Rails and the Ruby Garbage Collector
How to Speed Up Your Rails App

Peter Zhu

Ruby Core Committer
Senior Developer, Shopify

blog.peterzhu.ca/assets/rails_world_2023_slides.pdf

What’s a garbage collector?

- Garbage collectors are responsible for the entire lifecycle of objects
- Garbage collectors perform memory allocations and deallocations
- Garbage collectors keep track of lifetimes of objects
- Ruby is a garbage collected language

What’s an object?

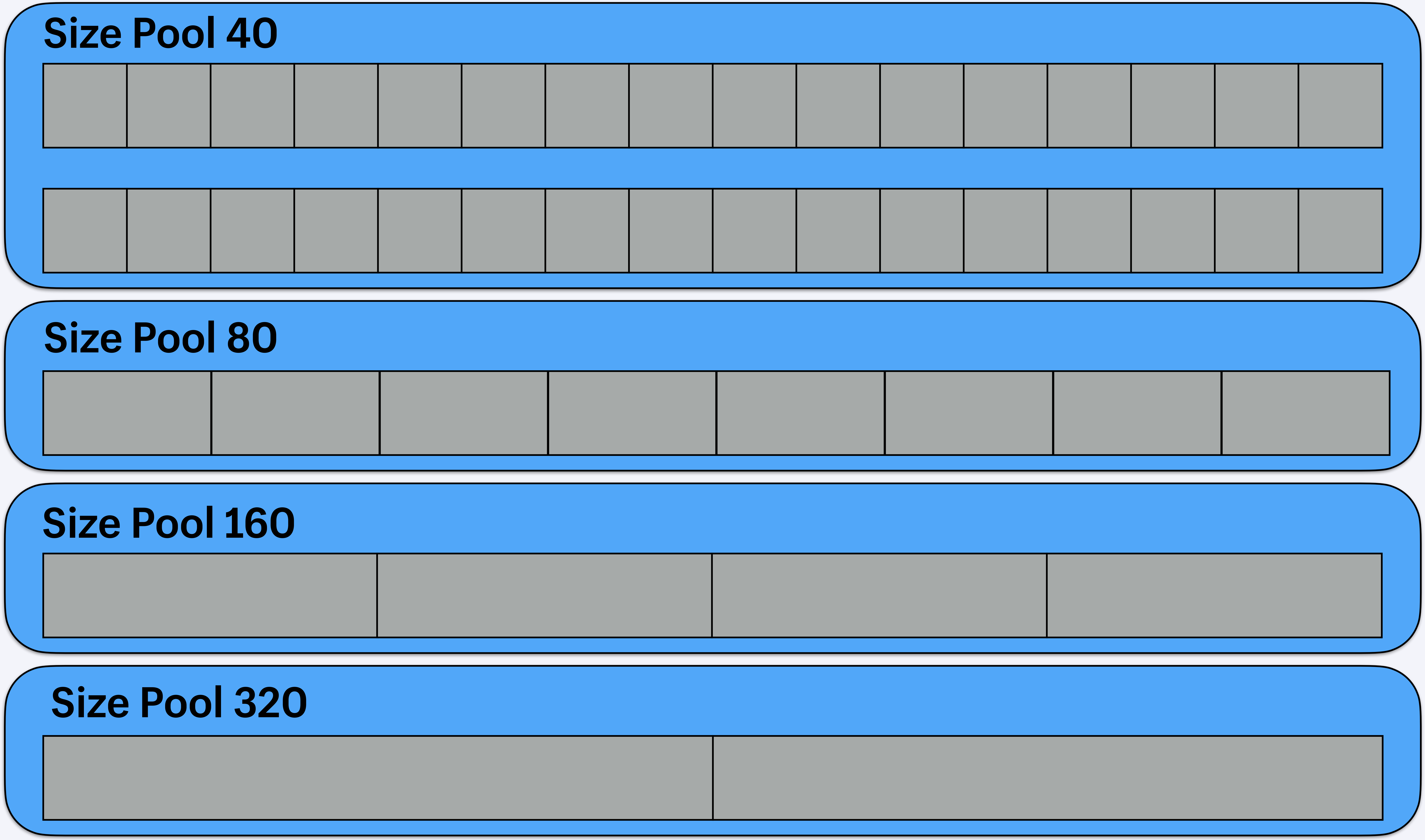
- Objects in Ruby live in slots
- Slots are acquired from the garbage collector
- Data that doesn’t fit in the slot is allocated externally

Pages

- Slots live in pages
- Pages are 64kb
- All the slots in a page are of the same size
- Fix sized slots avoids external fragmentation

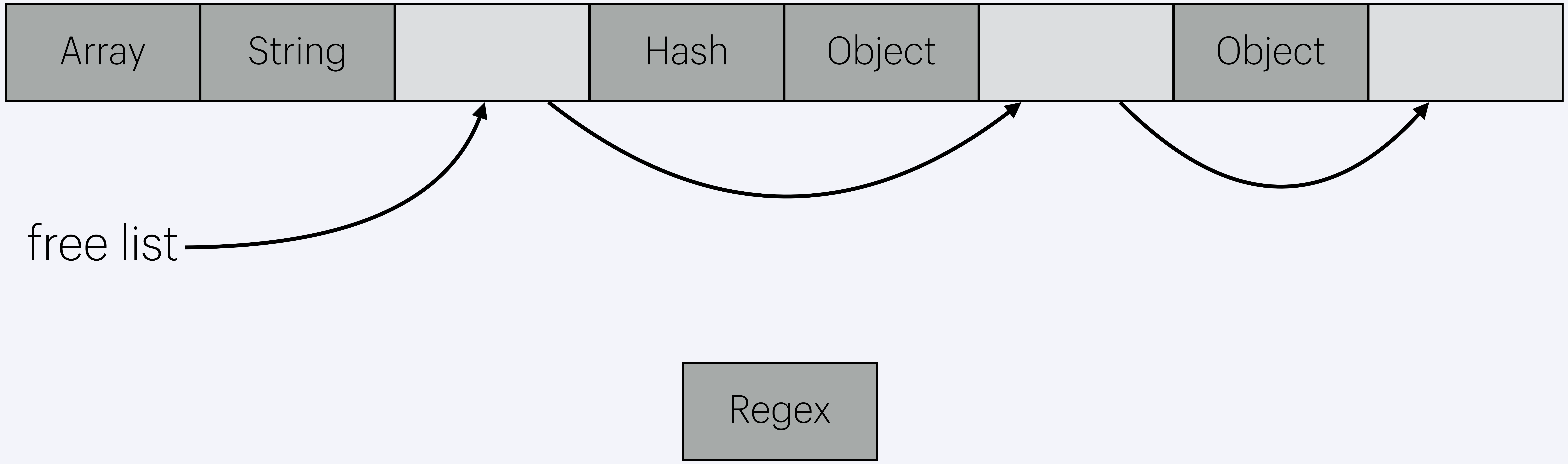
Size Pools

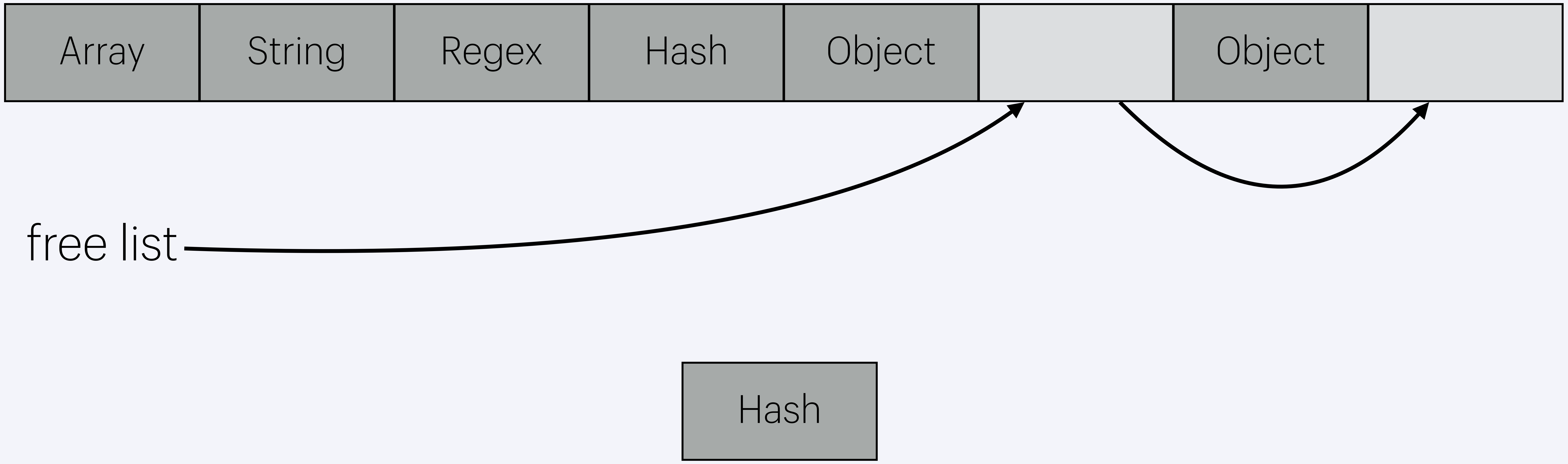
- Introduced in Ruby 3.2 for Variable Width Allocation
- Pages live in size pools
- Each size pool contains pages with the same slot size
- Currently 5 size pools: 40, 80, 160, 320, 640 bytes

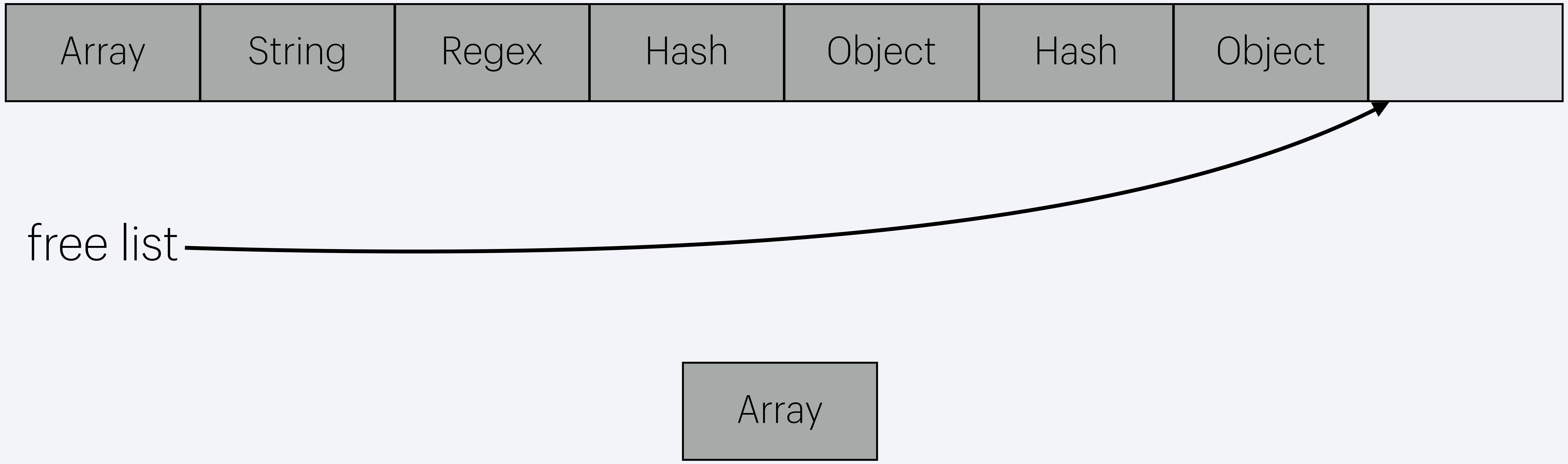

Size Pool 40
Size Pool 80
Size Pool 160
Size Pool 320

Object allocation

- Linked list of free slots called the “free list”
- An element is removed from the free list to allocate an object


