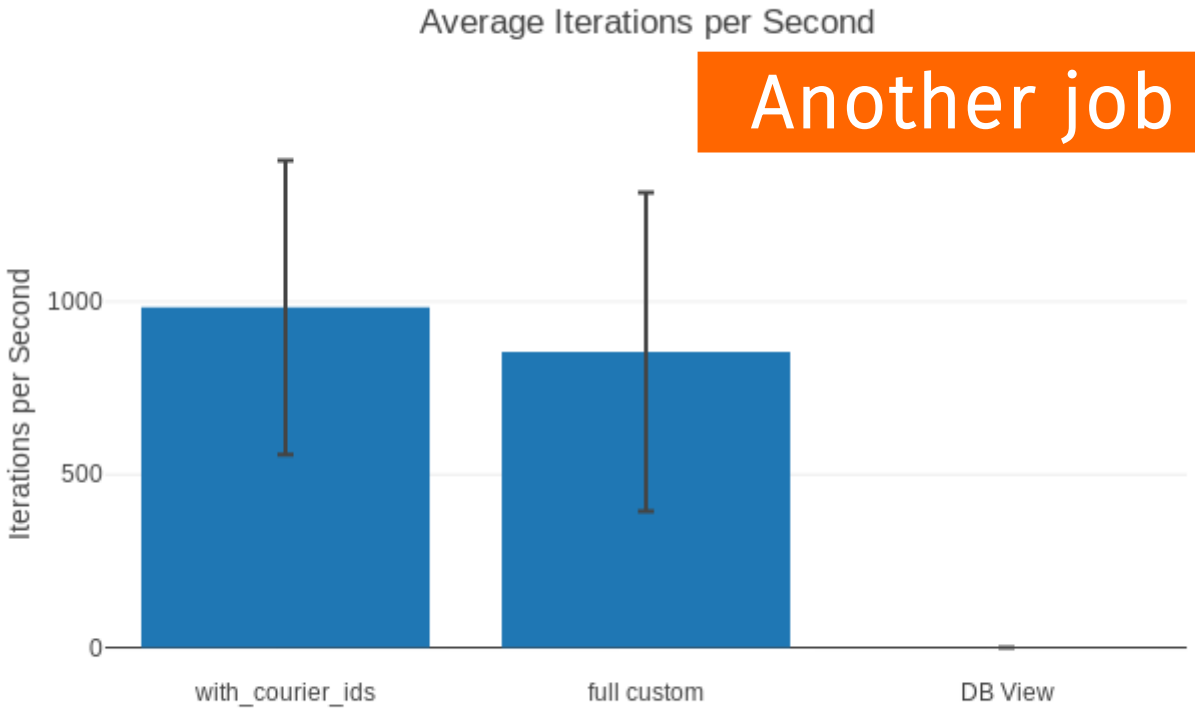
```elixir
Benchee.run %{
  "DB View" => fn ->
    LatestCourierLocation
    |> CourierLocation.with_courier_ids(courier_id)
    |> Repo.one(timeout: :infinity)
  end,
  "with_courier_ids" => fn ->
    CourierLocation.with_courier_ids(courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one(timeout: :infinity)
  end,
  "full custom" => fn ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one(timeout: :infinity)
  end
}
```

Only difference

```elixir
Benchee.run %{
  "DB View" => fn ->
    LatestCourierLocation
    |> CourierLocation.with_courier_ids(courier_id)
    |> Repo.one(timeout: :infinity)
  end,
  "with_courier_ids" => fn ->
    CourierLocation.with_courier_ids(courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one(timeout: :infinity)
  end,
  "full custom" => fn ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one(timeout: :infinity)
  end
}
```

Same

Another job well done?




```
Name                  ips        average  deviation         median         99th %
with_courier_ids   983.21        1.02 ms    ±43.22%        0.91 ms        3.19 ms
full custom        855.07        1.17 ms    ±53.86%        0.96 ms        4.25 ms
DB View              0.21     4704.70 ms     ±4.89%     4738.83 ms     4964.49 ms

Comparison:
with_courier_ids   983.21
full custom        855.07 - 1.15x slower
DB View              0.21 - 4625.70x slower
```

Another job well done?

Boom


Boom
Elixir.DBConnection.ConnectionError


Boom
Elixir.DBConnection.ConnectionError
Elixir.DBConnection.ConnectionError


Boom
Elixir.DBConnection.ConnectionError
Elixir.DBConnection.ConnectionError
Elixir.DBConnection.ConnectionError
Elixir.DBConnection.ConnectionError
Elixir.DBConnection.ConnectionError
Elixir.DBConnection.ConnectionError
Elixir.DBConnection.ConnectionError
Elixir.DBConnection.ConnectionError

Whaaaat?

Inputs to the rescue!

```
inputs = %{
  "Big 2.3 Million locations" => 3799,
  "No locations"              => 8901,
  "~200k locations"           => 4238,
  "~20k locations"            => 4201
}

Benchee.run %{
  ...
  "full custom" => fn(courier_id) ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one(timeout: :infinity)
  end
}, inputs: inputs, time: 25, warmup: 5
```

Inputs to the rescue!

```
inputs = %{
  "Big 2.3 Million locations" => 3799,
  "No locations"              => 8901,
  "~200k locations"           => 4238,
  "~20k locations"            => 4201
}

Benchee.run %{
  ...
  "full custom" => fn(courier_id) ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one(timeout: :infinity)
  end
}, inputs: inputs, time: 25, warmup: 5
```

Inputs to the rescue!

```
inputs = %{
  "Big 2.3 Million locations" => 3799,
  "No locations"              => 8901,
  "~200k locations"           => 4238,
  "~20k locations"            => 4201
}

Benchee.run %{
  ...
  "full custom" => fn(courier_id) ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one(timeout: :infinity)
  end
}, inputs: inputs, time: 25, warmup: 5
```

Inputs to the rescue!

```
inputs = %{
  "Big 2.3 Million locations" => 3799,
  "No locations"              => 8901,
  "~200k locations"           => 4238,
  "~20k locations"            => 4201
}

Benchee.run %{
  ...
  "full custom" => fn(courier_id) ->
    CourierLocation
    |> Ecto.Query.where(courier_id: ^courier_id)
    |> Ecto.Query.order_by(desc: :time)
    |> Ecto.Query.limit(1)
    |> Repo.one(timeout: :infinity)
  end
}, inputs: inputs, time: 25, warmup: 5
```

```
##### With input Big 2.5 million locations #####
with_courier_ids  937.18
full custom       843.24 - 1.11x slower
DB View             0.22 - 4261.57x slower

##### With input ~200k locations #####
DB View             3.57
with_courier_ids   0.109 - 32.84x slower
full custom       0.0978 - 36.53x slower

##### With input ~20k locations #####
DB View             31.73
with_courier_ids   0.104 - 305.37x slower
full custom       0.0897 - 353.61x slower

##### With input No locations #####
Comparison:
DB View           1885.48
with_courier_ids   0.0522 - 36123.13x slower
full custom        0.0505 - 37367.23x slower
```

```
##### With input Big 2.5 million locations #####
with_courier_ids  937.18
full custom       843.24 - 1.11x slower
DB View             0.22 - 4261.57x slower

##### With input ~200k locations #####
DB View             3.57
with_courier_ids   0.109 - 32.84x slower
full custom       0.0978 - 36.53x slower

##### With input ~20k locations #####
DB View             31.73
with_courier_ids   0.104 - 305.37x slower
full custom       0.0897 - 353.61x slower

##### With input No locations #####
Comparison:
DB View           1885.48
with_courier_ids   0.0522 - 36123.13x slower
full custom        0.0505 - 37367.23x slower
```

woops

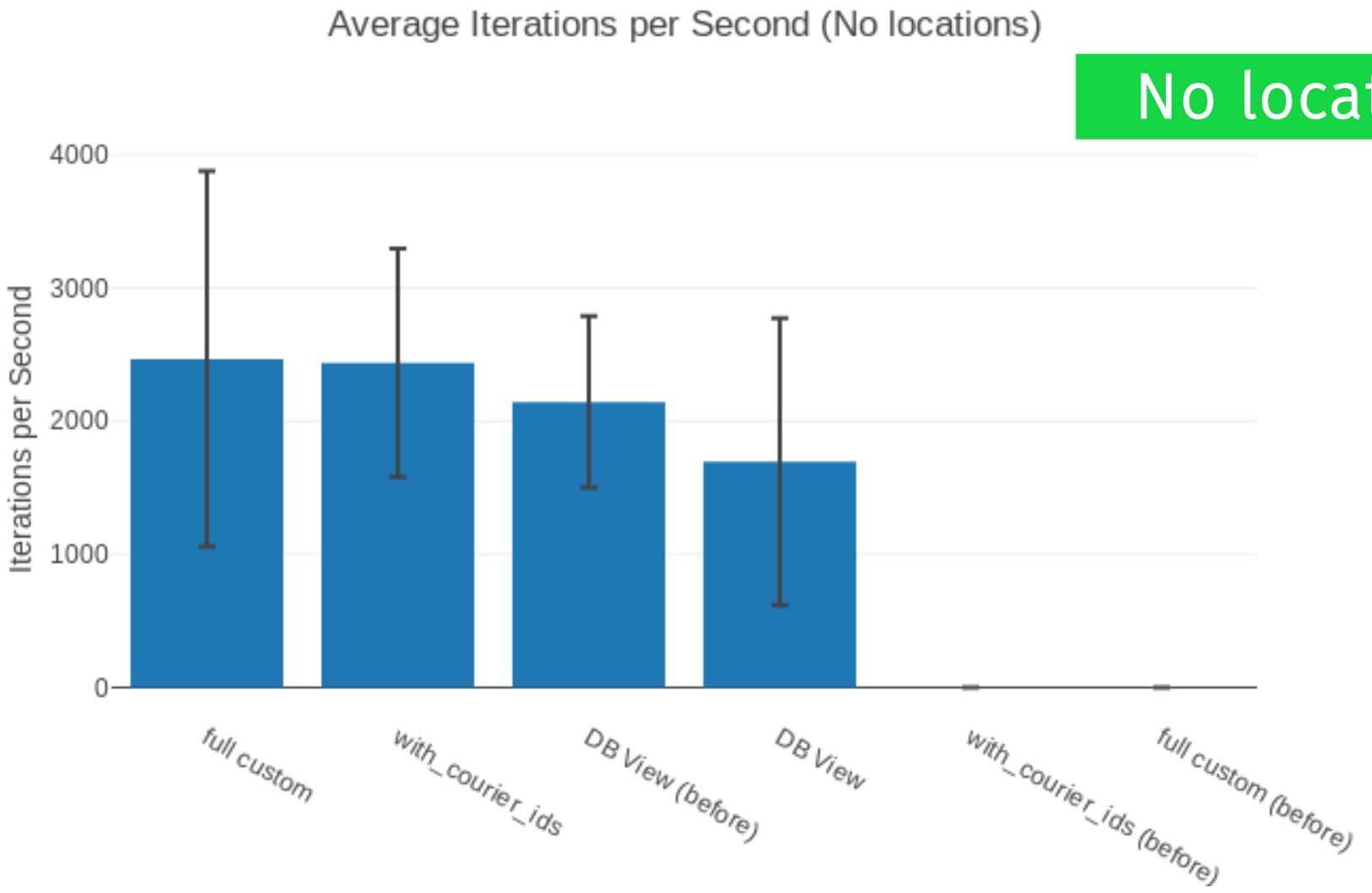
```
EXPLAIN ANALYZE
  SELECT c0."id",
         c0."courier_id",
         c0."location",
         c0."time",
         c0."accuracy",
         c0."inserted_at",
         c0."updated_at"
    FROM "courier_locations" AS c0
   WHERE (c0."courier_id" = 3799)
   ORDER BY c0."time" DESC LIMIT 1;

                                  QUERY PLAN
-------------------------------------------------------------------------
 Limit  (cost=0.43..0.83 rows=1 width=72)
        (actual time=1.840..1.841 rows=1 loops=1)
    ->  Index Scan Backward using courier_locations_time_index on
         courier_locations c0
        (cost=0.43..932600.17 rows=2386932 width=72)
        actual time=1.837..1.837 rows=1 loops=1)
          Filter: (courier_id = 3799)
          Rows Removed by Filter: 1371
 Planning time: 0.190 ms
 Execution time: 1.894 ms
(6 rows)
```

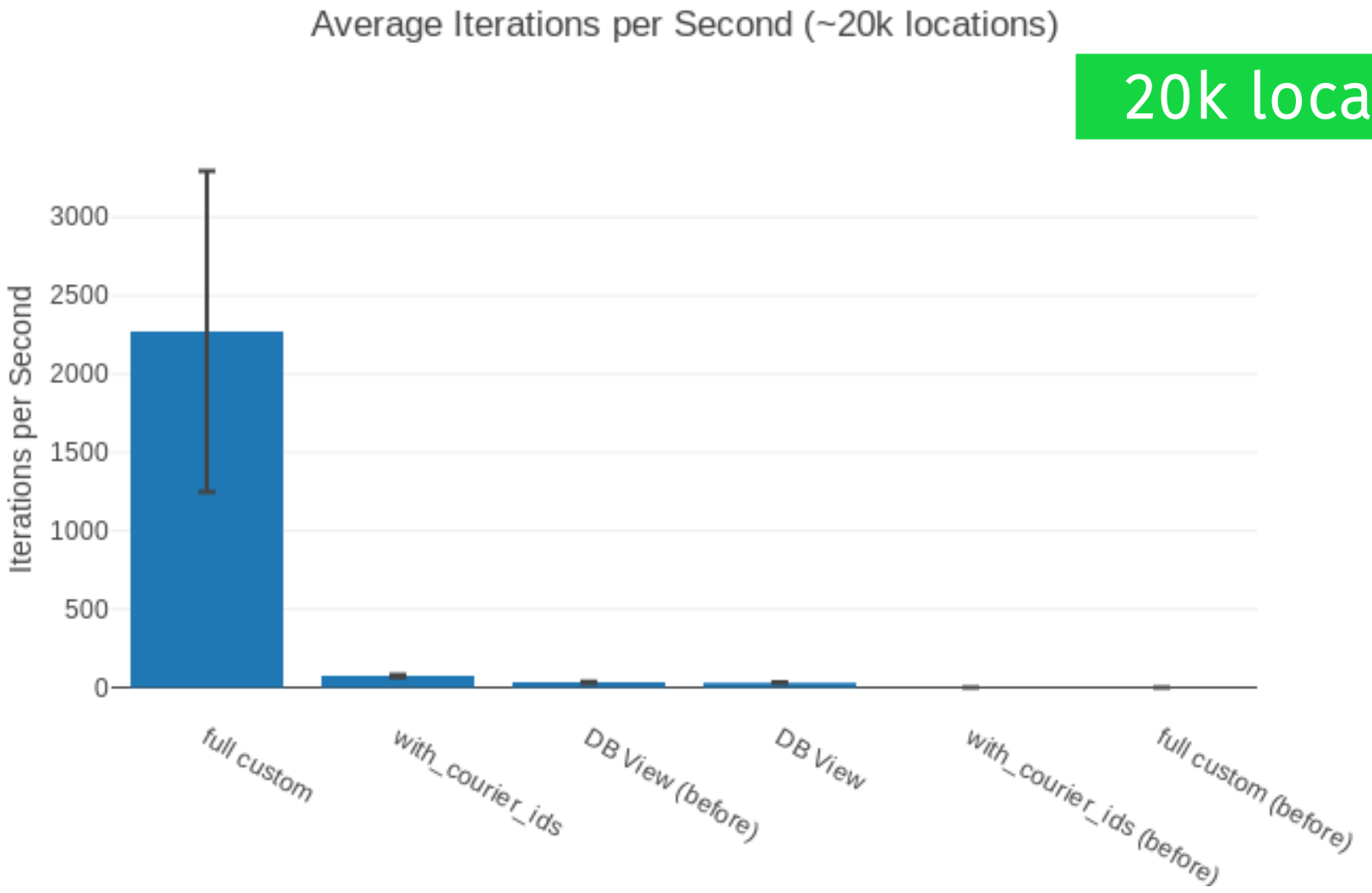
```
EXPLAIN ANALYZE
  SELECT c0."id",
         c0."courier_id",
         c0."location",
         c0."time",
         c0."accuracy",
         c0."inserted_at",
         c0."updated_at"
    FROM "courier_locations" AS c0
   WHERE (c0."courier_id" = 3799)
   ORDER BY c0."time" DESC LIMIT 1;

                                  QUERY PLAN
-------------------------------------------------------------------------
 Limit  (cost=0.43..0.83 rows=1 width=72)
        (actual time=1.840..1.841 rows=1 loops=1)
    ->  Index Scan Backward using courier_locations_time_index on
         courier_locations c0
        (cost=0.43..932600.17 rows=2386932 width=72)
        actual time=1.837..1.837 rows=1 loops=1)
          Filter: (courier_id = 3799)
          Rows Removed by Filter: 1371
 Planning time: 0.190 ms
 Execution time: 1.894 ms
(6 rows)
```