Array
String
Hash
Object
Object
free list
Regex


Array
String
Regex
Hash
Object
Object
free list
Hash

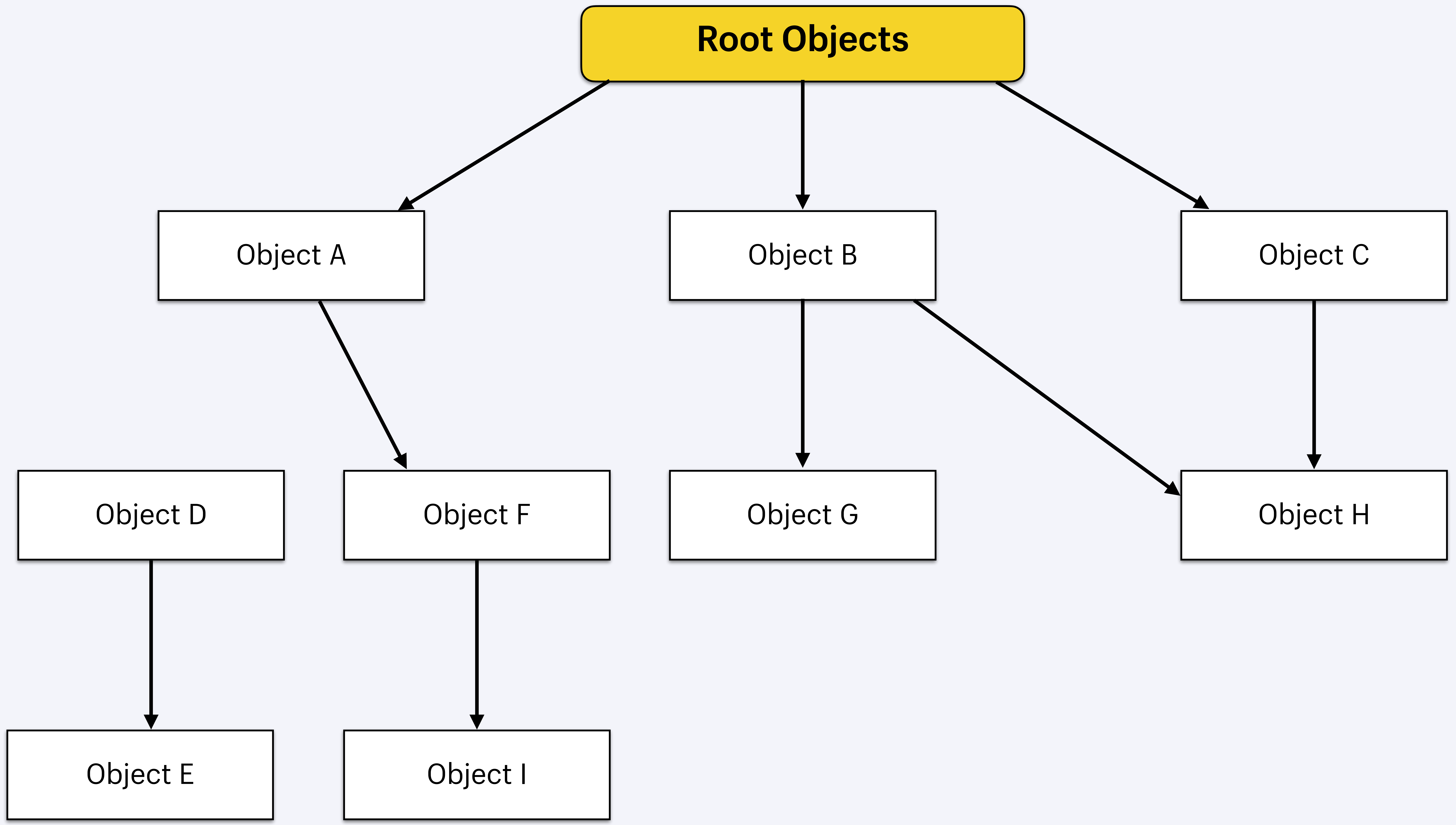

Array
String
Regex
Hash
Object
Hash
Object
free list
Array

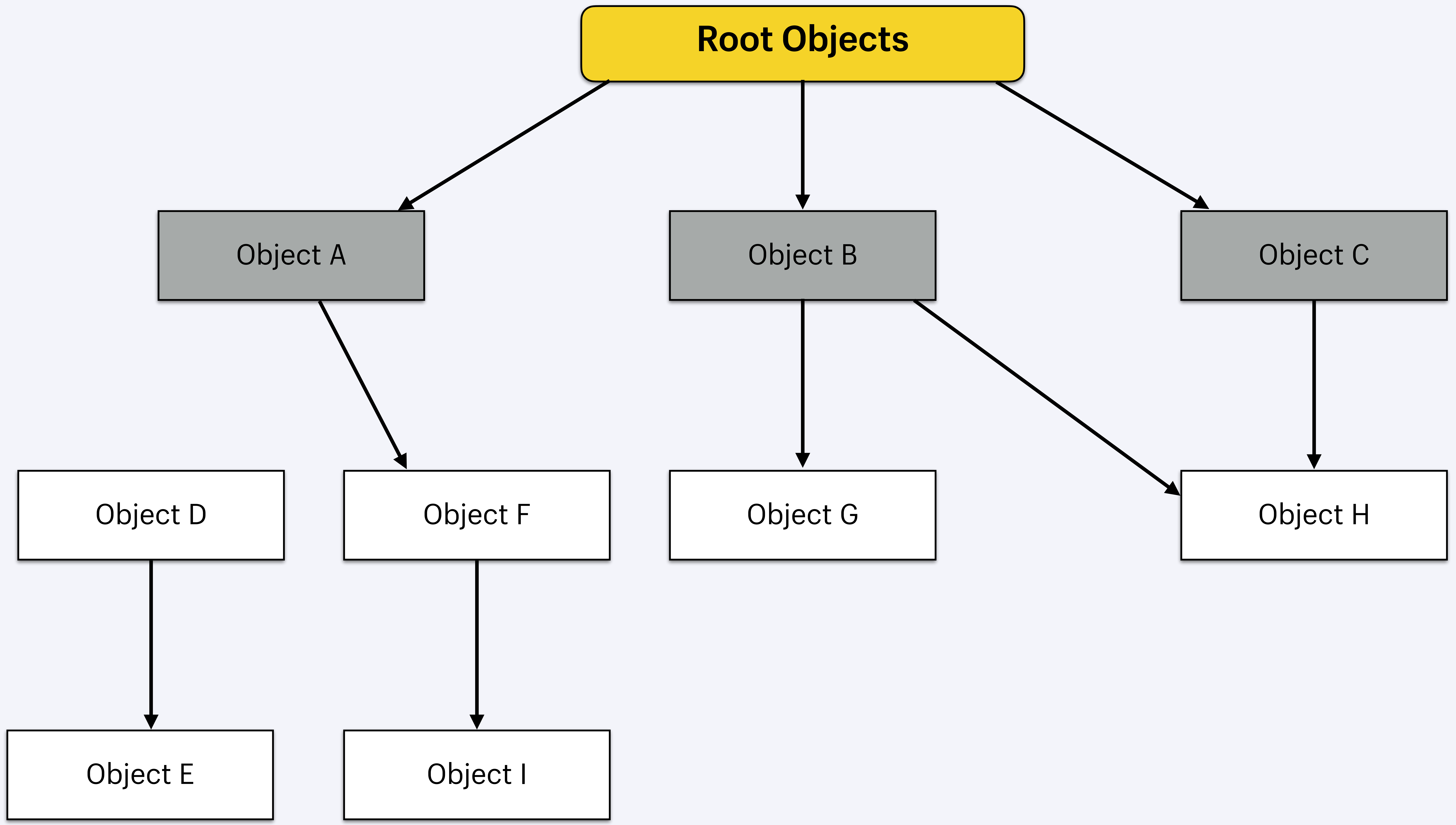
Garbage collection

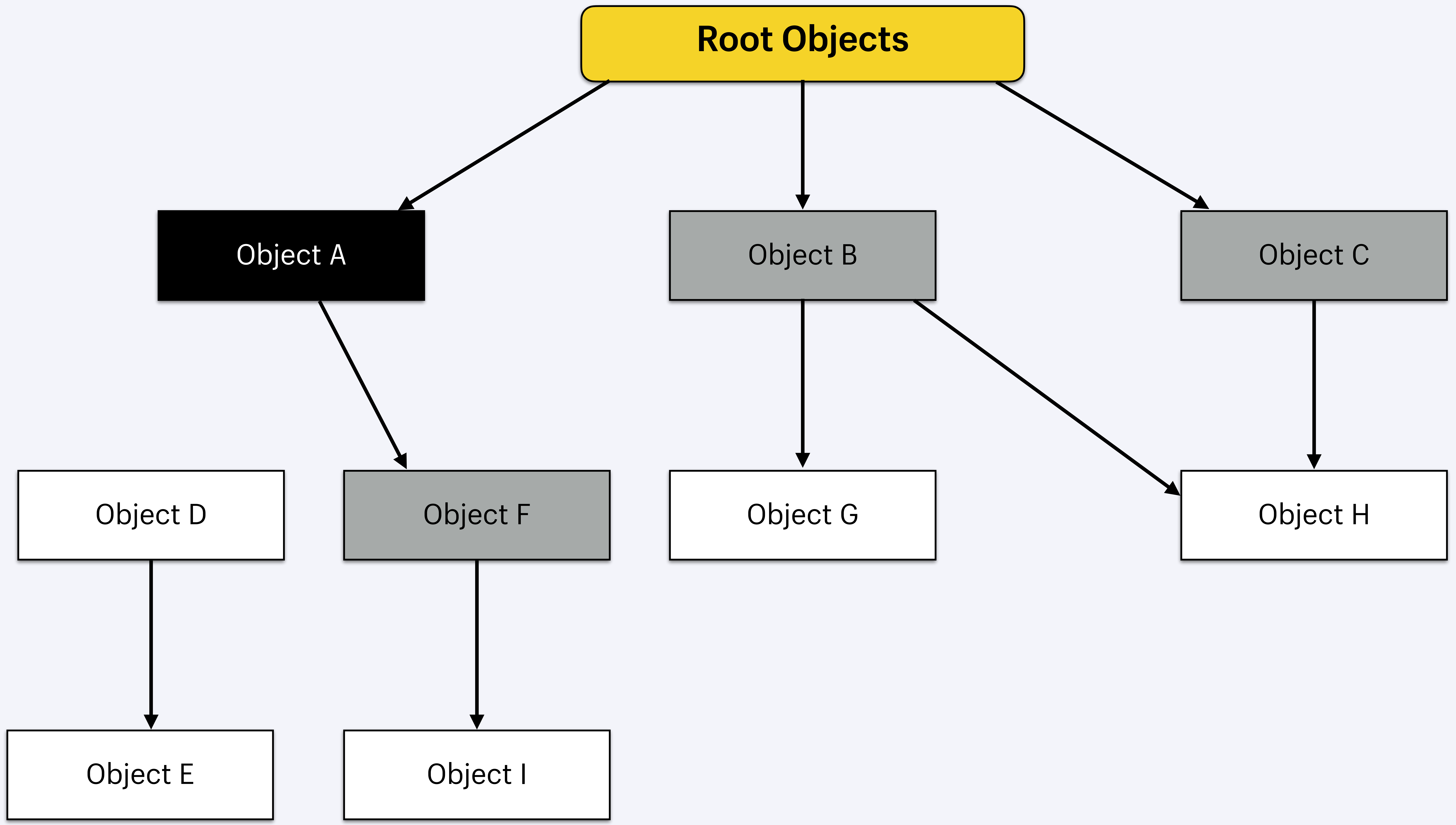
- Two phases in a garbage collection cycle: mark and sweep
- In the mark phase, live objects are marked
- In the sweep phase, unmarked objects are reclaimed by the garbage collector
- The (optional) compaction phase runs during sweeping to reduce fragmentation
- Ruby uses a “stop-the-world” garbage collector

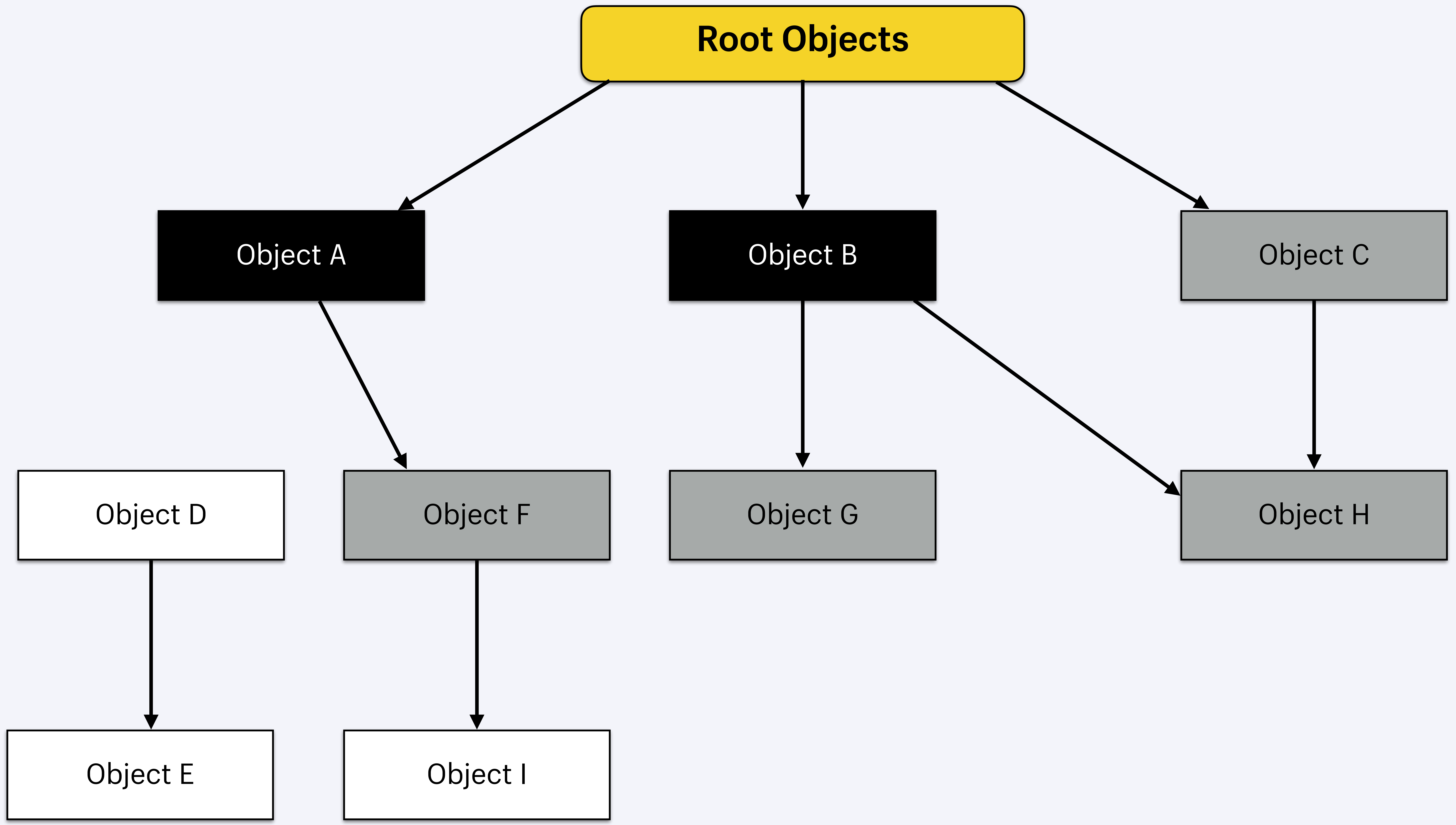
Marking phase

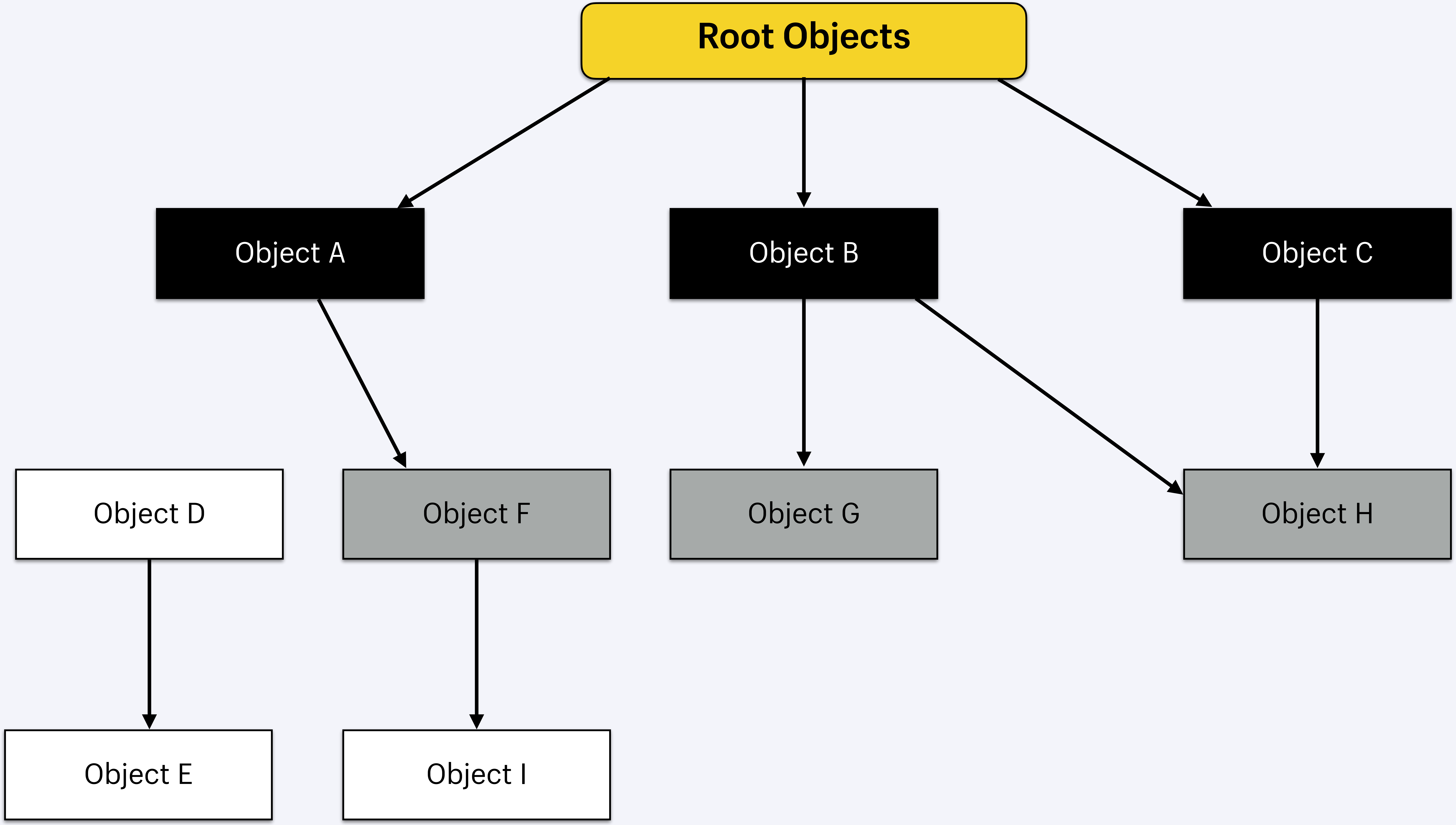
- Marking phase traverses object references to determine live objects
- Ruby marks objects with one of three colours:
  - White for unmarked objects
  - Grey for objects that are marked, but not traversed
  - Black for marked and traversed objects
- At the end of marking, all unmarked objects are dead

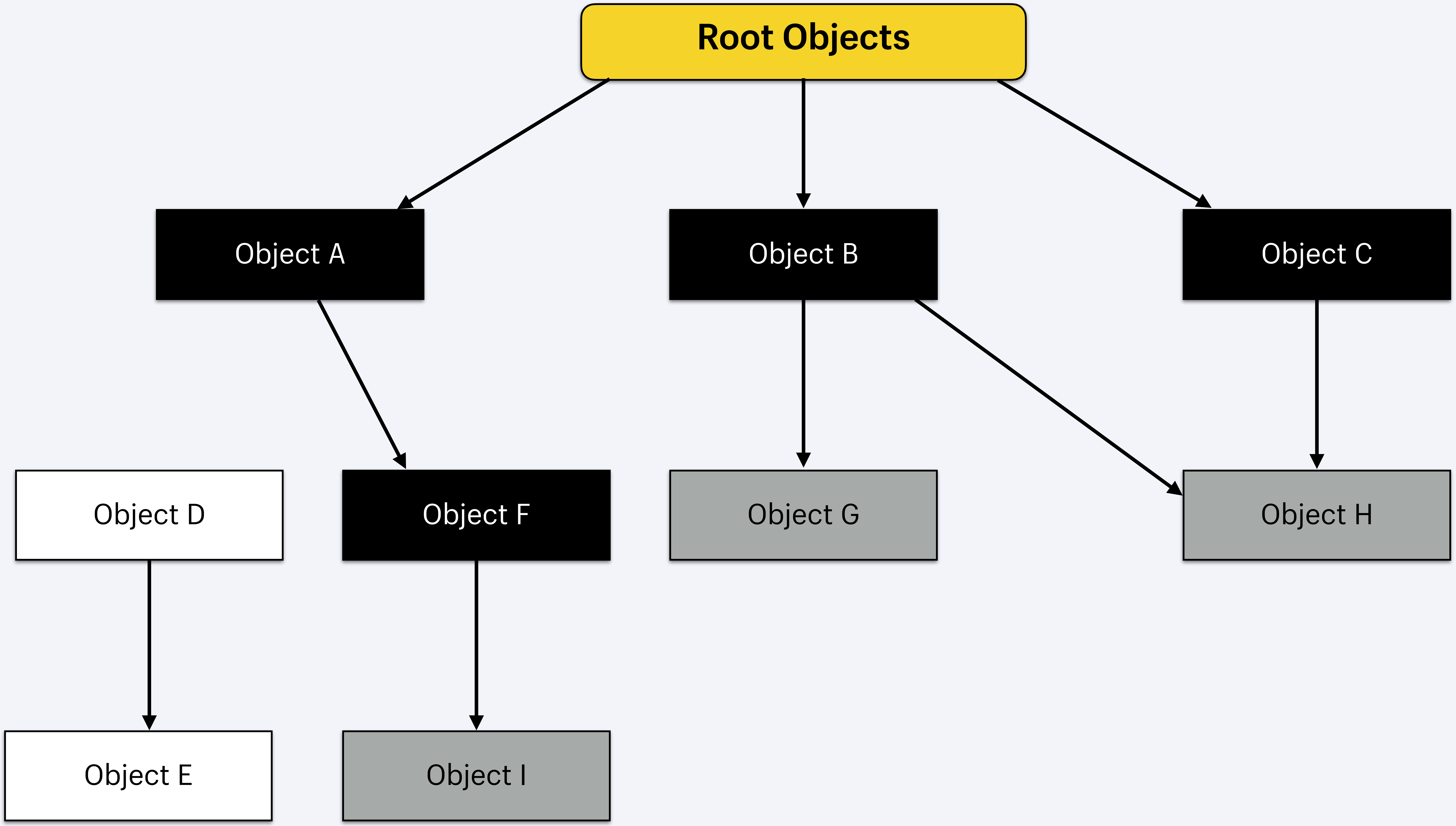

Root Objects
Object A
Object B
Object C
Object D
Object F
Object G
Object H
Object E
Object I

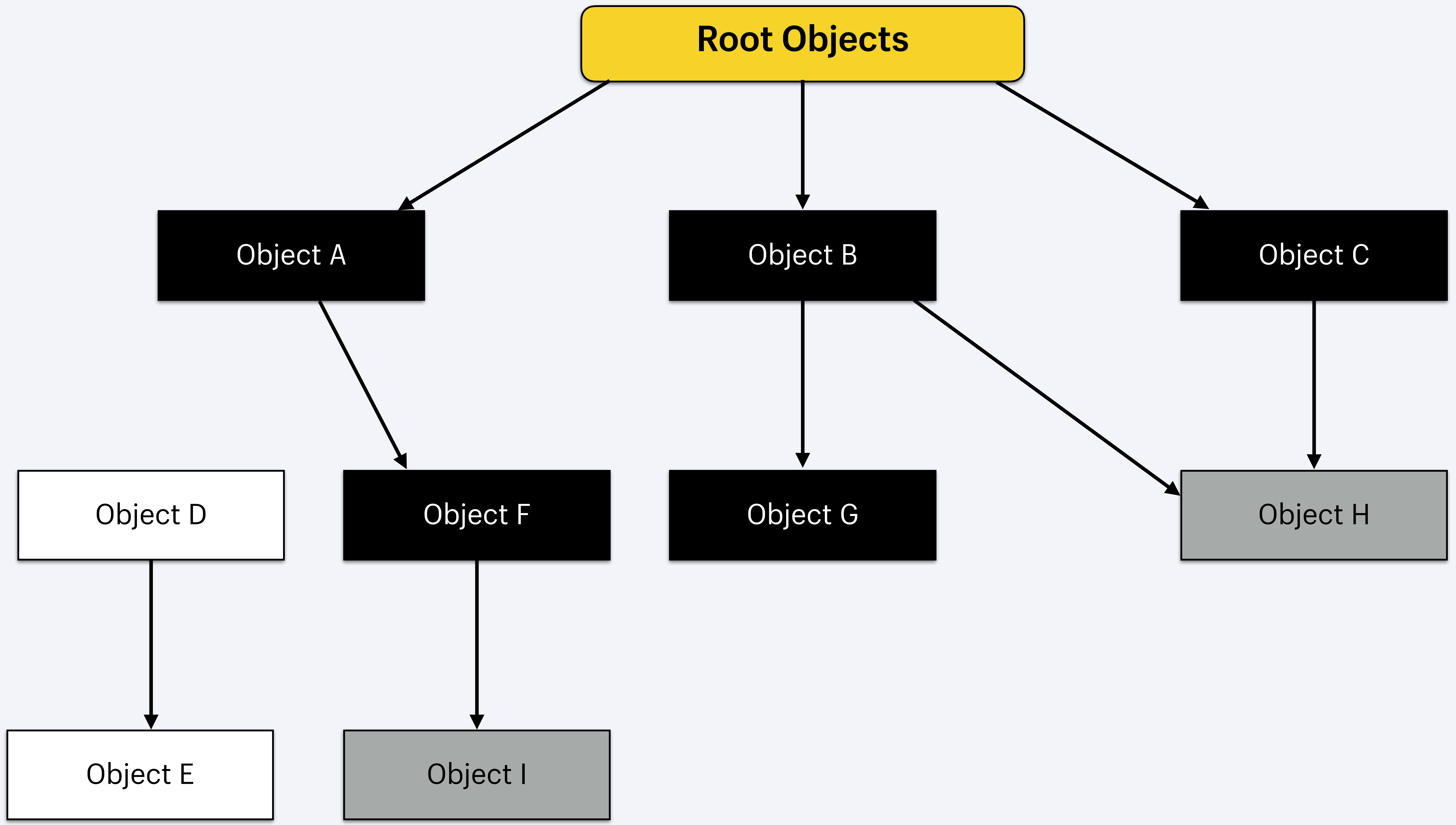

Root Objects
Object A
Object B
Object C
Object D
Object F
Object G
Object H
Object E
Object I