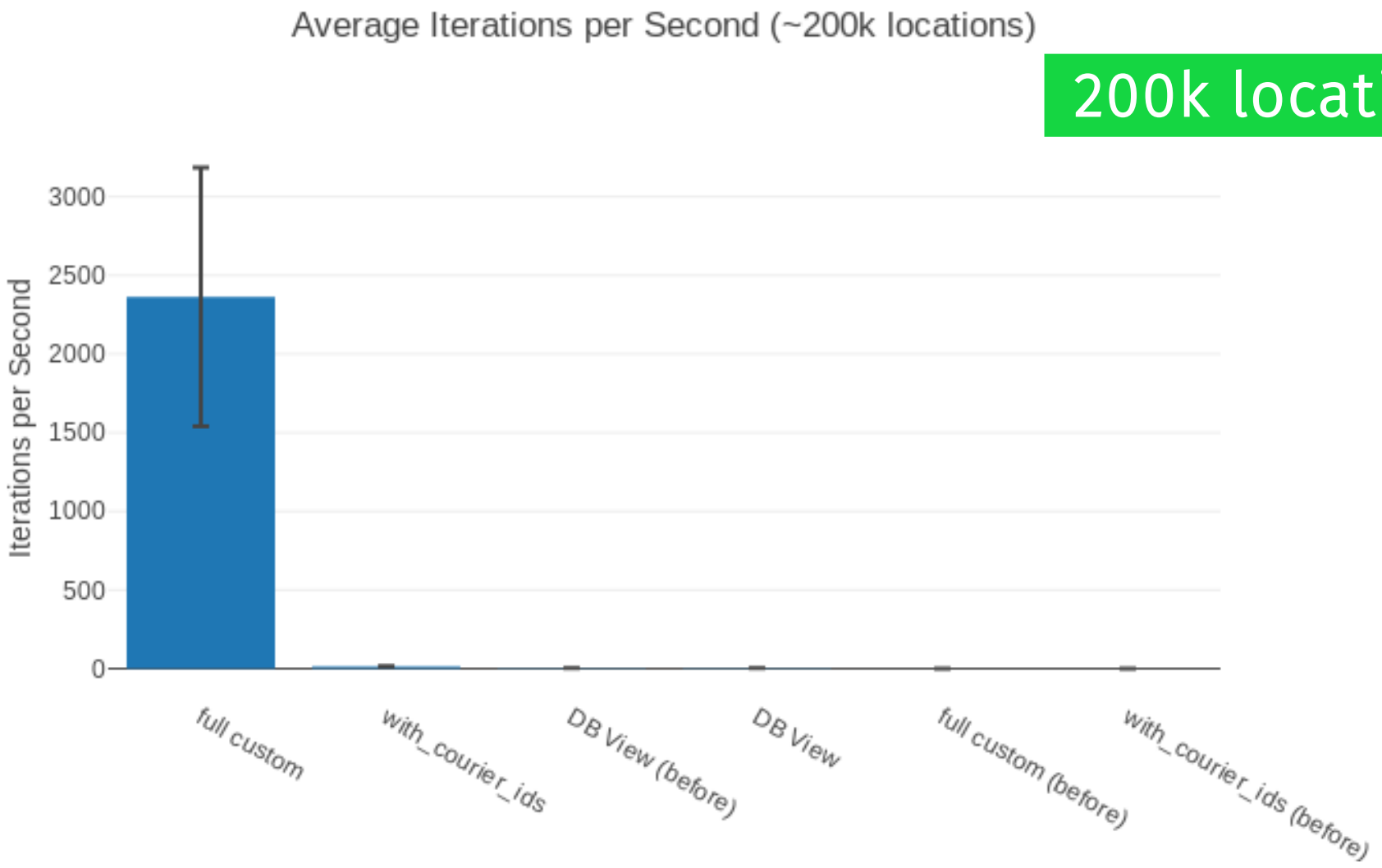
```
EXPLAIN ANALYZE
  SELECT c0."id",
         c0."courier_id",
         c0."location",
         c0."time",
         c0."accuracy",
         c0."inserted_at",
         c0."updated_at"
    FROM "courier_locations" AS c0
   WHERE (c0."courier_id" = 3799)
   ORDER BY c0."time" DESC LIMIT 1;

                                  QUERY PLAN
-------------------------------------------------------------------------
 Limit  (cost=0.43..0.83 rows=1 width=72)
        (actual time=1.840..1.841 rows=1 loops=1)
    ->  Index Scan Backward using courier_locations_time_index on
        courier_locations c0
       (cost=0.43..932600.17 rows=2386932 width=72)
       actual time=1.837..1.837 rows=1 loops=1)
         Filter: (courier_id = 3799)
         Rows Removed by Filter: 1371
 Planning time: 0.190 ms
 Execution time: 1.894 ms
(6 rows)
```

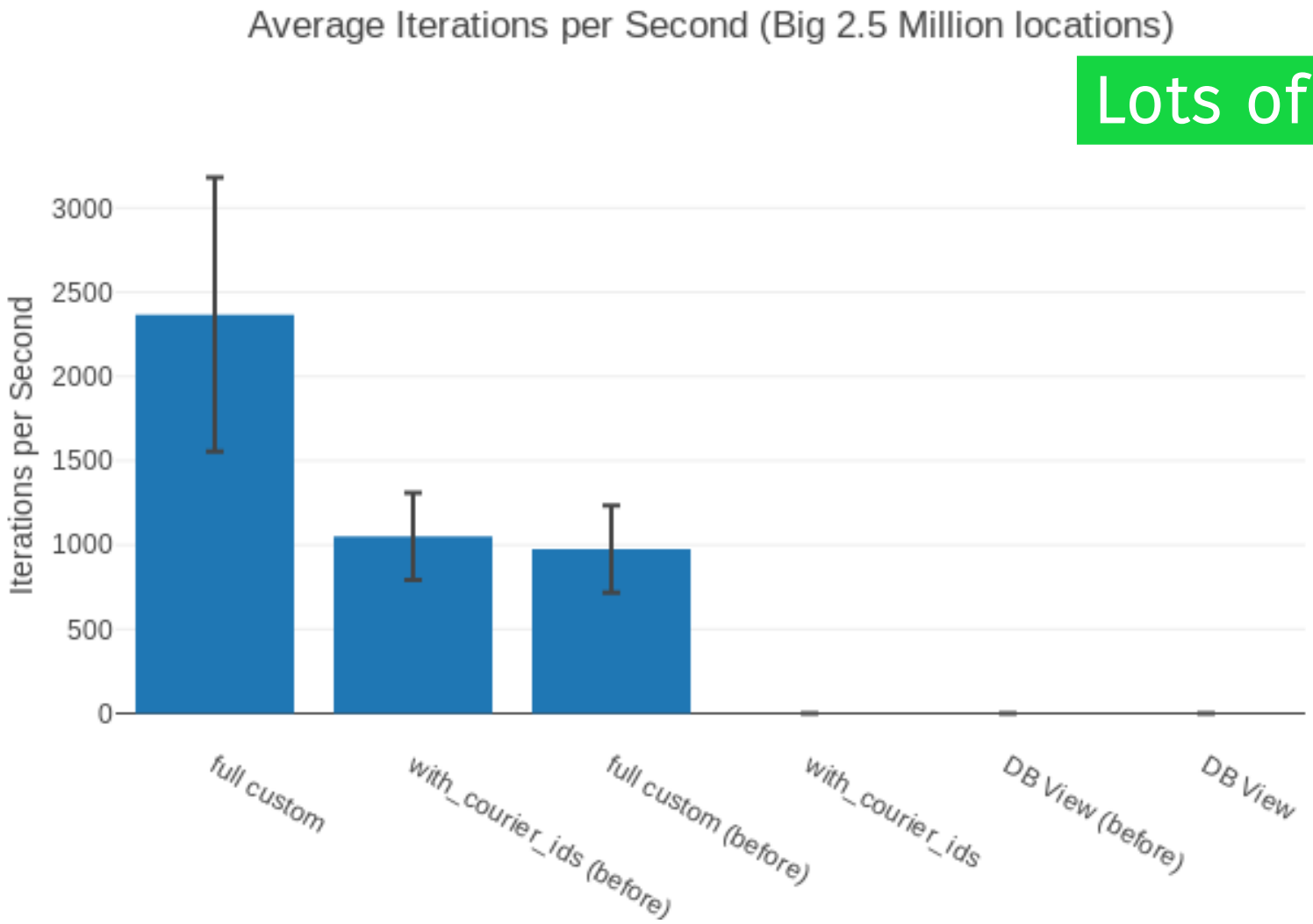
```
EXPLAIN ANALYZE
  SELECT c0."id",
         c0."courier_id",
         c0."location",
         c0."time",
         c0."accuracy",
         c0."inserted_at",
         c0."updated_at"
    FROM "courier_locations" AS c0
   WHERE (c0."courier_id" = 3799)
   ORDER BY c0."time" DESC LIMIT 1;

                                  QUERY PLAN
-------------------------------------------------------------------------
 Limit  (cost=0.43..0.83 rows=1 width=72)
        (actual time=1.840..1.841 rows=1 loops=1)
    ->  Index Scan Backward using courier_locations_time_index on
        courier_locations c0
       (cost=0.43..932600.17 rows=2386932 width=72)
       actual time=1.837..1.837 rows=1 loops=1)
         Filter: (courier_id = 3799)
         Rows Removed by Filter: 1371
 Planning time: 0.190 ms
 Execution time: 1.894 ms
(6 rows)
```

Combined Indexes


Average Iterations per Second (No locations)
No locations
Iterations per Second
4000
3000
2000
1000
0
full custom
with_courier_ids
DB View (before)
DB View
with_courier_ids (before)
full custom (before)


Average Iterations per Second (~20k locations)
20k locations
Iterations per Second
3000
2500
2000
1500
1000
500
0
full custom
with_courier_ids
DB View (before)
DB View
with_courier_ids (before)
full custom (before)


Average Iterations per Second (~200k locations)
200k locations
Iterations per Second
3000
2500
2000
1500
1000
500
0
full custom
with_courier_ids
DB View (before)
DB View
full custom (before)
with_courier_ids (before)


Average Iterations per Second (Big 2.5 Million locations)
Lots of locations
Iterations per Second
3000
2500
2000
1500
1000
500
0
full custom
with_courier_ids (before)
full custom (before)
with_courier_ids
DB View (before)
DB View

Let’s talk about

Tail Call Optimization

body

```
defmodule MyMap do
  def map_body([], _func), do: []
  def map_body([head | tail], func) do
    [func.(head) | map_body(tail, func)]
  end
end
```

body

```
defmodule MyMap do
  def map_body([], _func), do: []
  def map_body([head | tail], func) do
    [func.(head) | map_body(tail, func)]
  end
end
```

body

```
defmodule MyMap do
  def map_body([], _func), do: []
  def map_body([head | tail], func) do
    [func.(head) | map_body(tail, func)]
  end
end
```

body

```
defmodule MyMap do
  def map_body([], _func), do: []
  def map_body([head | tail], func) do
    [func.(head) | map_body(tail, func)]
  end
end
```

body

```elixir
defmodule MyMap do
  def map_body([], _func), do: []
  def map_body([head | tail], func) do
    [func.(head) | map_body(tail, func)]
  end
end
```

TCO

```
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco([], list, function)
  end

  defp do_map_tco(acc, [], _function) do
    acc
  end
  defp do_map_tco(acc, [head | tail], func) do
    do_map_tco([func.(head) | acc], tail, func)
  end
end
```

TCO

```
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco([], list, function)
  end

  defp do_map_tco(acc, [], _function) do
    acc
  end
  defp do_map_tco(acc, [head | tail], func) do
    do_map_tco([func.(head) | acc], tail, func)
  end
end
```

TCO

```
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco([], list, function)
  end

  defp do_map_tco(acc, [], _function) do
    acc
  end
  defp do_map_tco(acc, [head | tail], func) do
    do_map_tco([func.(head) | acc], tail, func)
  end
end
```

TCO

```
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco([], list, function)
  end

  defp do_map_tco(acc, [], _function) do
    acc
  end
  defp do_map_tco(acc, [head | tail], func) do
    do_map_tco([func.(head) | acc], tail, func)
  end
end
```

TCO

```
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco([], list, function)
  end

  defp do_map_tco(acc, [], _function) do
    acc
  end
  defp do_map_tco(acc, [head | tail], func) do
    do_map_tco([func.(head) | acc], tail, func)
  end
end
```

TCO

```
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco([], list, function)
  end

  defp do_map_tco(acc, [], _function) do
    acc
  end
  defp do_map_tco(acc, [head | tail], func) do
    do_map_tco([func.(head) | acc], tail, func)
  end
end
```

TCO

```
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco([], list, function)
  end

  defp do_map_tco(acc, [], _function) do
    acc
  end
  defp do_map_tco(acc, [head | tail], func) do
    do_map_tco([func.(head) | acc], tail, func)
  end
end
```

TCO

```
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco([], list, function)
  end

  defp do_map_tco(acc, [], _function) do
    acc
  end
  defp do_map_tco(acc, [head | tail], func) do
    do_map_tco([func.(head) | acc], tail, func)
  end
end
```

TCO

```
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco([], list, function)
  end

  defp do_map_tco(acc, [], _function) do
    acc
  end
  defp do_map_tco(acc, [head | tail], func) do
    do_map_tco([func.(head) | acc], tail, func)
  end
end
```

arg_order

```
defmodule MyMap do
  def map_tco(list, function) do
    Enum.reverse do_map_tco(list, function, [])
  end

  defp do_map_tco([], _function, acc) do
    acc
  end
  defp do_map_tco([head | tail], func, acc) do
    do_map_tco(tail, func, [func.(head) | acc])
  end
end
```

TCO

```
map_fun = fn(i) -> i + 1 end
inputs = %{
  "Small (10 Thousand)"    => Enum.to_list(1..10_000),
  "Middle (100 Thousand)" => Enum.to_list(1..100_000),
  "Big (1 Million)"         => Enum.to_list(1..1_000_000),
  "Bigger (5 Million)"     => Enum.to_list(1..5_000_000)
}

Benchee.run %{
  "tail-recursive" =>
    fn(list) -> MyMap.map_tco(list, map_fun) end,
  "stdlib map" =>
    fn(list) -> Enum.map(list, map_fun) end,
  "body-recursive" =>
    fn(list) -> MyMap.map_body(list, map_fun) end,
  "tail-rec arg-order" =>
    fn(list) -> MyMap.map_tco_arg_order(list, map_fun) end
}
```

```
map_fun = fn(i) -> i + 1 end
inputs = %{
  "Small (10 Thousand)"    => Enum.to_list(1..10_000),
  "Middle (100 Thousand)" => Enum.to_list(1..100_000),
  "Big (1 Million)"         => Enum.to_list(1..1_000_000),
  "Bigger (5 Million)"     => Enum.to_list(1..5_000_000)
}

Benchee.run %{
  "tail-recursive" =>
    fn(list) -> MyMap.map_tco(list, map_fun) end,
  "stdlib map" =>
    fn(list) -> Enum.map(list, map_fun) end,
  "body-recursive" =>
    fn(list) -> MyMap.map_body(list, map_fun) end,
  "tail-rec arg-order" =>
    fn(list) -> MyMap.map_tco_arg_order(list, map_fun) end
}
```