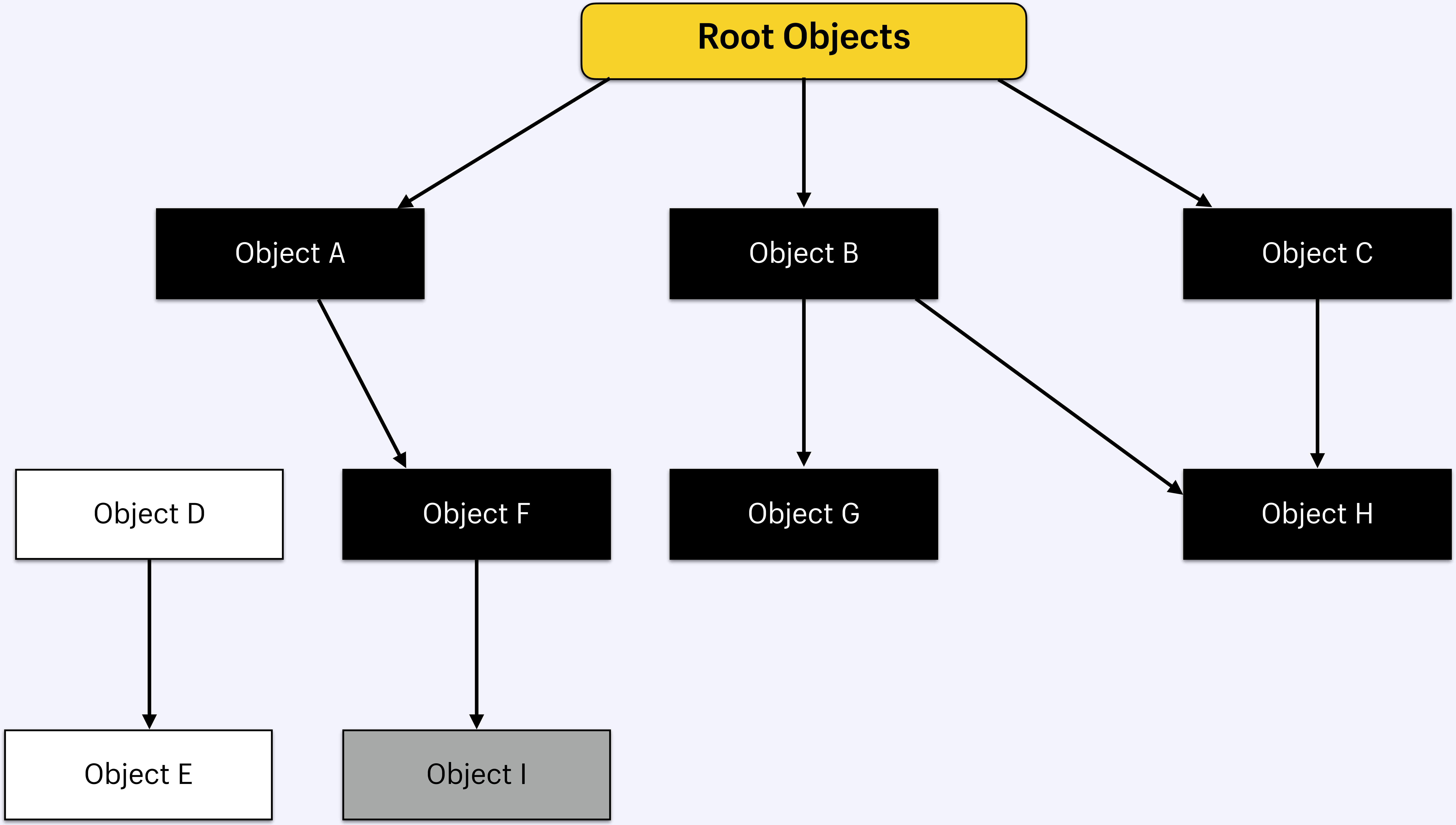

Root Objects
Object A
Object B
Object C
Object D
Object F
Object G
Object H
Object E
Object I

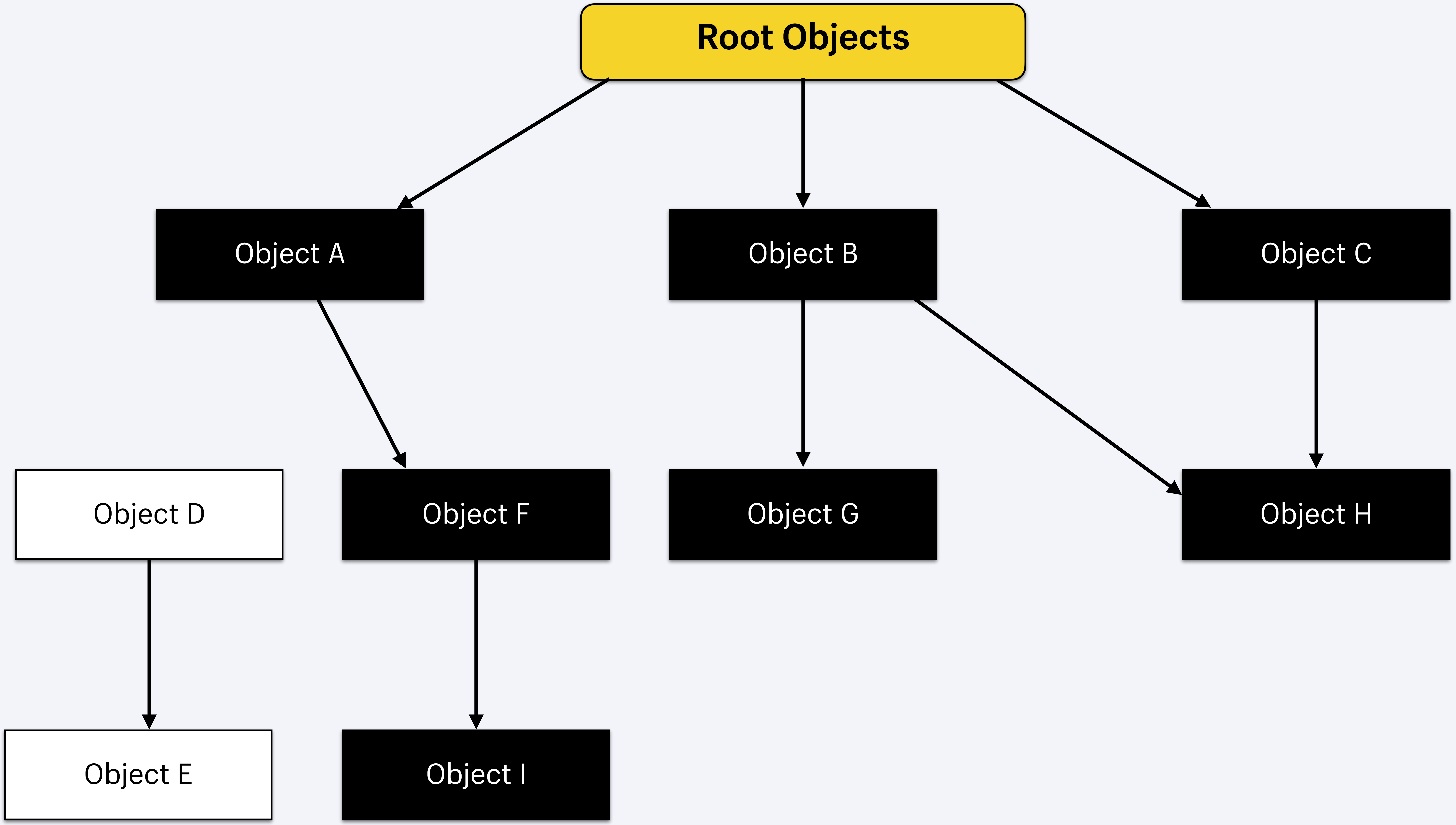

Root Objects
Object A
Object B
Object C
Object D
Object F
Object G
Object H
Object E
Object I


Root Objects
Object A
Object B
Object C
Object D
Object F
Object G
Object H
Object E
Object I


Root Objects
Object A
Object B
Object C
Object D
Object F
Object G
Object H
Object E
Object I

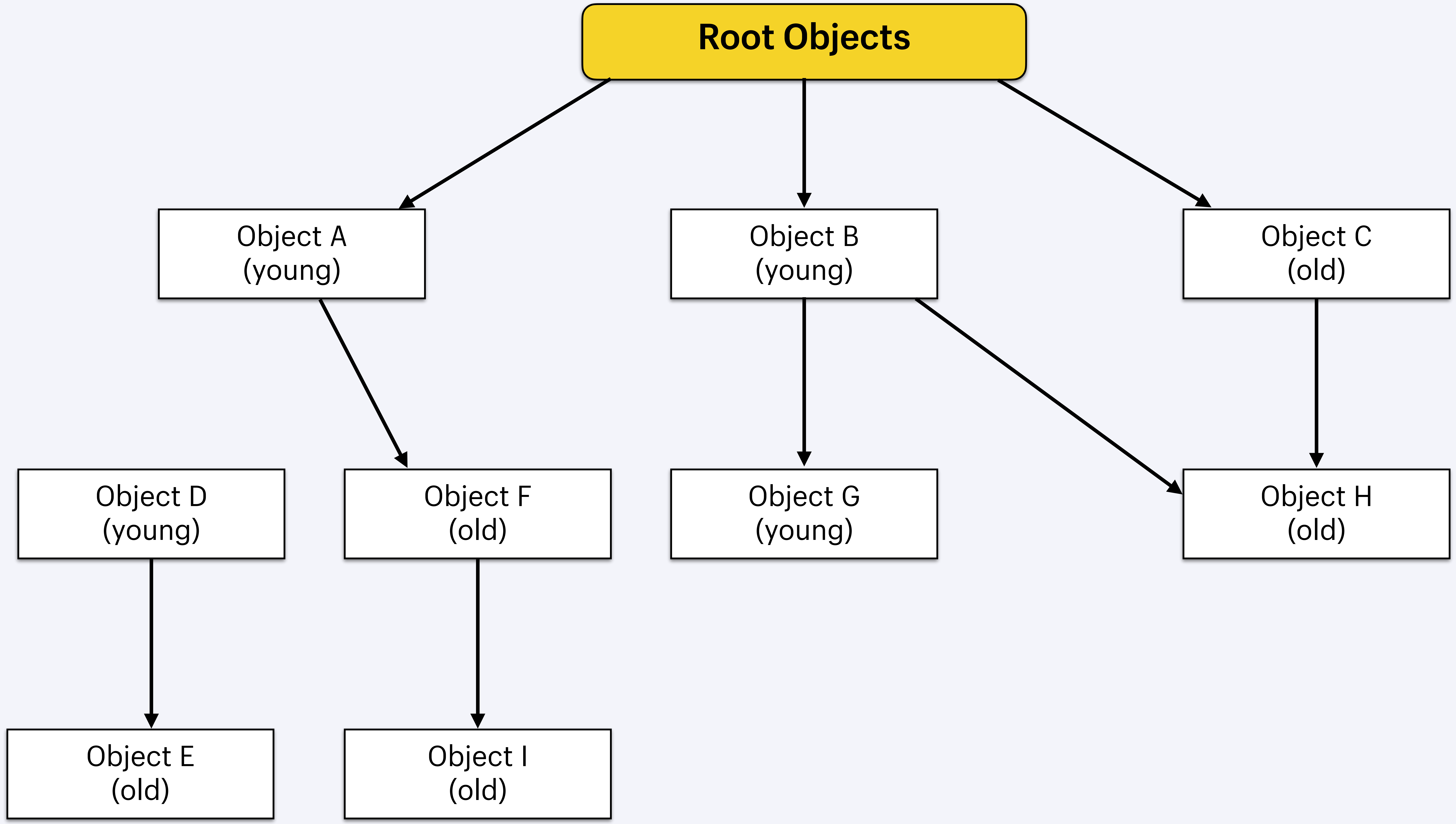

Root Objects
Object A
Object B
Object C
Object D
Object F
Object G
Object H
Object E
Object I

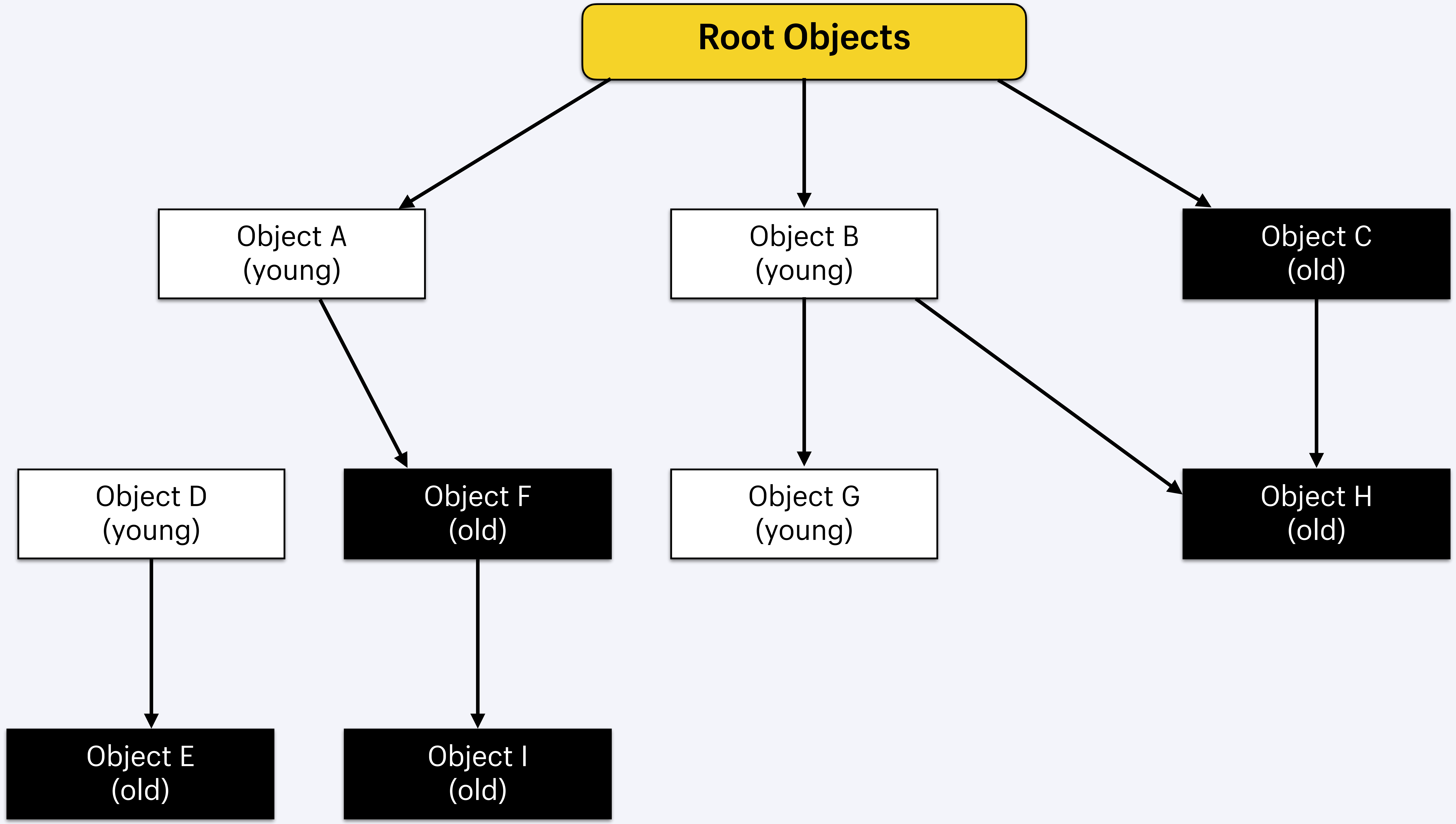

Root Objects
Object A
Object B
Object C
Object D
Object F
Object G
Object H
Object E
Object I

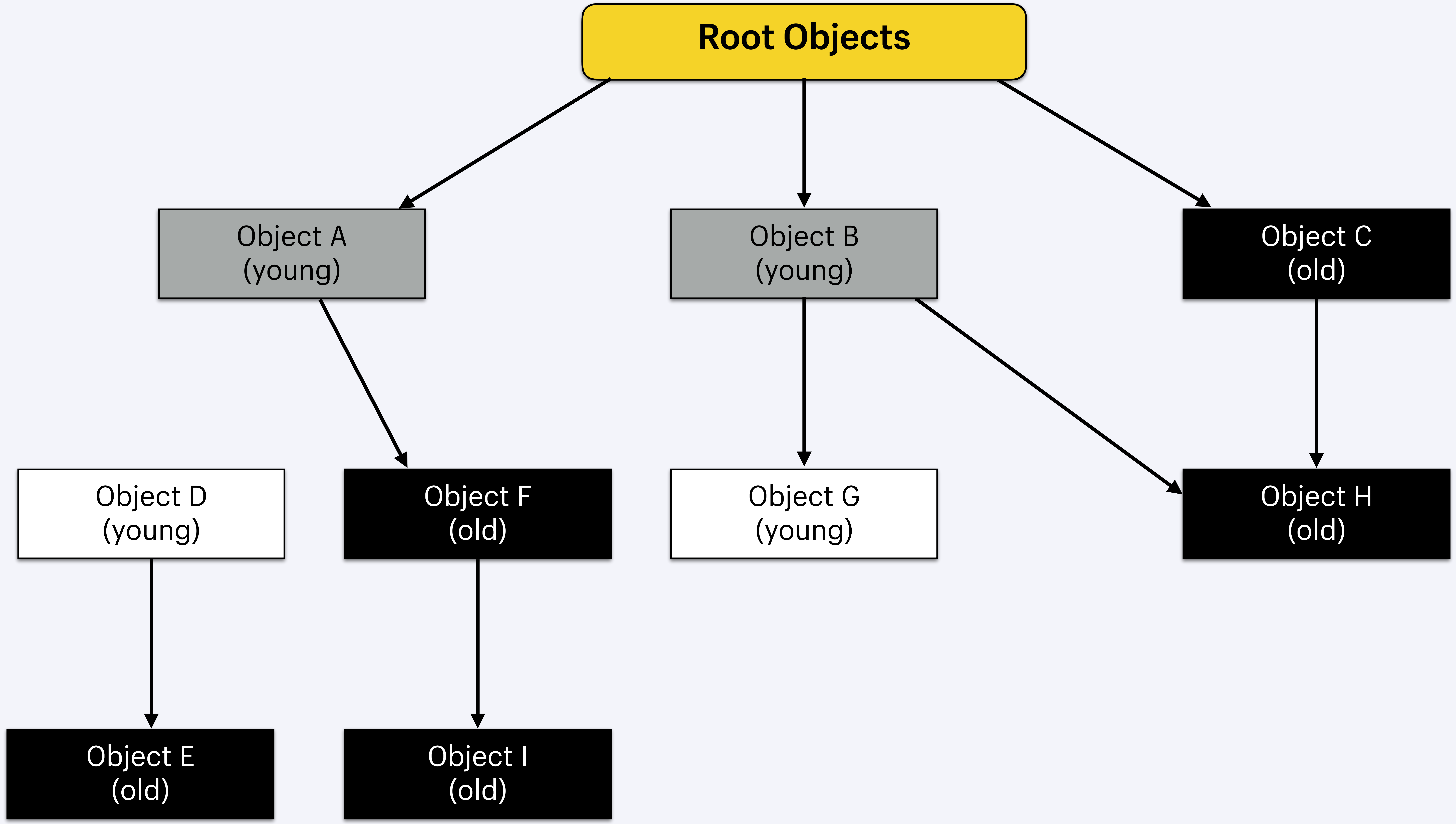

Root Objects
Object A
Object B
Object C
Object D
Object F
Object G
Object H
Object E
Object I

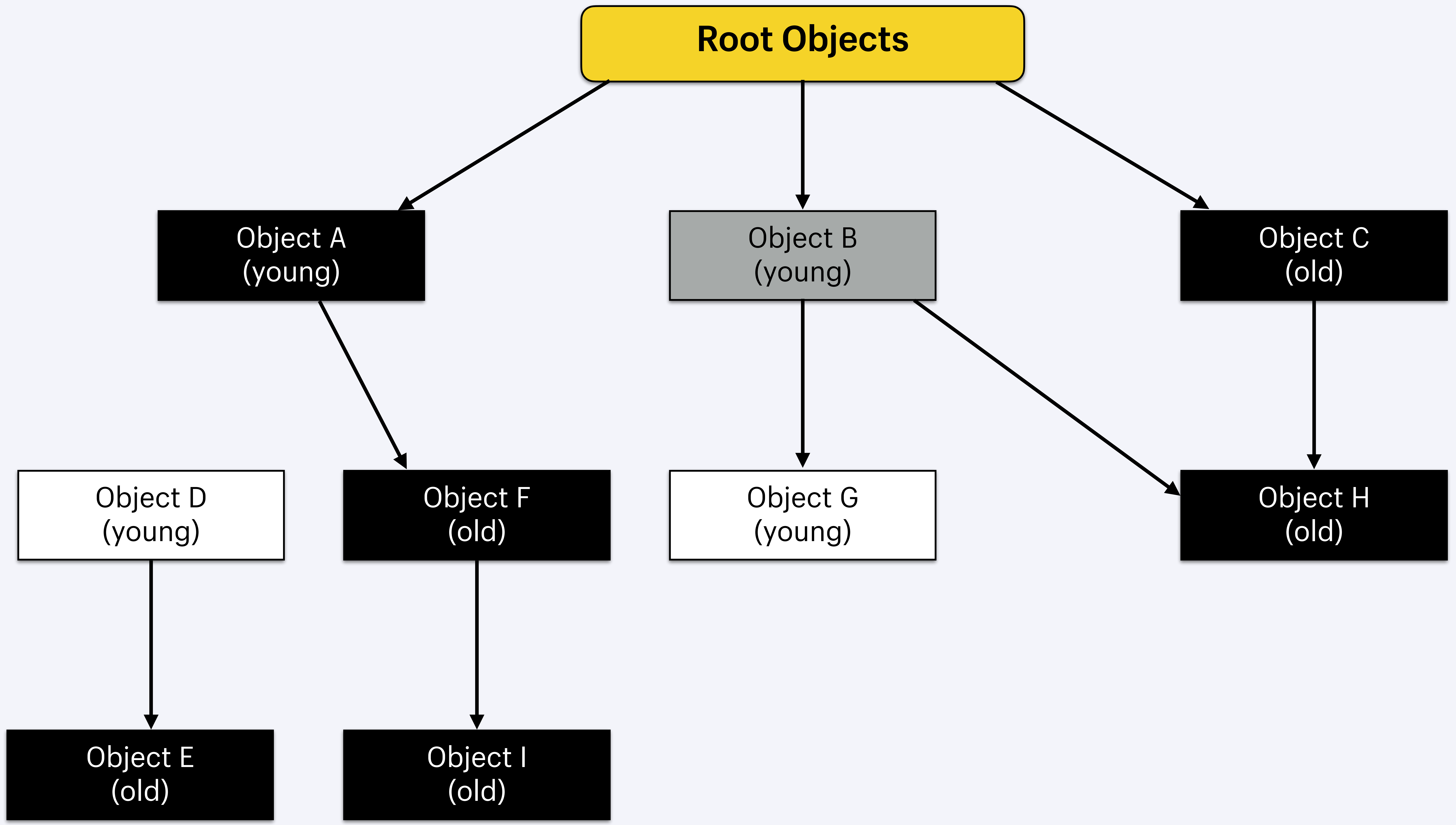
Generational garbage collector

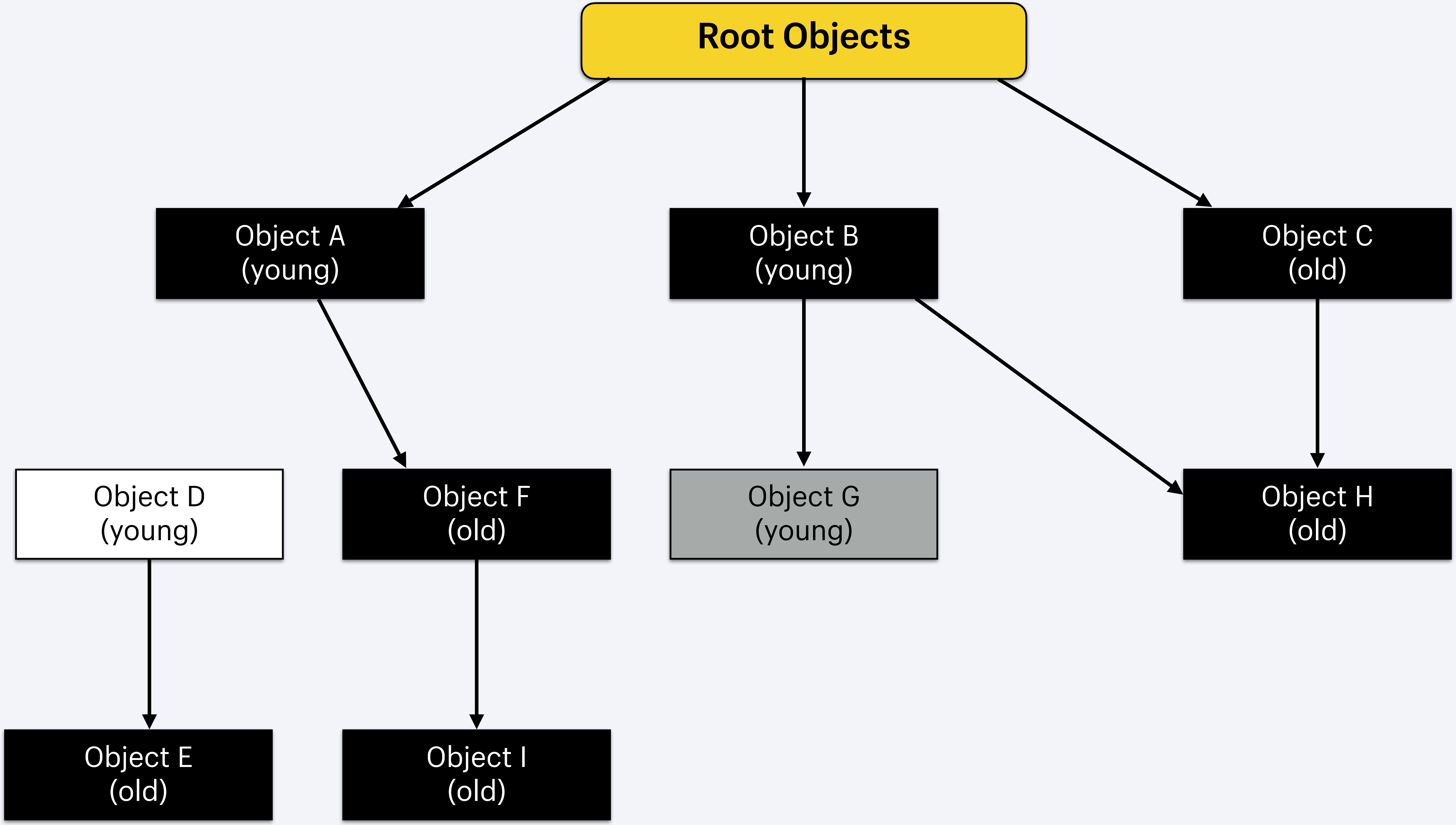
- Ruby uses a generational garbage collector
- Objects are either transient or immortal
- Newly created objects are in the young generation
- Long lived objects are promoted to the old generation
- Minor garbage collection cycles mark only young objects
- Major garbage collection cycles mark all objects

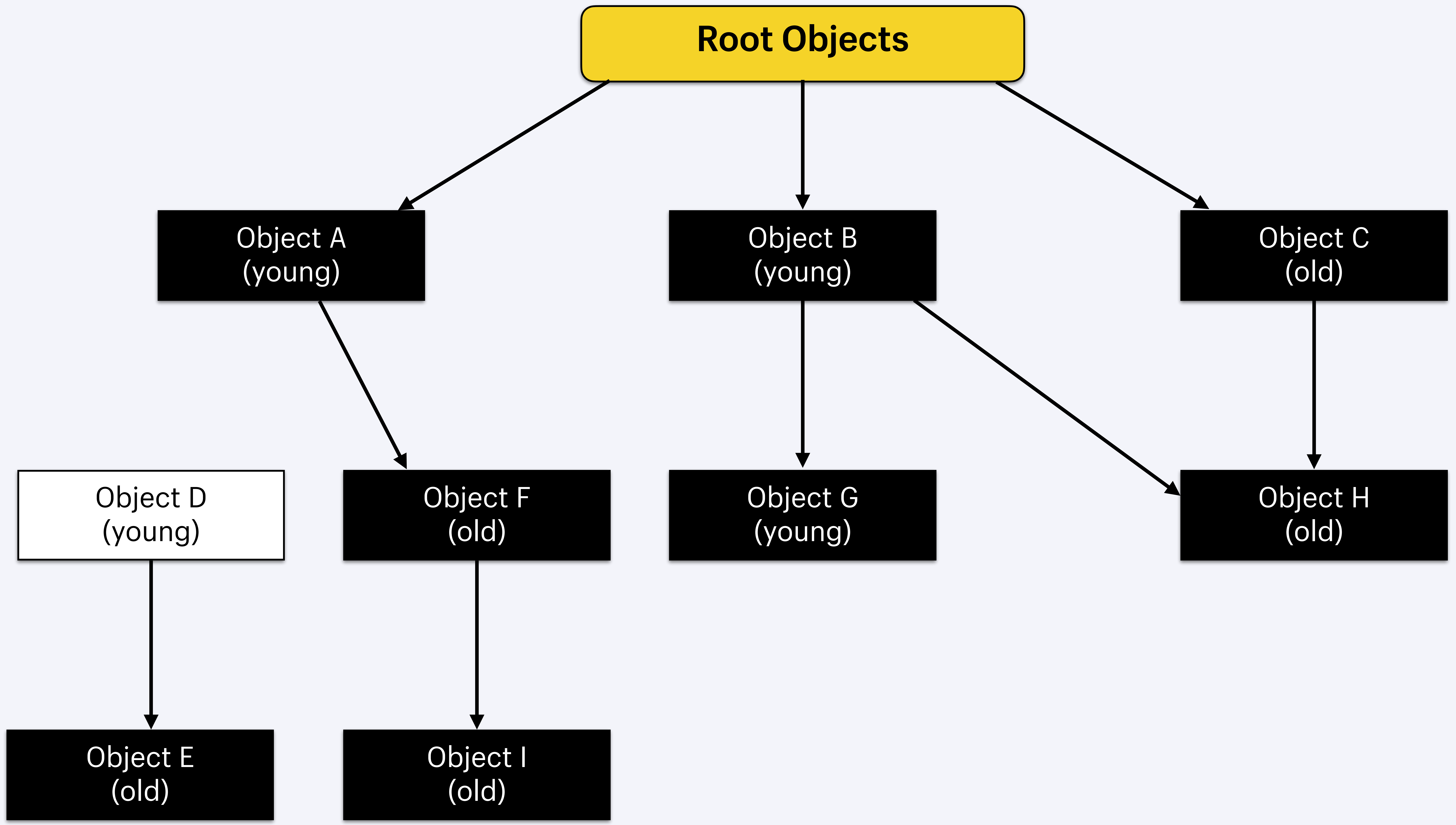

Root Objects
Object A (young)
Object B (young)
Object C (old)
Object D (young)
Object F (old)
Object G (young)
Object H (old)
Object E (old)
Object I (old)


Root Objects
Object A
(young)
Object B
(young)
Object C
(old)
Object D
(young)
Object F
(old)
Object G
(young)
Object H
(old)
Object E
(old)
Object I
(old)


Root Objects
Object A (young)
Object B (young)
Object C (old)
Object D (young)
Object F (old)
Object G (young)
Object H (old)
Object E (old)
Object I (old)

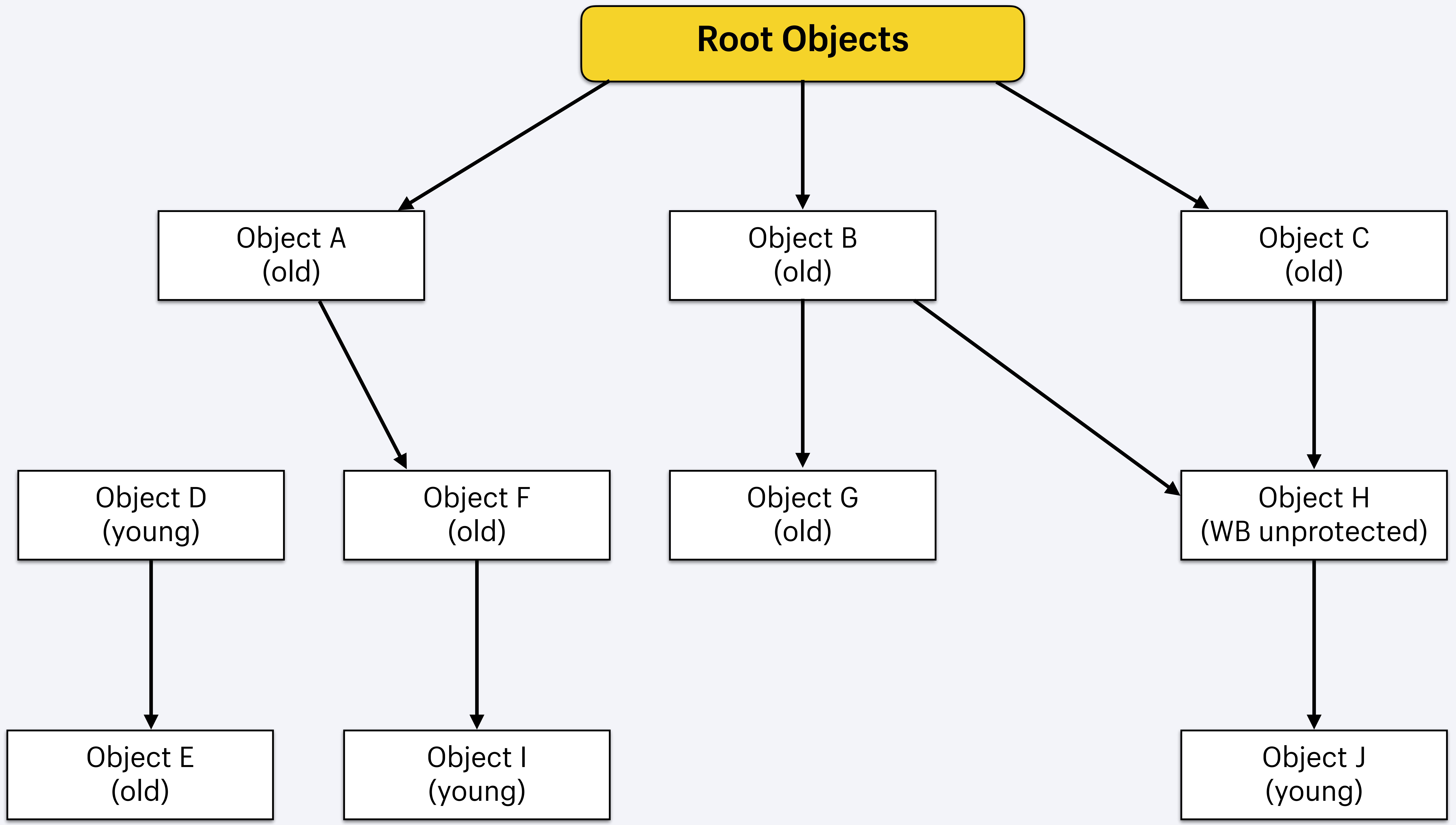

Root Objects
Object A (young)
Object B (young)
Object C (old)
Object D (young)
Object F (old)
Object G (young)
Object H (old)
Object E (old)
Object I (old)

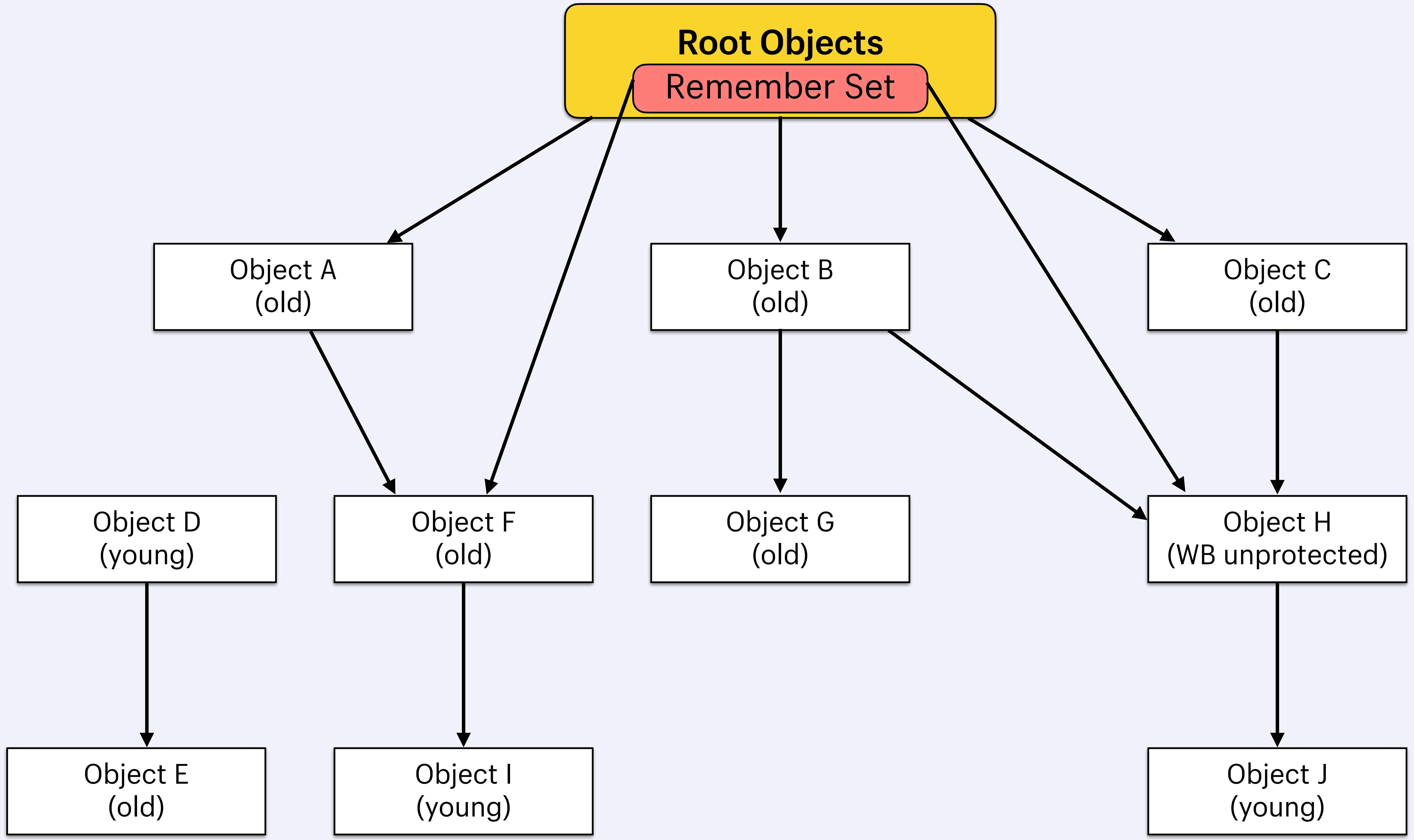

Root Objects
Object A (young)
Object B (young)
Object C (old)
Object D (young)
Object F (old)
Object G (young)
Object H (old)
Object E (old)
Object I (old)