TCO

TCO

```
map_fun = fn(i) -> i + 1 end
inputs = %{
  "Small (10 Thousand)"    => Enum.to_list(1..10_000),
  "Middle (100 Thousand)" => Enum.to_list(1..100_000),
  "Big (1 Million)"         => Enum.to_list(1..1_000_000),
  "Bigger (5 Million)"     => Enum.to_list(1..5_000_000)
}
```

```
Benchee.run %{
  "tail-recursive" =>
    fn(list) -> MyMap.map_tco(list, map_fun) end,
  "stdlib map" =>
    fn(list) -> Enum.map(list, map_fun) end,
  "body-recursive" =>
    fn(list) -> MyMap.map_body(list, map_fun) end,
  "tail-rec arg-order" =>
    fn(list) -> MyMap.map_tco_arg_order(list, map_fun) end
}
```

TCO

```
##### With input Small (10 Thousand) #####
stdlib map                 5.05 K
body-recursive             5.00 K - 1.01x slower
tail-rec arg-order         4.07 K - 1.24x slower
tail-recursive             3.76 K - 1.34x slower

##### With input Middle (100 Thousand) #####
stdlib map                 468.49
body-recursive             467.04 - 1.00x slower
tail-rec arg-order         452.53 - 1.04x slower
tail-recursive             417.20 - 1.12x slower

##### With input Big (1 Million) #####
body-recursive              40.33
stdlib map                  38.89 - 1.04x slower
tail-rec arg-order          37.69 - 1.07x slower
tail-recursive              33.29 - 1.21x slower

##### With input Bigger (5 Million) #####
tail-rec arg-order           6.68
tail-recursive               6.35 - 1.05x slower
stdlib map                   5.60 - 1.19x slower
body-recursive               5.39 - 1.24x slower
```

TCO

```
##### With input Small (10 Thousand) #####
stdlib map                 5.05 K
body-recursive             5.00 K - 1.01x slower
tail-rec arg-order         4.07 K - 1.24x slower
tail-recursive             3.76 K - 1.34x slower

##### With input Middle (100 Thousand) #####
stdlib map                468.49
body-recursive            467.04 - 1.00x slower
tail-rec arg-order        452.53 - 1.04x slower
tail-recursive            417.20 - 1.12x slower

##### With input Big (1 Million) #####
body-recursive             40.33
stdlib map                 38.89 - 1.04x slower
tail-rec arg-order         37.69 - 1.07x slower
tail-recursive             33.29 - 1.21x slower

##### With input Bigger (5 Million) #####
tail-rec arg-order          6.68
tail-recursive              6.35 - 1.05x slower
stdlib map                  5.60 - 1.19x slower
body-recursive              5.39 - 1.24x slower
```

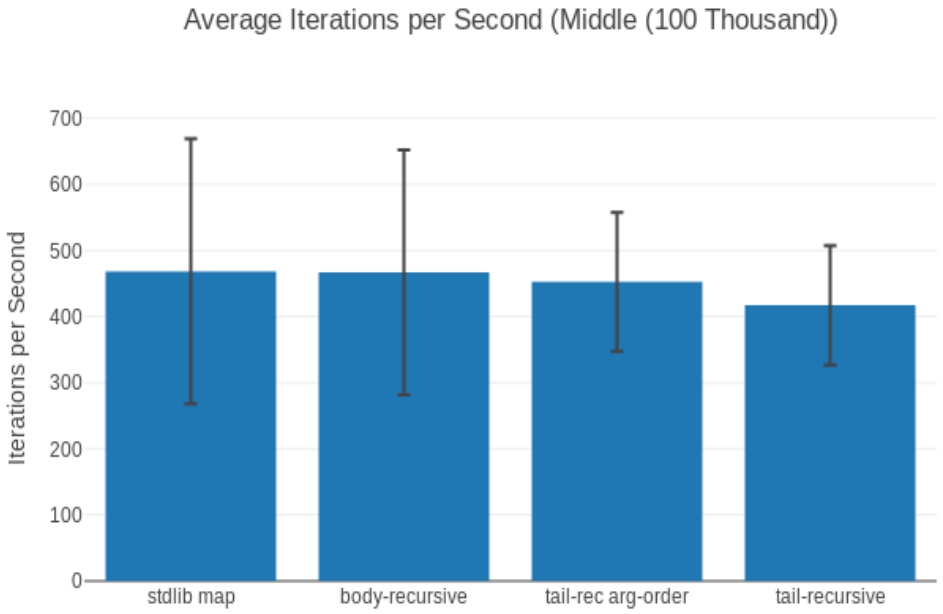
Big Inputs

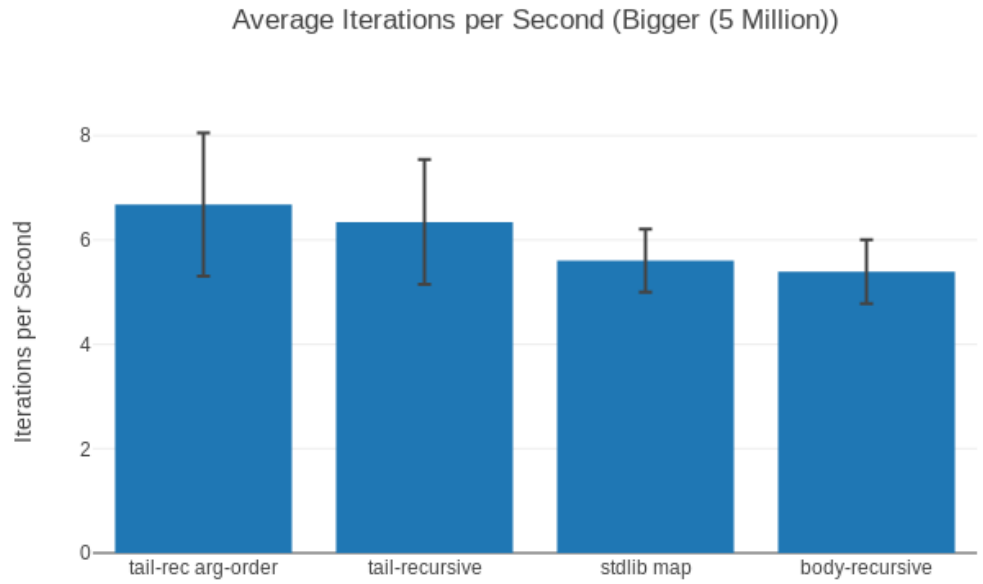
```
##### With input Small (10 Thousand) #####
stdlib map                  5.05 K
body-recursive              5.00 K - 1.01x slower
tail-rec arg-order          4.07 K - 1.24x slower
tail-recursive              3.76 K - 1.34x slower

##### With input Middle (100 Thousand) #####
stdlib map                  468.49
body-recursive              467.04 - 1.00x slower
tail-rec arg-order          452.53 - 1.04x slower
tail-recursive              417.20 - 1.12x slower

##### With input Big (1 Million) #####
body-recursive               40.33
stdlib map                   38.89 - 1.04x slower
tail-rec arg-order           37.69 - 1.07x slower
tail-recursive               33.29 - 1.21x slower

##### With input Bigger (5 Million) #####
tail-rec arg-order            6.68
tail-recursive                6.35 - 1.05x slower
stdlib map                    5.60 - 1.19x slower
body-recursive                5.39 - 1.24x slower
```

Order matters!

```
##### With input Small (10 Thousand) #####
stdlib map                  5.05 K
body-recursive              5.00 K - 1.01x slower
tail-rec arg-order          4.07 K - 1.24x slower
tail-recursive              3.76 K - 1.34x slower

##### With input Middle (100 Thousand) #####
stdlib map                  468.49
body-recursive              467.04 - 1.00x slower
tail-rec arg-order          452.53 - 1.04x slower
tail-recursive              417.20 - 1.12x slower

##### With input Big (1 Million) #####
body-recursive               40.33
stdlib map                   38.89 - 1.04x slower
tail-rec arg-order           37.69 - 1.07x slower
tail-recursive               33.29 - 1.21x slower

##### With input Bigger (5 Million) #####
tail-rec arg-order             6.68
tail-recursive                 6.35 - 1.05x slower
stdlib map                     5.60 - 1.19x slower
body-recursive                 5.39 - 1.24x slower
```

TCO


Average Iterations per Second (Middle (100 Thousand))
Iterations per Second
700
600
500
400
300
200
100
0
stdlib map
body-recursive
tail-rec arg-order
tail-recursive


Average Iterations per Second (Bigger (5 Million))
Iterations per Second
8
6
4
2
0
tail-rec arg-order
tail-recursive
stdlib map
body-recursive

What about Memory?