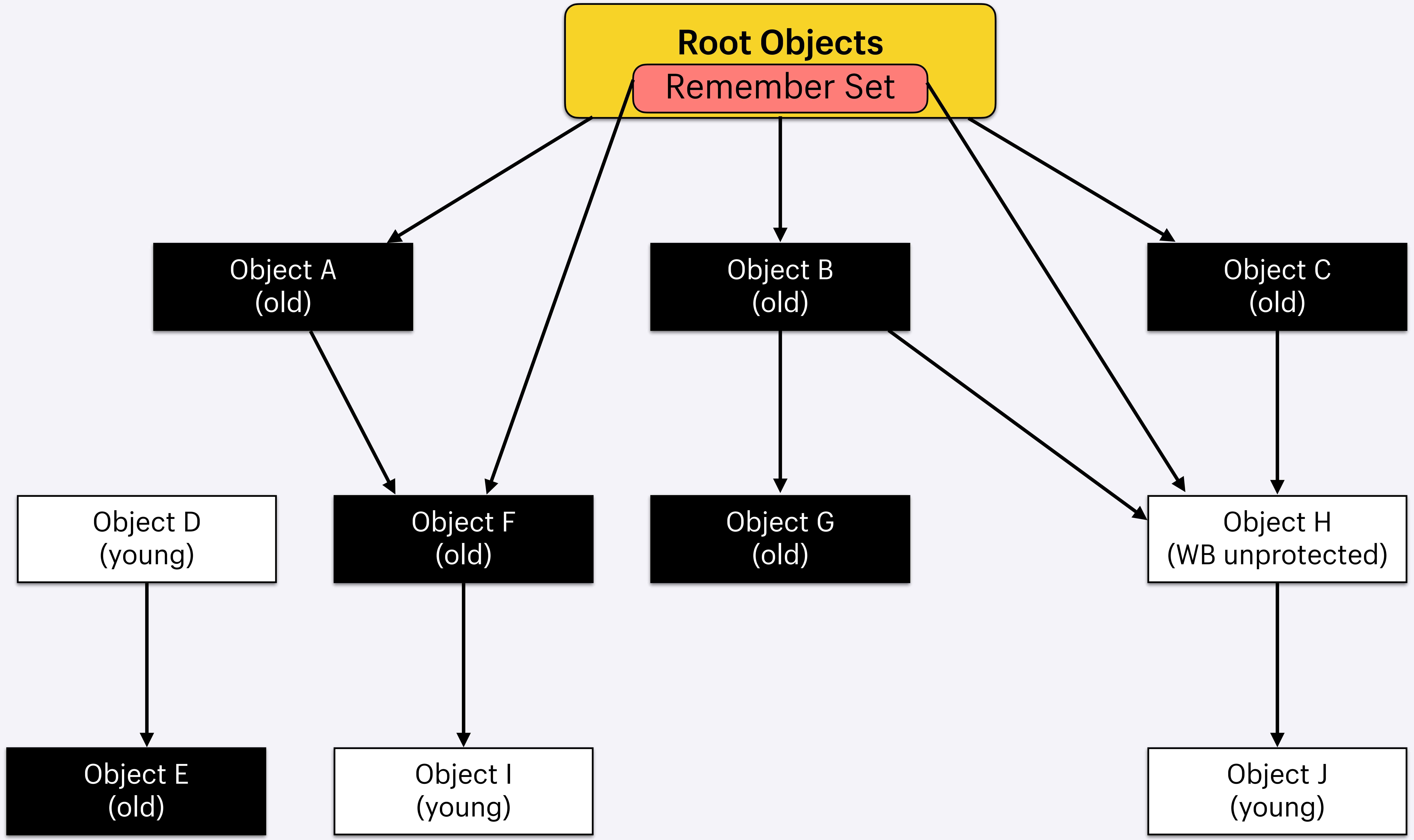

Root Objects
Object A (young)
Object B (young)
Object C (old)
Object D (young)
Object F (old)
Object G (young)
Object H (old)
Object E (old)
Object I (old)

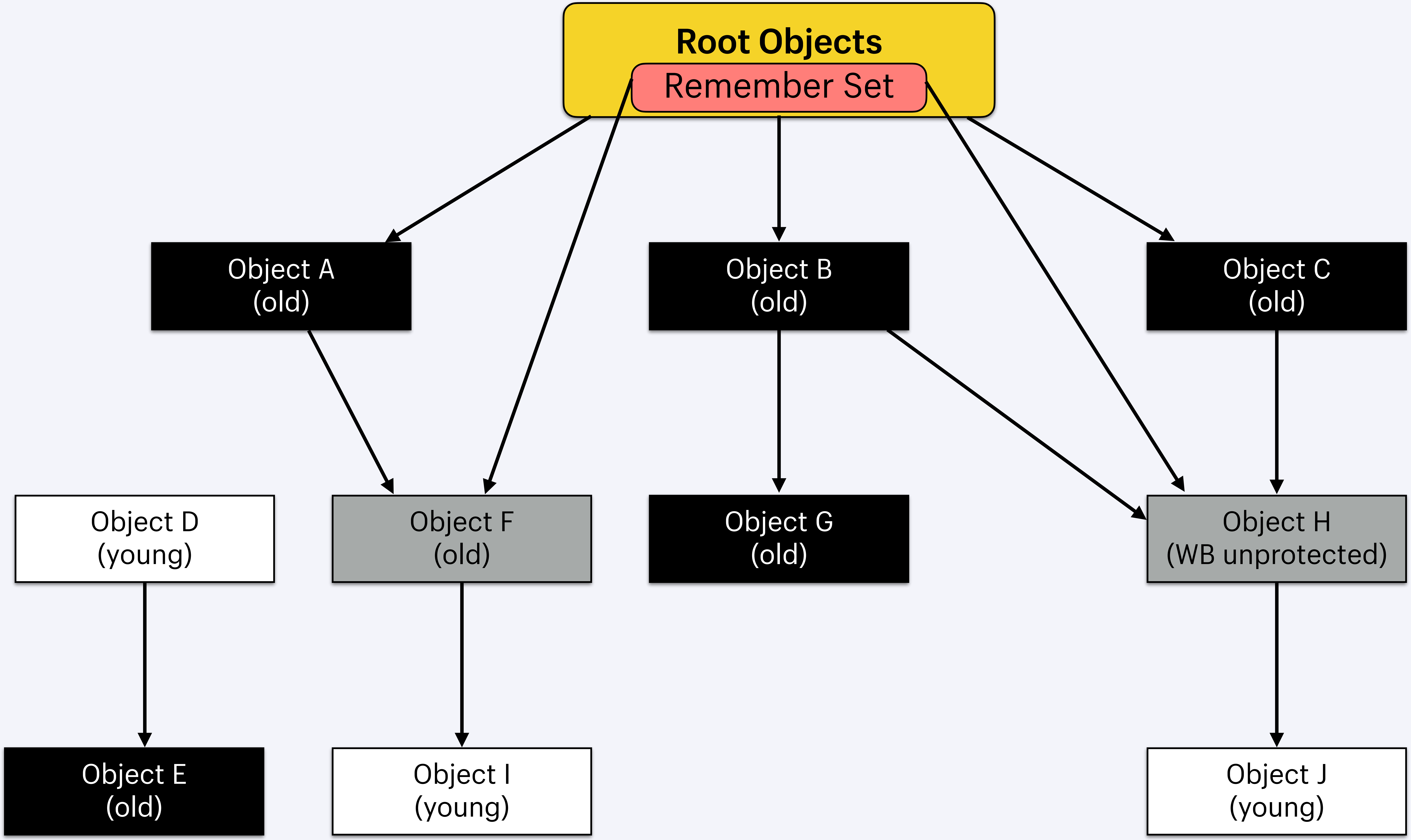
Generational GC challenges

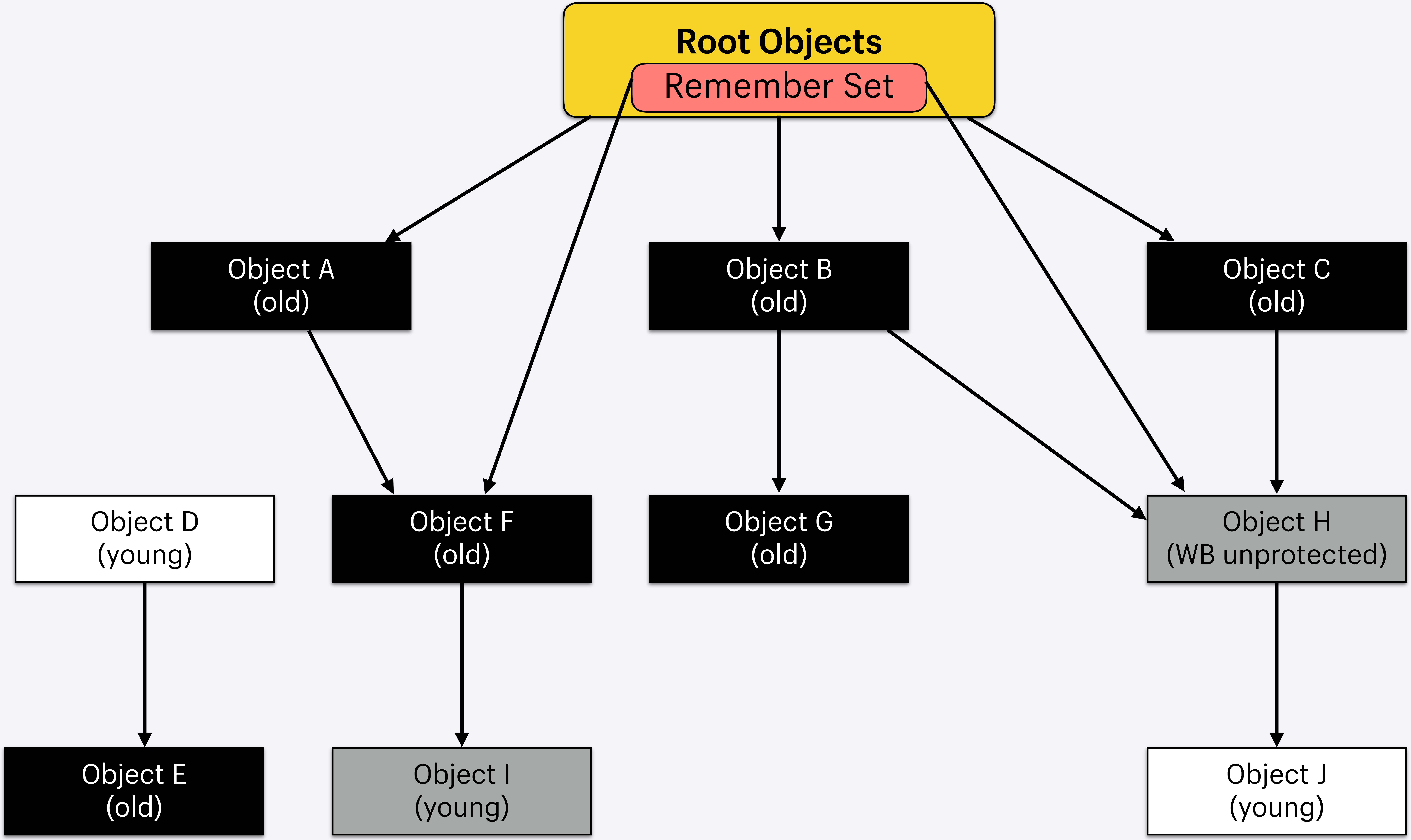
- What if we add a reference from an old object to a young object?
  - Need write barriers
  - Old object is placed in the remember set
  - Remember set marked during minor garbage collection cycles
- Objects that don’t support write barriers are called “write barrier unprotected”
- What about write barrier unprotected objects?
  - Also placed in the remember set

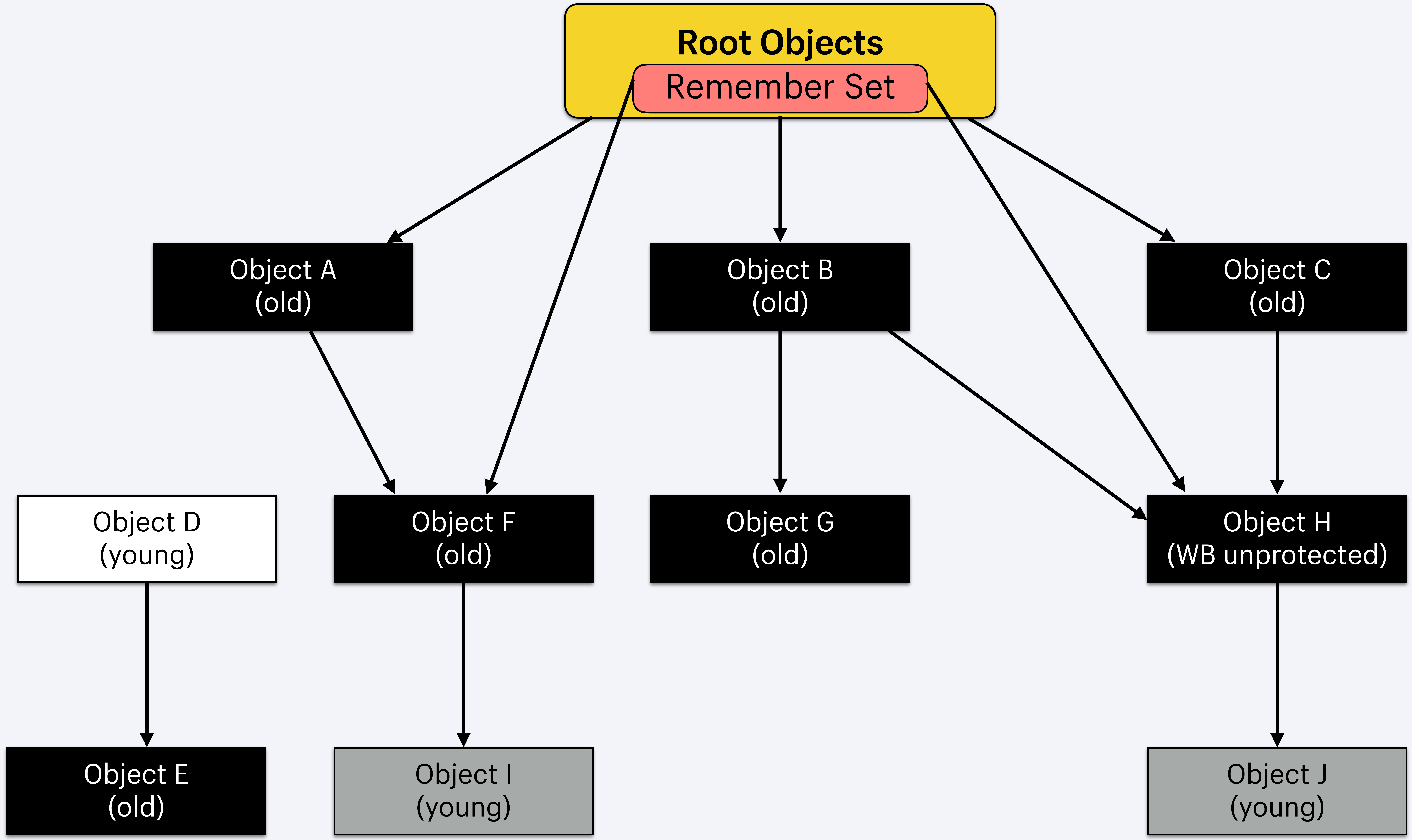

Root Objects
Object A
(old)
Object B
(old)
Object C
(old)
Object D
(young)
Object F
(old)
Object G
(old)
Object H
(WB unprotected)
Object E
(old)
Object I
(young)
Object J
(young)

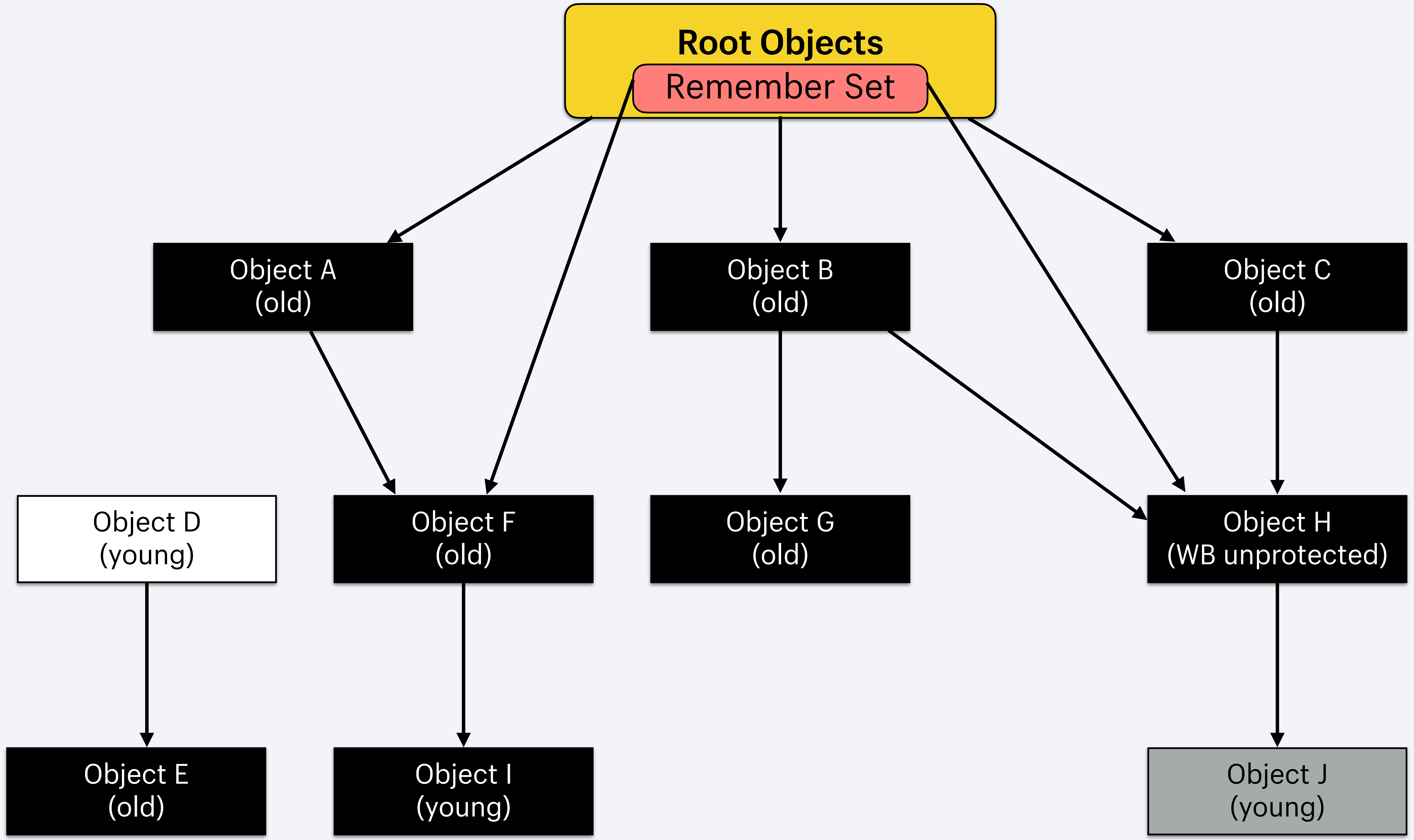

Root Objects
Remember Set
Object A (old)
Object B (old)
Object C (old)
Object D (young)
Object F (old)
Object G (old)
Object H (WB unprotected)
Object E (old)
Object I (young)
Object J (young)

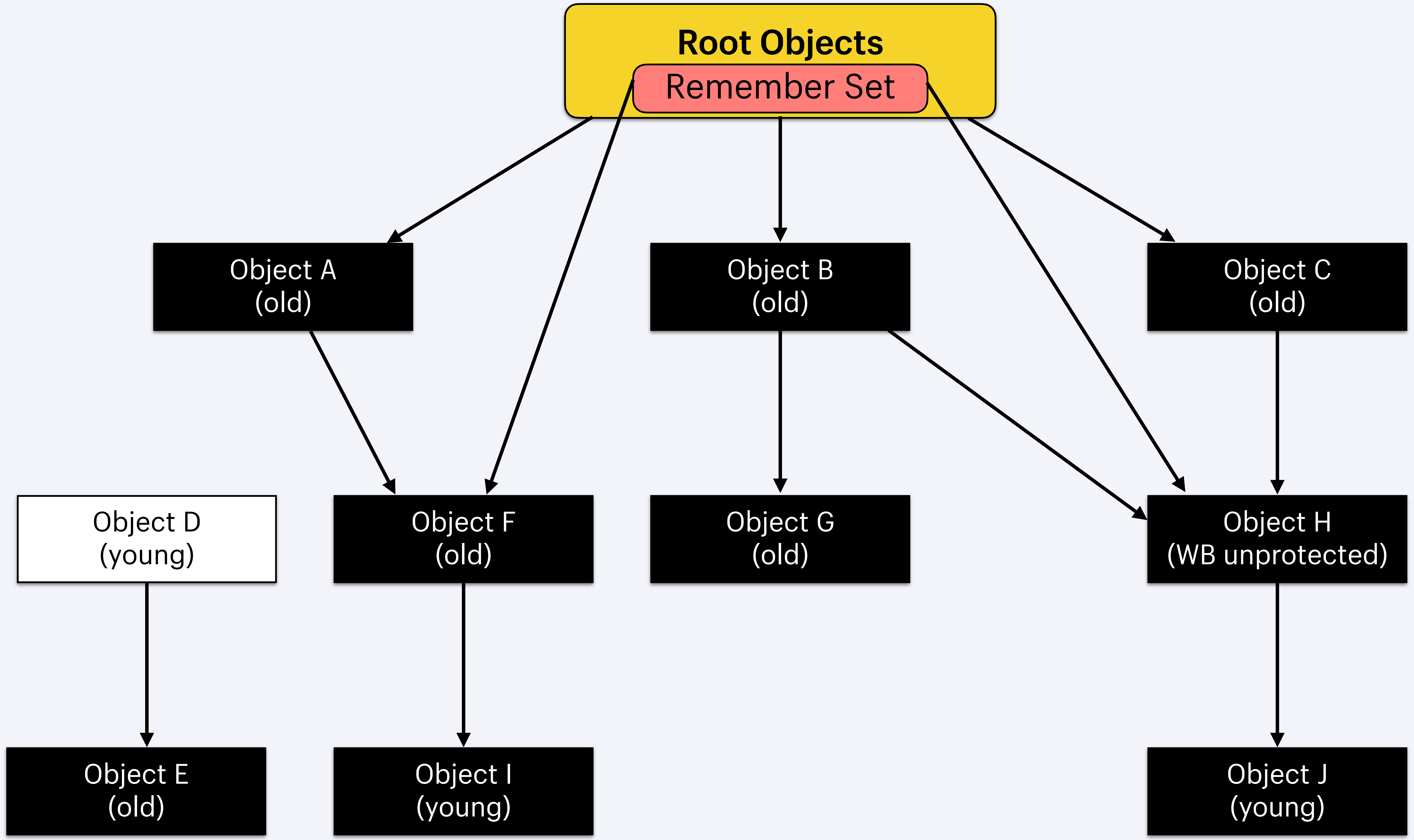

Root Objects
Remember Set
Object A (old)
Object B (old)
Object C (old)
Object D (young)
Object F (old)
Object G (old)
Object H (WB unprotected)
Object E (old)
Object I (young)
Object J (young)


Root Objects
Remember Set
Object A (old)
Object B (old)
Object C (old)
Object D (young)
Object F (old)
Object G (old)
Object H (WB unprotected)
Object E (old)
Object I (young)
Object J (young)


Root Objects
Remember Set
Object A
(old)
Object B
(old)
Object C
(old)
Object D
(young)
Object F
(old)
Object G
(old)
Object H
(WB unprotected)
Object E
(old)
Object I
(young)
Object J
(young)

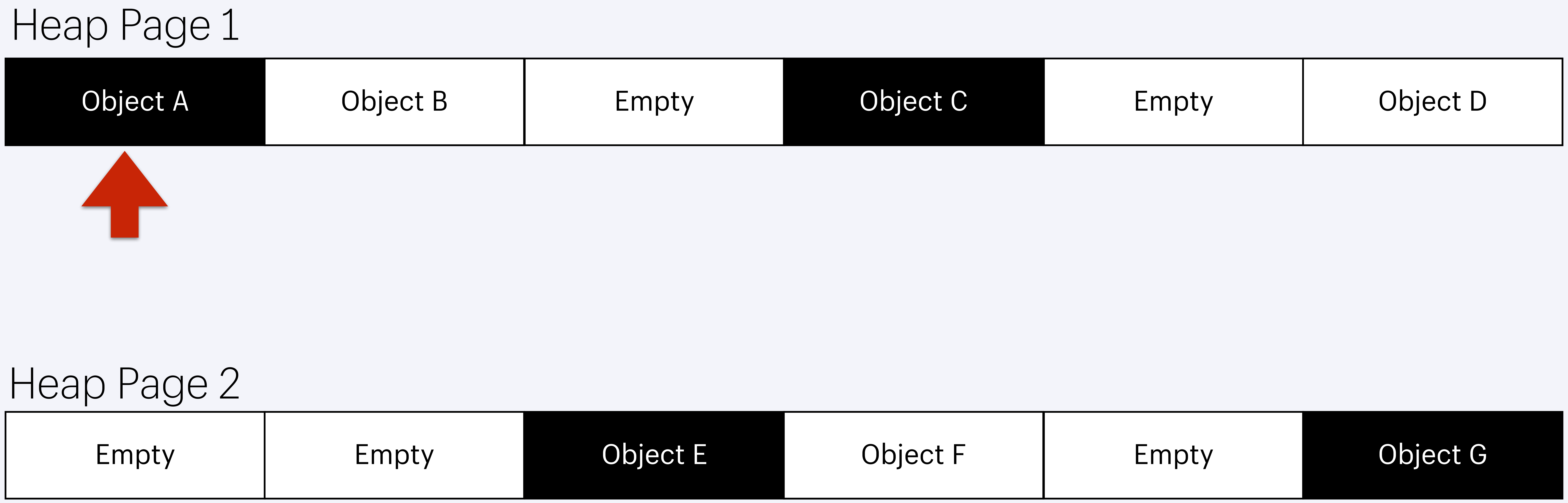

Root Objects
Remember Set
Object A (old)
Object B (old)
Object C (old)
Object D (young)
Object F (old)
Object G (old)
Object H (WB unprotected)
Object E (old)
Object I (young)
Object J (young)


Root Objects
Remember Set
Object A (old)
Object B (old)
Object C (old)
Object D (young)
Object F (old)
Object G (old)
Object H (WB unprotected)
Object E (old)
Object I (young)
Object J (young)


Root Objects
Remember Set
Object A (old)
Object B (old)
Object C (old)
Object D (young)
Object F (old)
Object G (old)
Object H (WB unprotected)
Object E (old)
Object I (young)
Object J (young)

Sweeping phase

- Marking phase determined the liveliness of every object
- Unmarked objects are dead
- Sweeping phase frees the resources of all dead objects

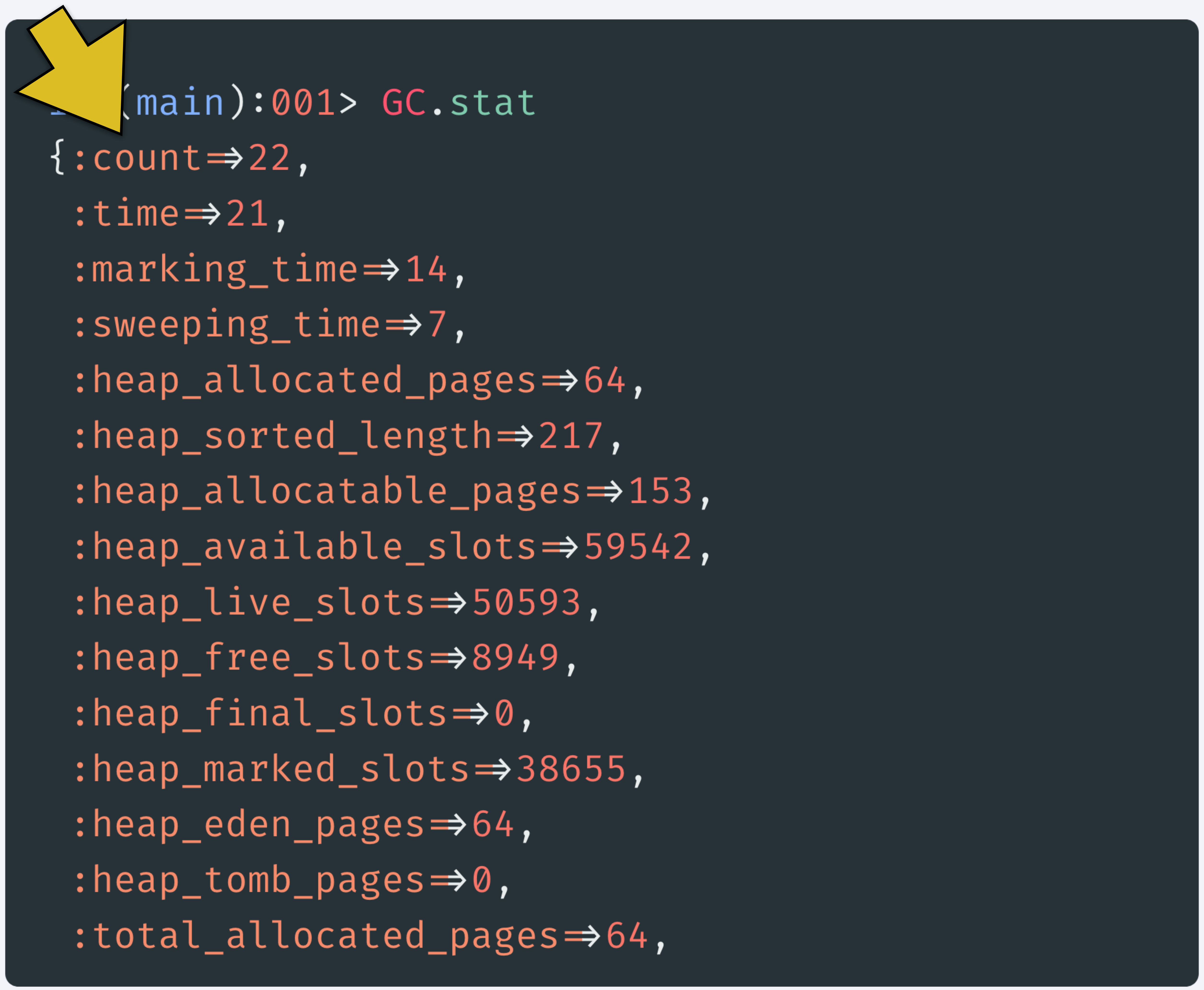

Heap Page 1
Object A
Object B
Empty
Object C
Empty
Object D
Heap Page 2
Empty
Empty
Object E
Object F
Empty
Object G

Compaction phase

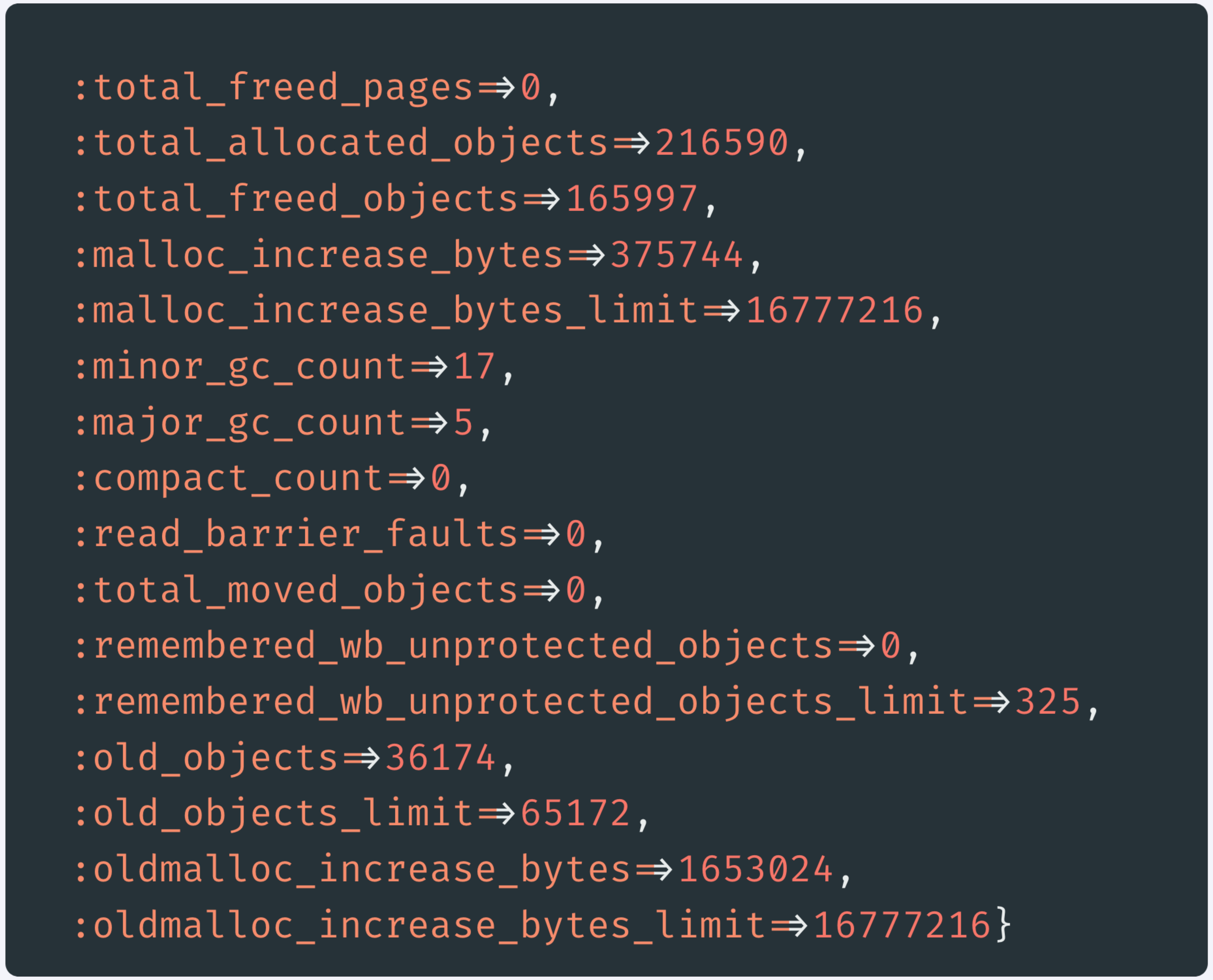
- Reduces fragmentation in the Ruby heap by moving objects
- Moves objects to the optimal size
- Optional and can be enabled by calling *GC.compact* or setting *GC.auto_compact = true*