Calculate memory stats #180

WIP

Open **devonestes** wants to merge 12 commits into `master` from `calculate-memory-stats`

Conversation 3 | Commits 12 | Files changed 17

devonestes commented 2 days ago — Collaborator

Ok, at this point a lot is going on here. Right now there are *minimal* tests on the new behavior, which I've used to refactor the initial messy implementation into something not too bad. At this point, things I still have left to do are:

- rename some functions that still imply relationship to run time measurements
- add a bunch more tests for new behavior
- document this fully
- add extended output for memory usage stats if requested

The major things I've done in this PR so far are:

- add **@michalmuskala**'s new memory measurement module
- add memory measurement to the console formatter
- split the console formatter into three modules - one for run times, one for memory, and one for shared functions between the two of them

So, right now just about everything is working (except the extended statistics for memory), and if y'all wanted to you could play with it. It'll take some time to fully test and document everything, which is why I wanted to put this up now to get feedback while I'm starting on that stuff.

1

Memory

```
##### With input Small (10 Thousand) #####
stdlib map                156.85 KB
body-recursive            156.85 KB - 1.00x memory usage
tail-rec arg-order        291.46 KB - 1.86x memory usage
tail-recursive            291.46 KB - 1.86x memory usage

##### With input Middle (100 Thousand) #####
stdlib map                  1.53 MB
body-recursive              1.53 MB - 1.00x memory usage
tail-rec arg-order          1.80 MB - 1.18x memory usage
tail-recursive              1.80 MB - 1.18x memory usage

##### With input Big (1 Million) #####
stdlib map                 15.26 MB
body-recursive             15.26 MB - 1.00x memory usage
tail-rec arg-order         28.74 MB - 1.88x memory usage
tail-recursive             28.74 MB - 1.88x memory usage

##### With input Bigger (5 Million) #####
tail-rec arg-order        150.15 MB
tail-recursive            150.15 MB - 1.00x memory usage
stdlib map                 76.30 MB - 0.51x memory usage
body-recursive             76.30 MB - 0.51x memory usage
```

Memory

```
##### With input Small (10 Thousand) #####
stdlib map                156.85 KB
body-recursive            156.85 KB - 1.00x memory usage
tail-rec arg-order        291.46 KB - 1.86x memory usage
tail-recursive            291.46 KB - 1.86x memory usage

##### With input Middle (100 Thousand) #####
stdlib map                  1.53 MB
body-recursive              1.53 MB - 1.00x memory usage
tail-rec arg-order          1.80 MB - 1.18x memory usage
tail-recursive              1.80 MB - 1.18x memory usage

##### With input Big (1 Million) #####
stdlib map                 15.26 MB
body-recursive             15.26 MB - 1.00x memory usage
tail-rec arg-order         28.74 MB - 1.88x memory usage
tail-recursive             28.74 MB - 1.88x memory usage

##### With input Bigger (5 Million) #####
tail-rec arg-order        150.15 MB
tail-recursive            150.15 MB - 1.00x memory usage
stdlib map                 76.30 MB - 0.51x memory usage
body-recursive             76.30 MB - 0.51x memory usage
```

What even is a benchmark?

A transformation of inputs

```
config
|> Benchee.init
|> Benchee.system
|> Benchee.benchmark("job", fn -> magic end)
|> Benchee.measure
|> Benchee.statistics
|> Benchee.Formatters.Console.output
|> Benchee.Formatters.HTML.output
```

RUN YOUR OWN BENCHMARKS

Enjoy Benchmarking ❤

Tobias Pfeiffer
@PragTob
pragtob.info
github.com/PragTob/benchee

LIEFERY

Excursion into Statistics

Average

```
average = total_time / iterations
```

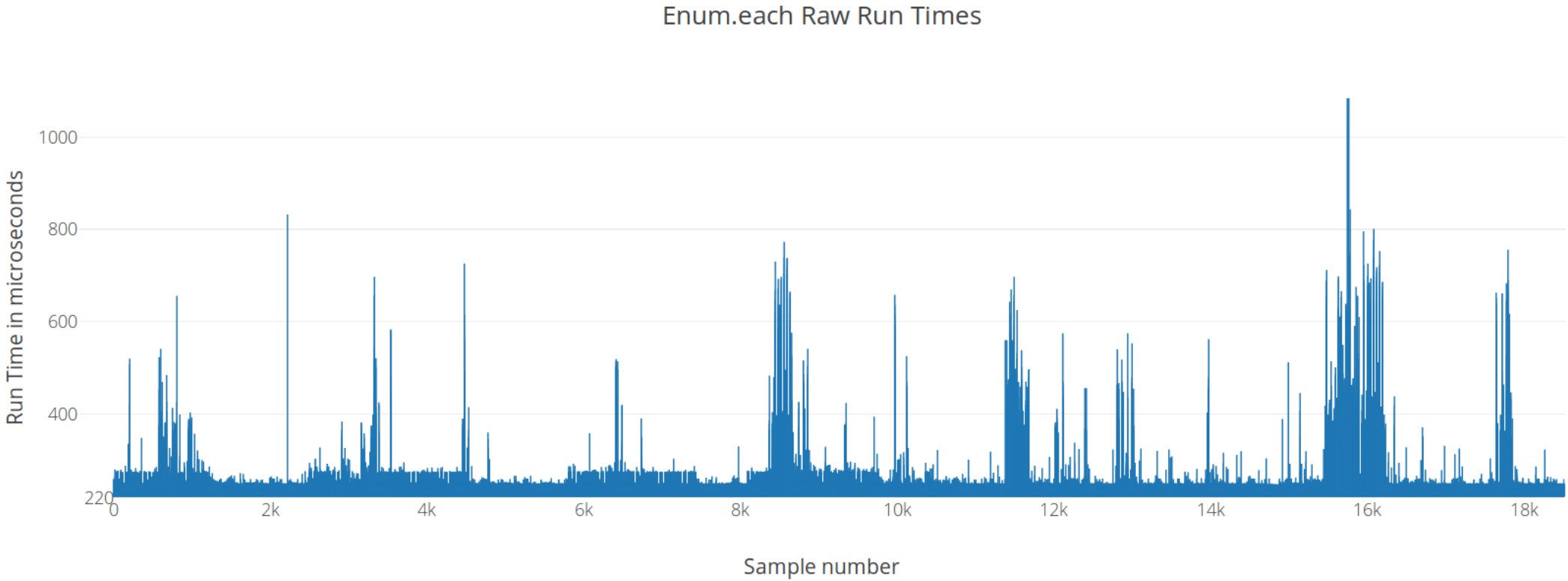
Standard Deviation

```
defp standard_deviation(samples, average, iterations) do
  total_variance = Enum.reduce samples, 0,  fn(sample, total) ->
    total + :math.pow((sample - average), 2)
  end
  variance = total_variance / iterations
  :math.sqrt variance
end
```

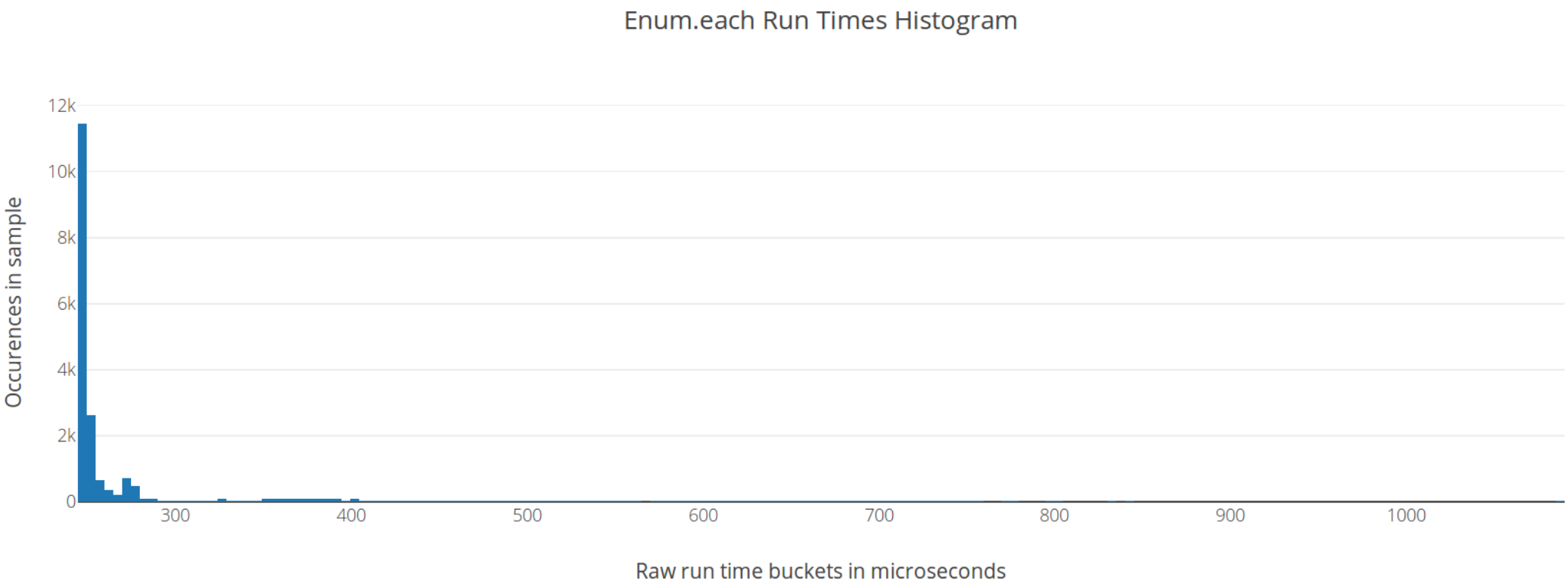
Spread of Values

```
defp standard_deviation(samples, average, iterations) do
  total_variance = Enum.reduce samples, 0,  fn(sample, total) ->
    total + :math.pow((sample - average), 2)
  end
  variance = total_variance / iterations
  :math.sqrt variance
end
```

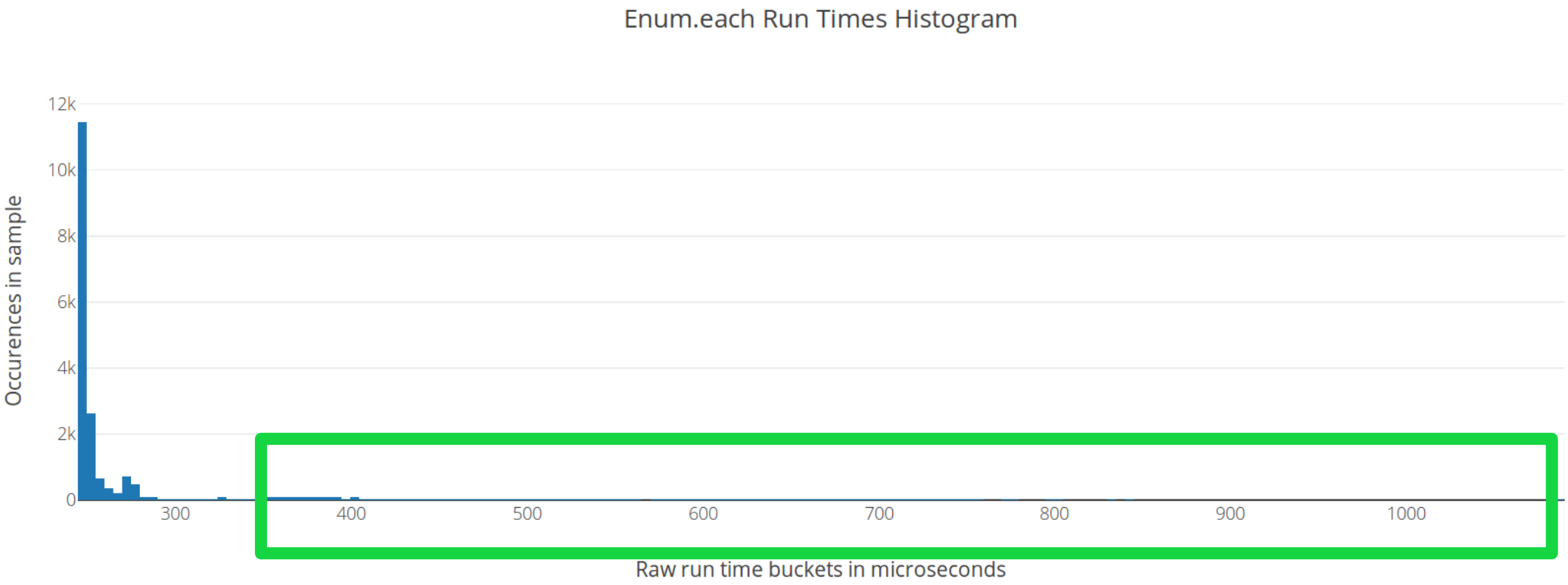
Raw Run Times



Histogram


Enum.each Run Times Histogram
12k
10k
8k
6k
4k
2k
0
Occurences in sample
300
400
500
600
700
800
900
1000
Raw run time buckets in microseconds

Outliers

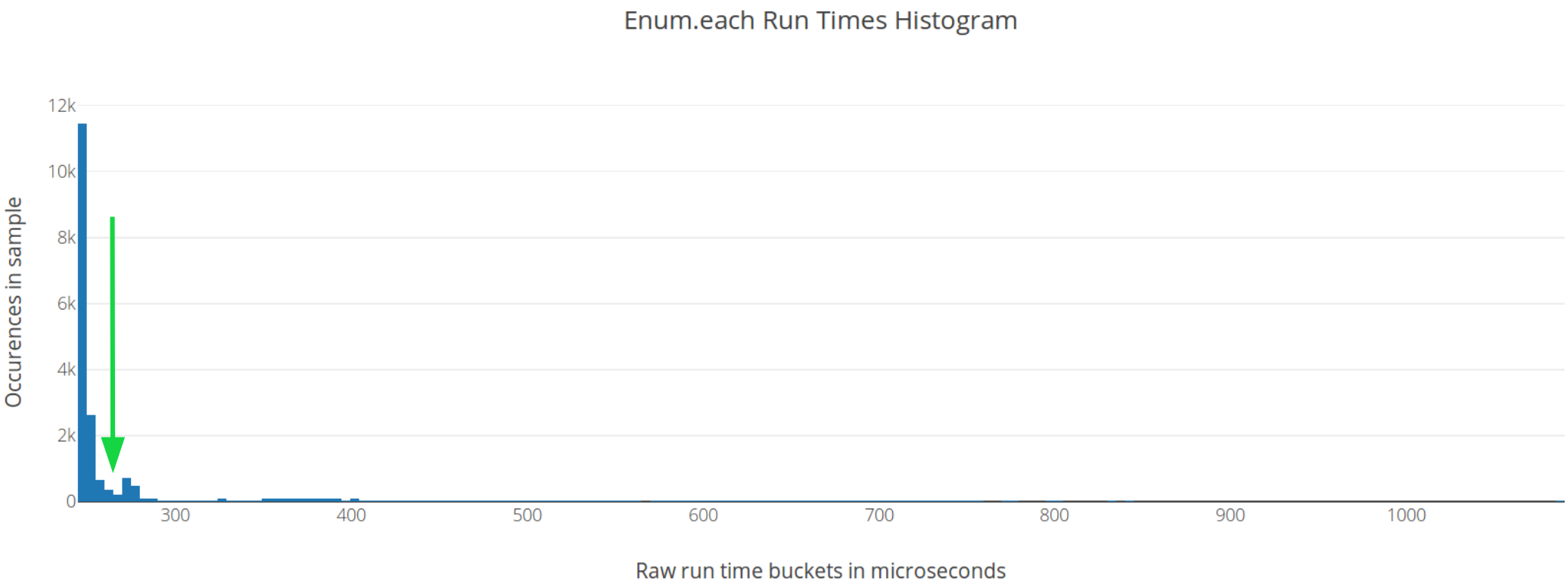

Enum.each Run Times Histogram
12k
10k
8k
6k
4k
2k
0
Occurences in sample
300
400
500
600
700
800
900
1000
Raw run time buckets in microseconds


Average Iterations per Seco
Standard Deviation
Iterations per Second
7000
6000
5000
4000
3000
2000
1000
0
Recursion
Enum.each
List comprehension