blog.peterzhu.ca/notes-on-ruby-gc/

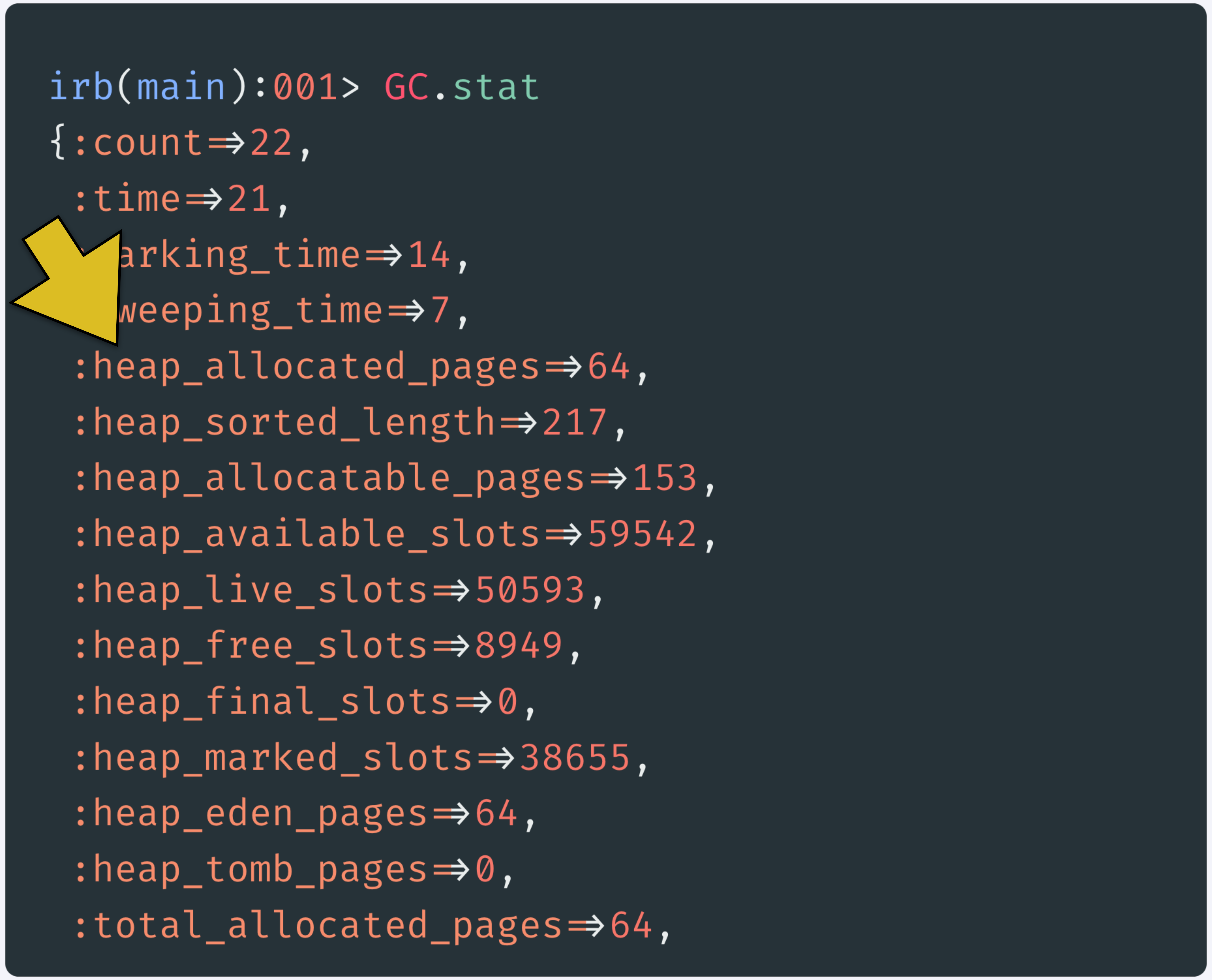
Collecting metrics



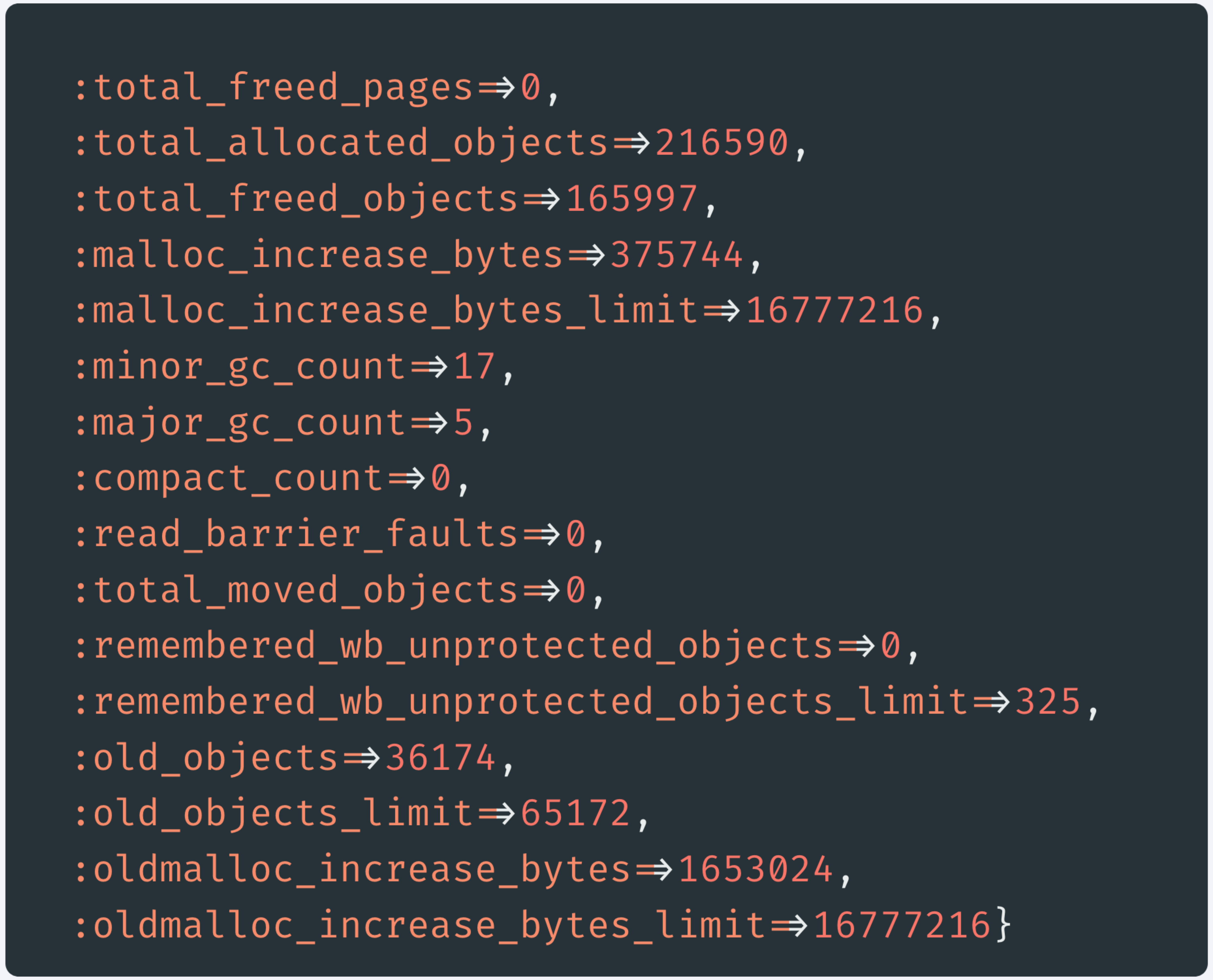
```
(main):001> GC.stat
{:count=>22,
 :time=>21,
 :marking_time=>14,
 :sweeping_time=>7,
 :heap_allocated_pages=>64,
 :heap_sorted_length=>217,
 :heap_allocatable_pages=>153,
 :heap_available_slots=>59542,
 :heap_live_slots=>50593,
 :heap_free_slots=>8949,
 :heap_final_slots=>0,
 :heap_marked_slots=>38655,
 :heap_eden_pages=>64,
 :heap_tomb_pages=>0,
 :total_allocated_pages=>64,
```



```
 :total_freed_pages=>0,
 :total_allocated_objects=>216590,
 :total_freed_objects=>165997,
 :malloc_increase_bytes=>375744,
 :malloc_increase_bytes_limit=>16777216,
 :minor_gc_count=>17,
 :major_gc_count=>5,
 :compact_count=>0,
 :read_barrier_faults=>0,
 :total_moved_objects=>0,
 :remembered_wb_unprotected_objects=>0,
 :remembered_wb_unprotected_objects_limit=>325,
 :old_objects=>36174,
 :old_objects_limit=>65172,
 :oldmalloc_increase_bytes=>1653024,
 :oldmalloc_increase_bytes_limit=>16777216}
```



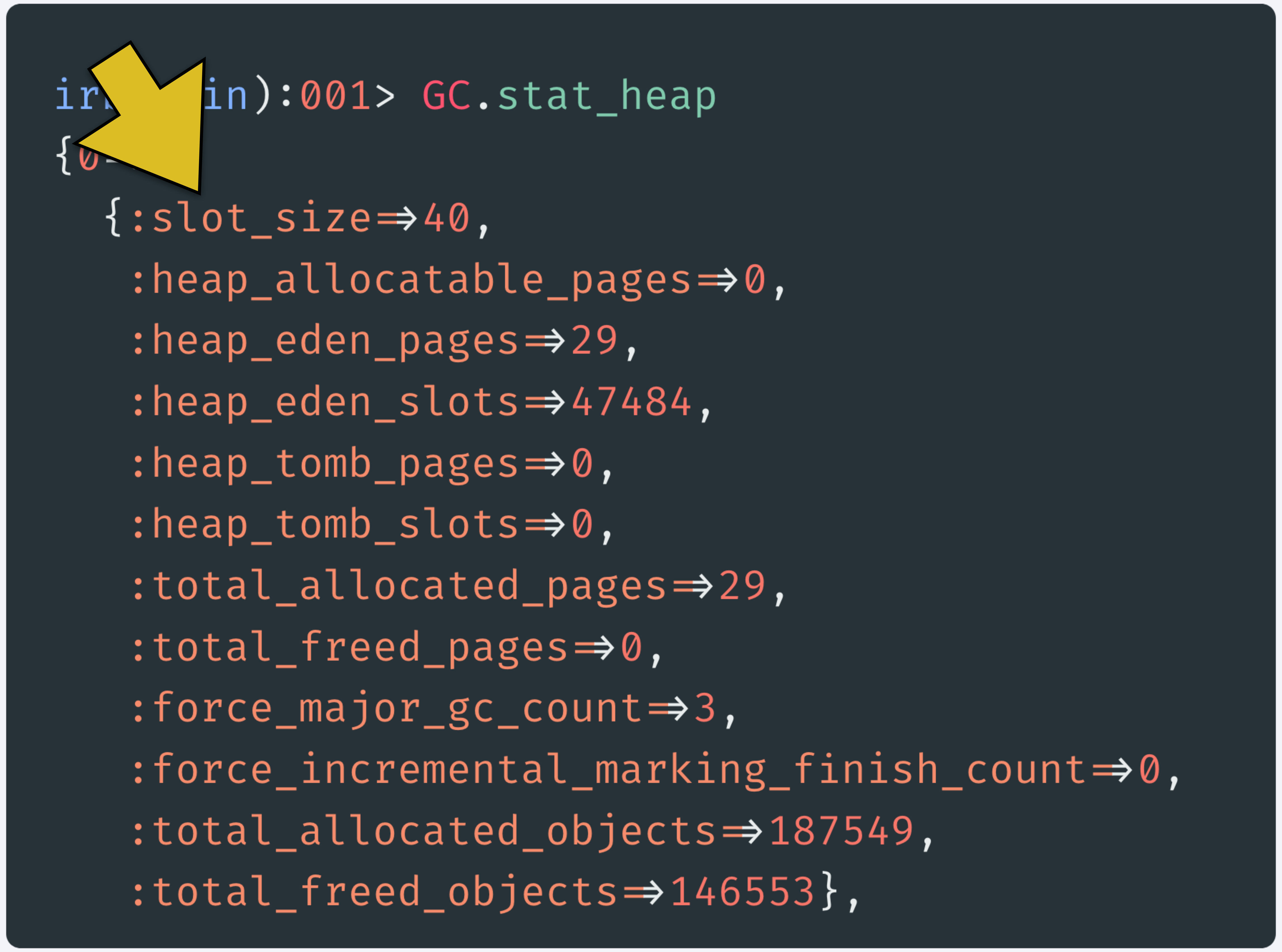
```
irb(main):001> GC.stat
{:count=>22,
 :time=>21,
  arking_time=>14,
  weeping_time=>7,
 :heap_allocated_pages=>64,
 :heap_sorted_length=>217,
 :heap_allocatable_pages=>153,
 :heap_available_slots=>59542,
 :heap_live_slots=>50593,
 :heap_free_slots=>8949,
 :heap_final_slots=>0,
 :heap_marked_slots=>38655,
 :heap_eden_pages=>64,
 :heap_tomb_pages=>0,
 :total_allocated_pages=>64,
```



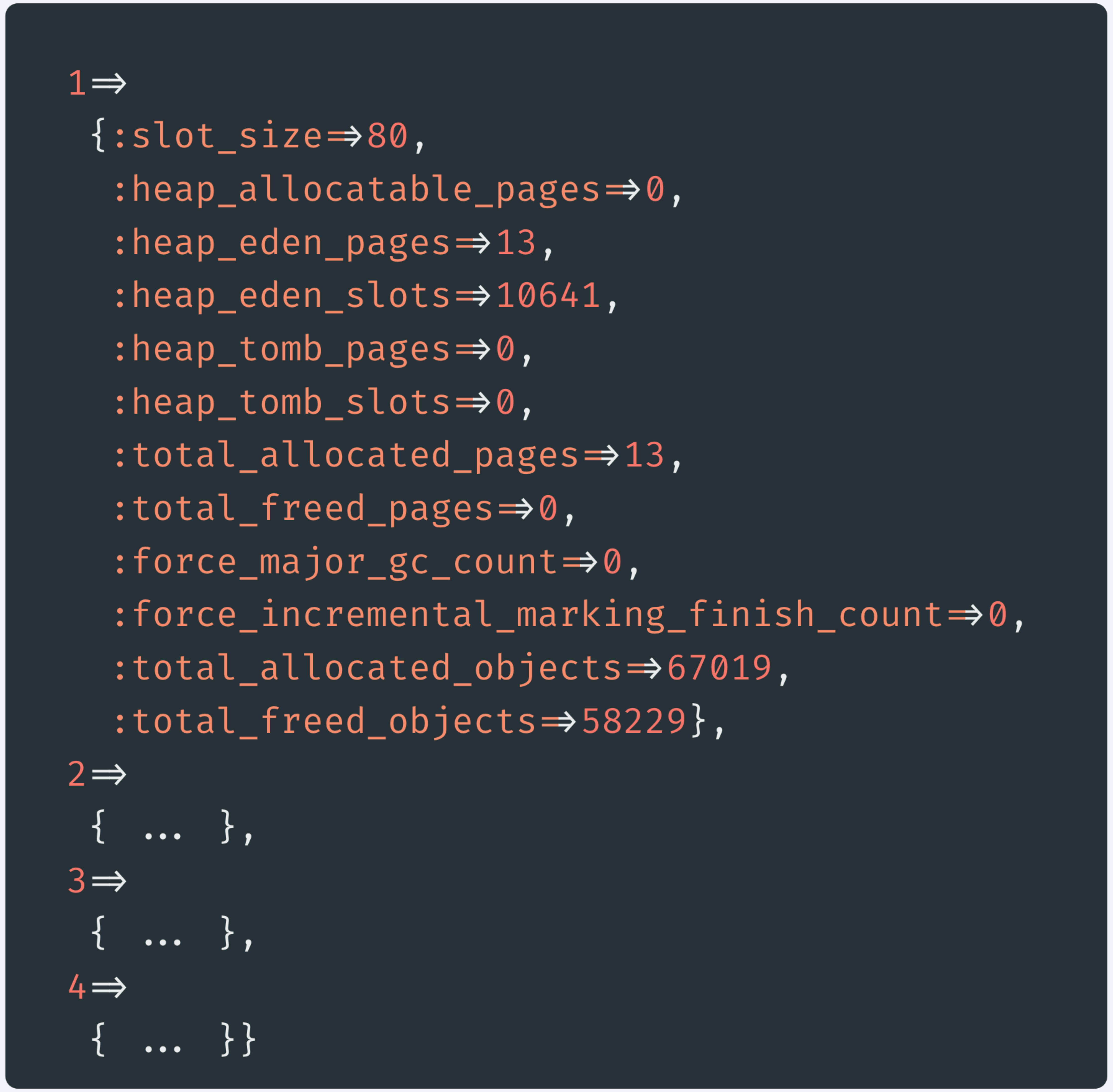
```
 :total_freed_pages=>0,
 :total_allocated_objects=>216590,
 :total_freed_objects=>165997,
 :malloc_increase_bytes=>375744,
 :malloc_increase_bytes_limit=>16777216,
 :minor_gc_count=>17,
 :major_gc_count=>5,
 :compact_count=>0,
 :read_barrier_faults=>0,
 :total_moved_objects=>0,
 :remembered_wb_unprotected_objects=>0,
 :remembered_wb_unprotected_objects_limit=>325,
 :old_objects=>36174,
 :old_objects_limit=>65172,
 :oldmalloc_increase_bytes=>1653024,
 :oldmalloc_increase_bytes_limit=>16777216}
```

```
irb(main):001> GC.stat
{:count=>22,
 :time=>21,
 :marking_time=>14,
 :sweeping_time=>7,
 :heap_allocated_pages=>64,
 :heap_sorted_length=>217,
 :heap_allocatable_pages=>153,
 :heap_available_slots=>59542,
 :heap_live_slots=>50593,
 :heap_free_slots=>8949,
 :heap_final_slots=>0,
 :heap_marked_slots=>38655,
 :heap_eden_pages=>64,
 :heap_tomb_pages=>0,
 :total_allocated_pages=>64,
```

```
  otal_freed_pages=>0,
 :total_allocated_objects=>216590,
 :total_freed_objects=>165997,
 :malloc_increase_bytes=>375744,
 :malloc_increase_bytes_limit=>16777216,
 :minor_gc_count=>17,
 :major_gc_count=>5,
 :compact_count=>0,
 :read_barrier_faults=>0,
 :total_moved_objects=>0,
 :remembered_wb_unprotected_objects=>0,
 :remembered_wb_unprotected_objects_limit=>325,
 :old_objects=>36174,
 :old_objects_limit=>65172,
 :oldmalloc_increase_bytes=>1653024,
 :oldmalloc_increase_bytes_limit=>16777216}
```



```
ir[illegible]in):001> GC.stat_heap
{0[illegible]
  {:slot_size=>40,
   :heap_allocatable_pages=>0,
   :heap_eden_pages=>29,
   :heap_eden_slots=>47484,
   :heap_tomb_pages=>0,
   :heap_tomb_slots=>0,
   :total_allocated_pages=>29,
   :total_freed_pages=>0,
   :force_major_gc_count=>3,
   :force_incremental_marking_finish_count=>0,
   :total_allocated_objects=>187549,
   :total_freed_objects=>146553},
```



```
1=>
 {:slot_size=>80,
  :heap_allocatable_pages=>0,
  :heap_eden_pages=>13,
  :heap_eden_slots=>10641,
  :heap_tomb_pages=>0,
  :heap_tomb_slots=>0,
  :total_allocated_pages=>13,
  :total_freed_pages=>0,
  :force_major_gc_count=>0,
  :force_incremental_marking_finish_count=>0,
  :total_allocated_objects=>67019,
  :total_freed_objects=>58229},
2=>
 { ... },
3=>
 { ... },
4=>
 { ... }}
```

GC tuning

Decreasing object allocations

- Less pressure on the garbage collector
- Faster marking and sweeping phases
- Find and optimize controllers that allocates lots of objects