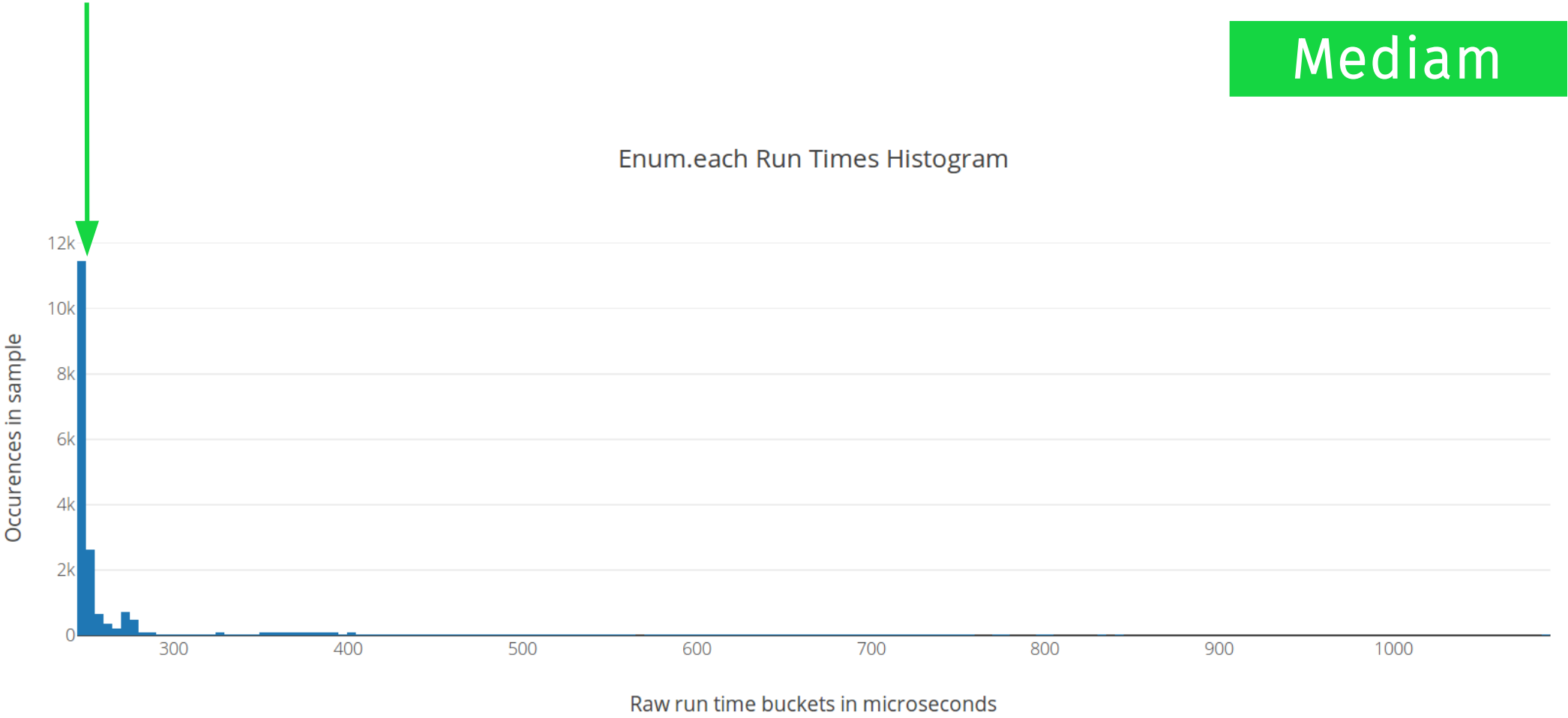
Median

```
defp compute_median(run_times, iterations) do
  sorted = Enum.sort(run_times)
  middle = div(iterations, 2)

  if Integer.is_odd(iterations) do
    sorted |> Enum.at(middle) |> to_float
  else
    (Enum.at(sorted, middle) +
      Enum.at(sorted, middle - 1)) / 2
  end
end
```

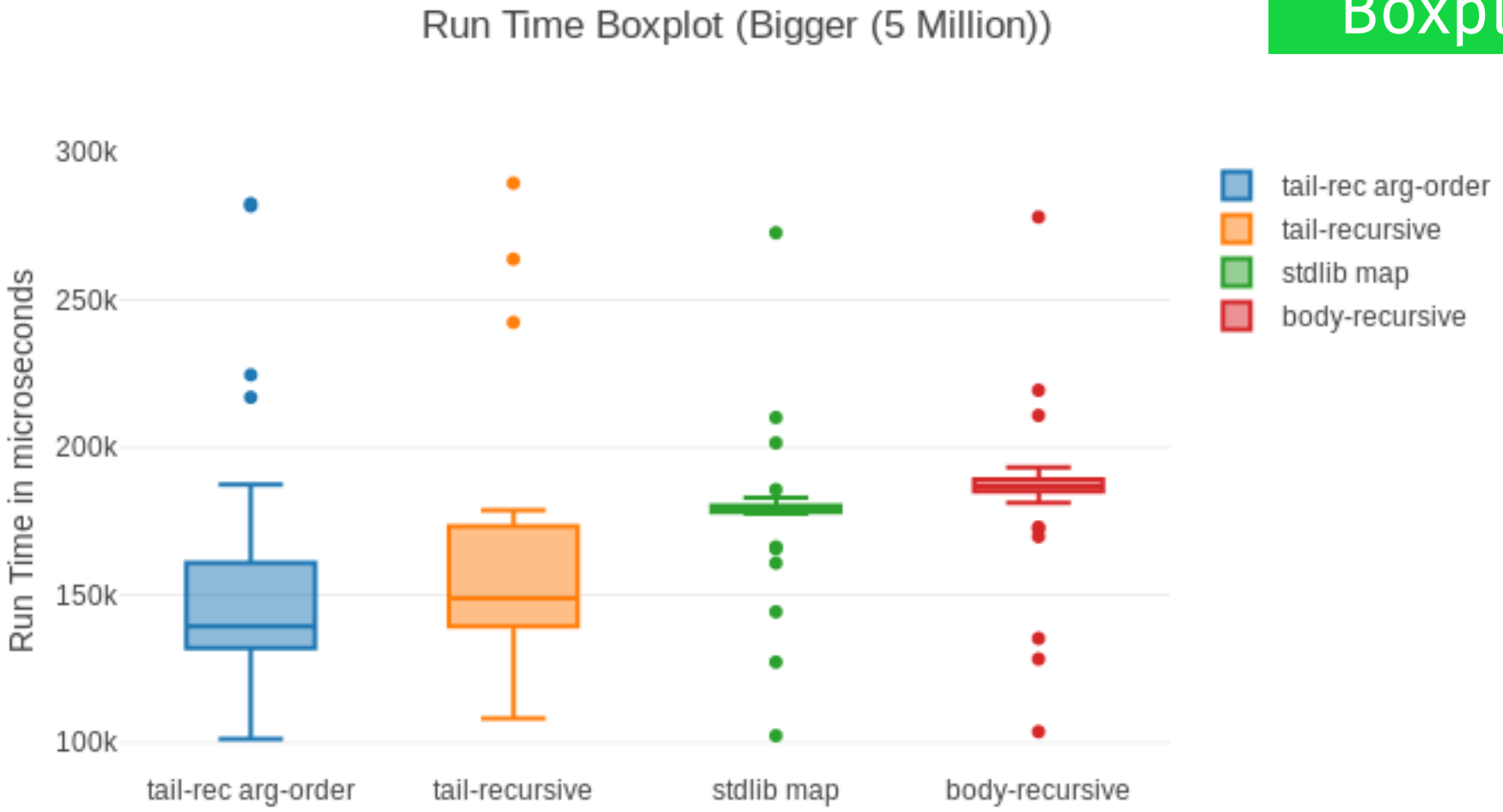
Average


Enum.each Run Times Histogram
12k
10k
8k
6k
4k
2k
0
Occurences in sample
300
400
500
600
700
800
900
1000
Raw run time buckets in microseconds

Mediam


Enum.each Run Times Histogram
12k
10k
8k
6k
4k
2k
0
Occurences in sample
300
400
500
600
700
800
900
1000
Raw run time buckets in microseconds

Boxplot


Run Time Boxplot (Bigger (5 Million))
Run Time in microseconds
300k
250k
200k
150k
100k
tail-rec arg-order
tail-recursive
stdlib map
body-recursive
tail-rec arg-order
tail-recursive
stdlib map
body-recursive