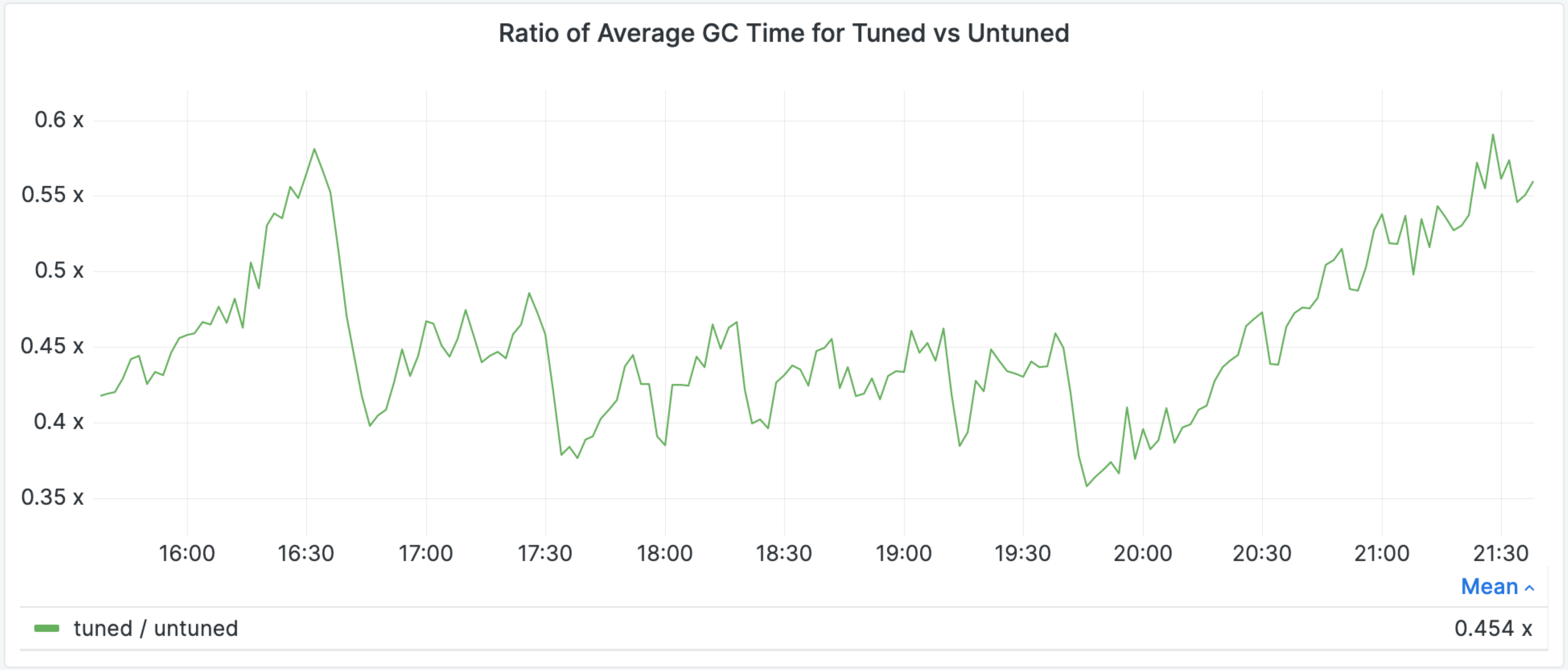
Reducing garbage collection cycles

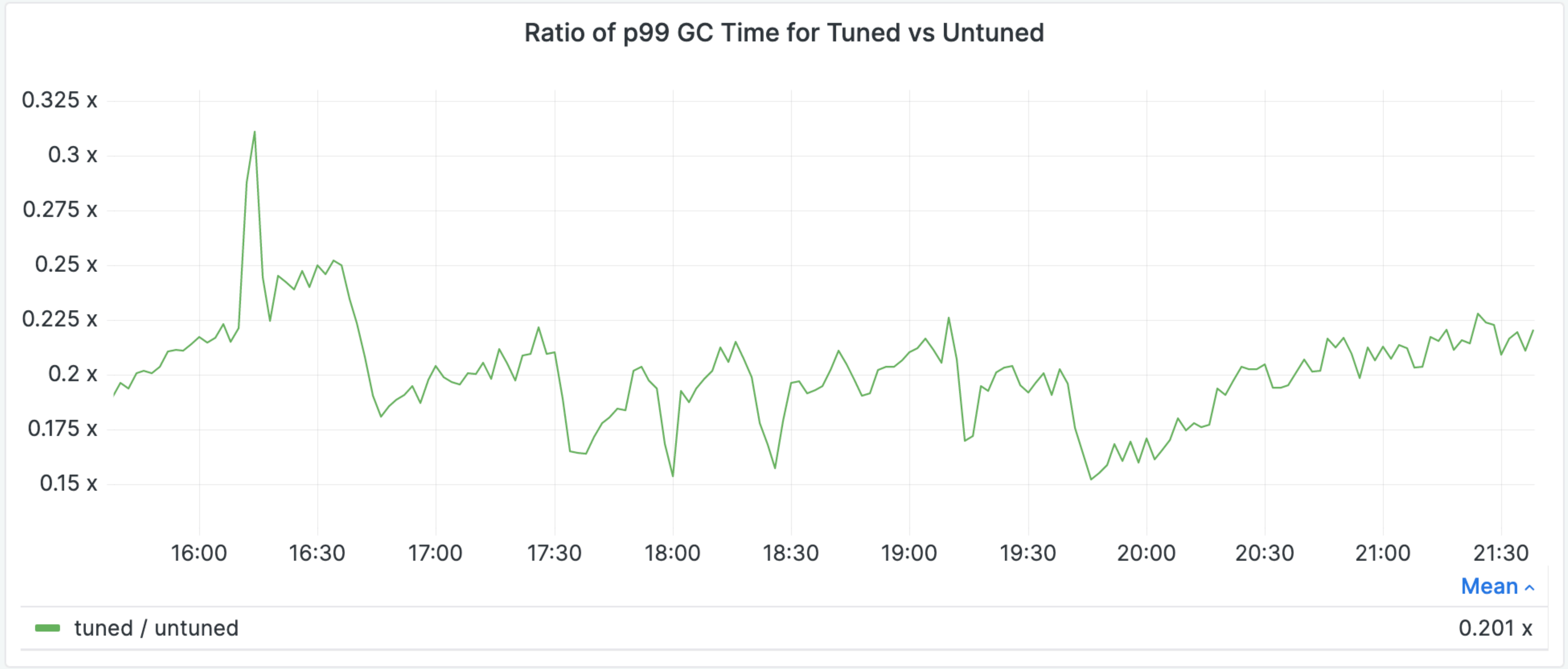
- Reduce the number of major garbage collection cycles
- Reduce GC cycles at boot using *RUBY_GC_HEAP_{0,1,2,3,4}_INIT_SLOTS* (3.3+) or *RUBY_GC_HEAP_INIT_SLOTS* environment variable
- Increase *RUBY_GC_OLDMALLOC_LIMIT* and *RUBY_GC_OLDMALLOC_LIMIT_MAX* environment variables

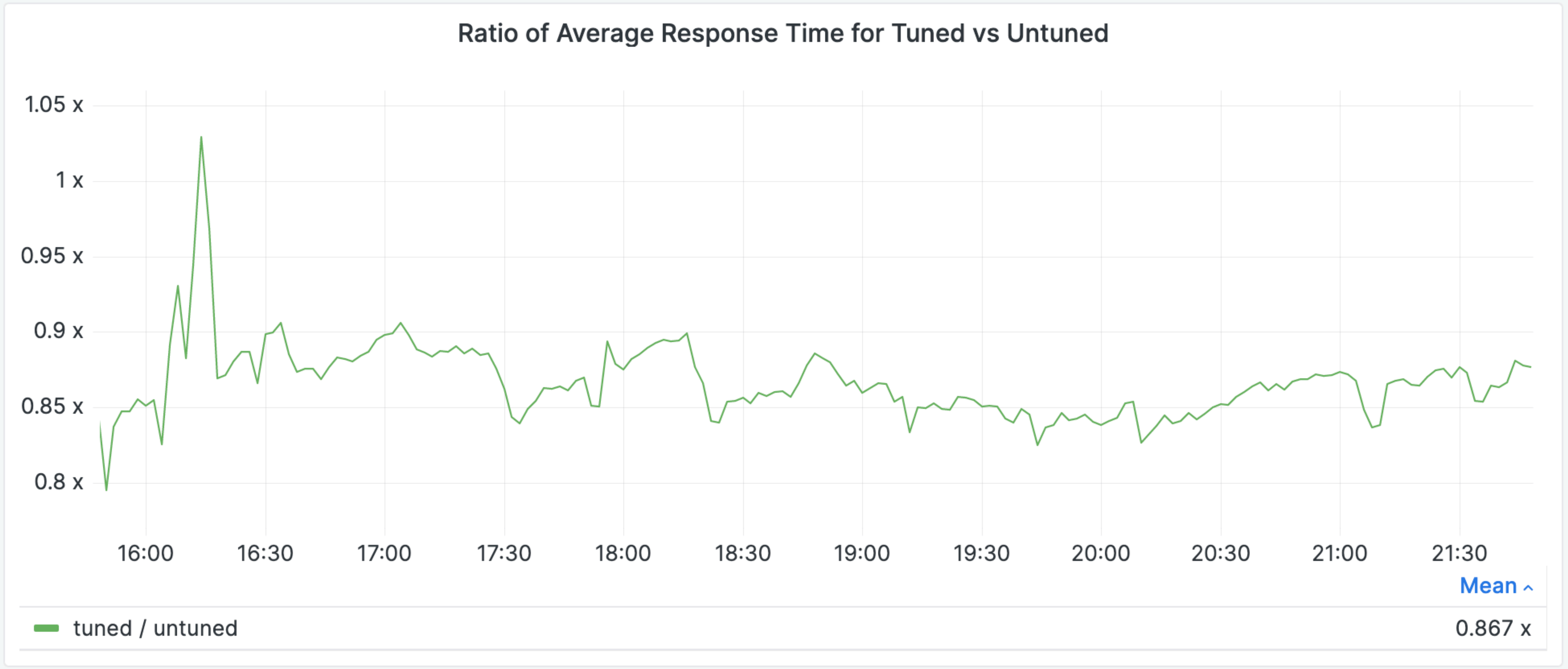
Out-of-band garbage collector

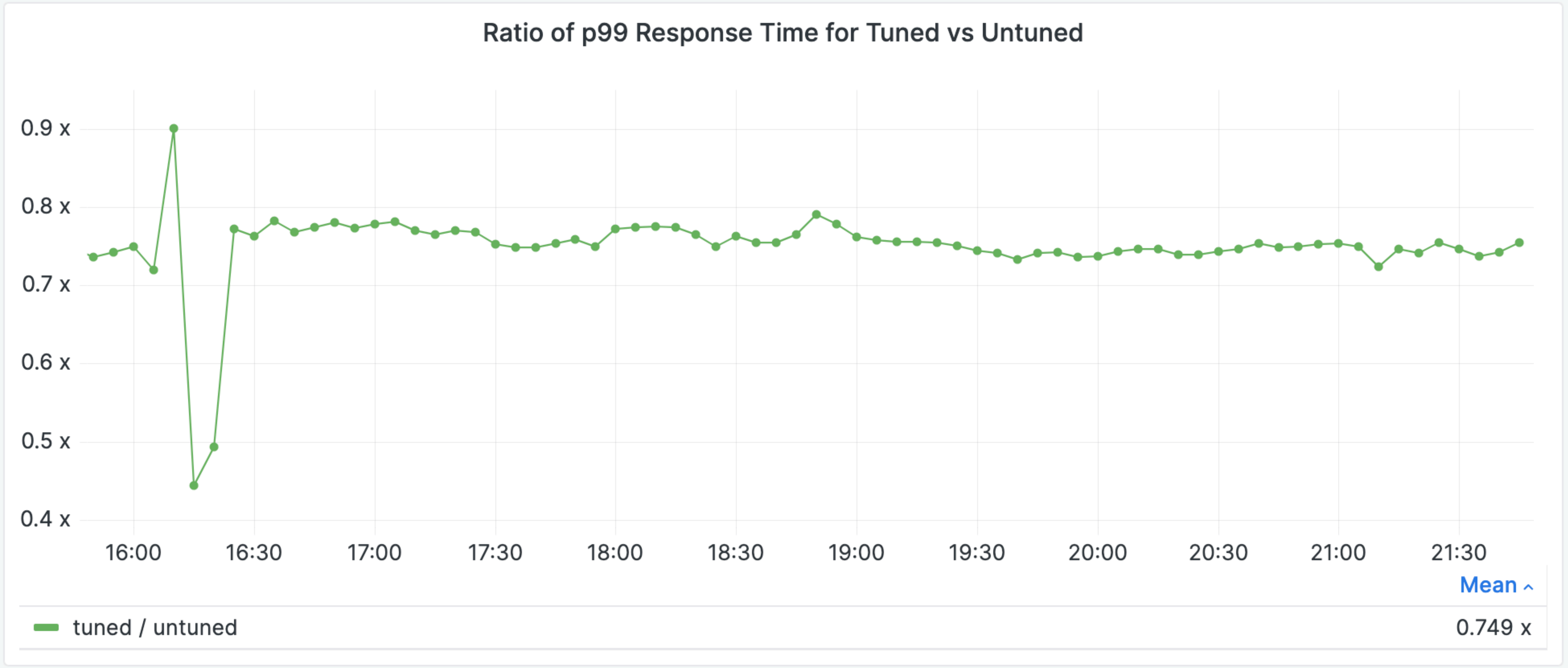
- Runs GC between requests
- Difficult to run on threaded web servers (e.g. Puma with multiple threads)
- Tricky to implement optimally
- Could decrease capacity if ran too frequently
- Ineffective if not ran often enough

Impacts of GC tuning

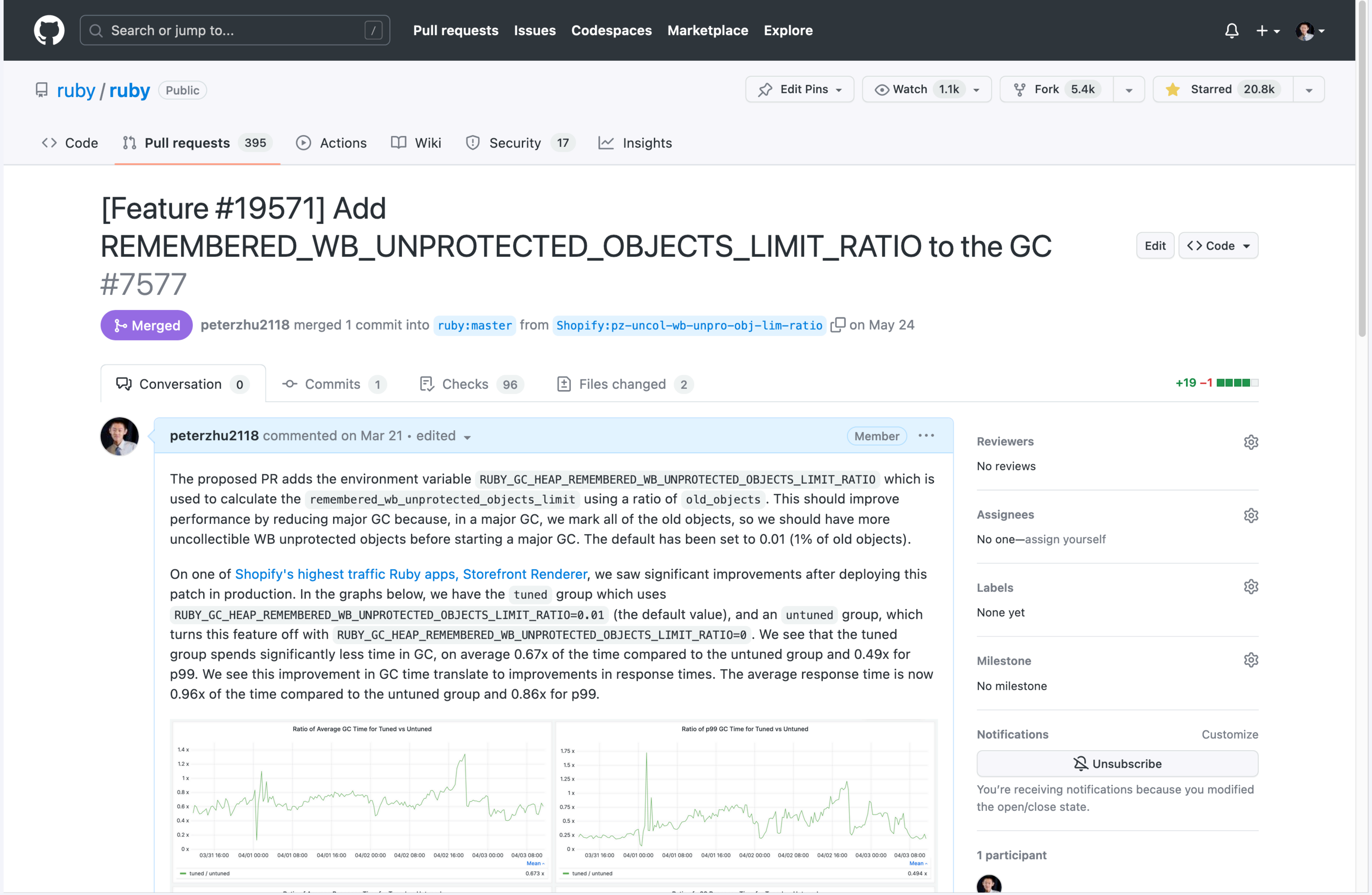

Ratio of Average GC Time for Tuned vs Untuned
0.6 x
0.55 x
0.5 x
0.45 x
0.4 x
0.35 x
16:00
16:30
17:00
17:30
18:00
18:30
19:00
19:30
20:00
20:30
21:00
21:30
Mean
tuned / untuned
0.454 x


Ratio of p99 GC Time for Tuned vs Untuned
0.325 x
0.3 x
0.275 x
0.25 x
0.225 x
0.2 x
0.175 x
0.15 x
16:00
16:30
17:00
17:30
18:00
18:30
19:00
19:30
20:00
20:30
21:00
21:30
Mean
tuned / untuned
0.201 x


Ratio of Average Response Time for Tuned vs Untuned
1.05 x
1 x
0.95 x
0.9 x
0.85 x
0.8 x
16:00
16:30
17:00
17:30
18:00
18:30
19:00
19:30
20:00
20:30
21:00
21:30
Mean
tuned / untuned
0.867 x


Ratio of p99 Response Time for Tuned vs Untuned
0.9 x
0.8 x
0.7 x
0.6 x
0.5 x
0.4 x
16:00
16:30
17:00
17:30
18:00
18:30
19:00
19:30
20:00
20:30
21:00
21:30
Mean
tuned / untuned
0.749 x

GC improvements in Ruby 3.3



[Feature #19571] Add REMEMBERED_WB_UNPROTECTED_OBJECTS_LIMIT_RATIO to the GC #7577

Merged peterzhu2118 merged 1 commit into `ruby:master` from `Shopify:pz-uncol-wb-unpro-obj-lim-ratio` on May 24

peterzhu2118 commented on Mar 21 • edited — Member

The proposed PR adds the environment variable `RUBY_GC_HEAP_REMEMBERED_WB_UNPROTECTED_OBJECTS_LIMIT_RATIO` which is used to calculate the `remembered_wb_unprotected_objects_limit` using a ratio of `old_objects`. This should improve performance by reducing major GC because, in a major GC, we mark all of the old objects, so we should have more uncollectible WB unprotected objects before starting a major GC. The default has been set to 0.01 (1% of old objects).

On one of Shopify's highest traffic Ruby apps, Storefront Renderer, we saw significant improvements after deploying this patch in production. In the graphs below, we have the `tuned` group which uses `RUBY_GC_HEAP_REMEMBERED_WB_UNPROTECTED_OBJECTS_LIMIT_RATIO=0.01` (the default value), and an `untuned` group, which turns this feature off with `RUBY_GC_HEAP_REMEMBERED_WB_UNPROTECTED_OBJECTS_LIMIT_RATIO=0`. We see that the tuned group spends significantly less time in GC, on average 0.67x of the time compared to the untuned group and 0.49x for p99. We see this improvement in GC time translate to improvements in response times. The average response time is now 0.96x of the time compared to the untuned group and 0.86x for p99.

Ratio of Average GC Time for Tuned vs Untuned
tuned / untuned
Mean 0.673 x
Ratio of p99 GC Time for Tuned vs Untuned
tuned / untuned
Mean 0.494 x

Reviewers
No reviews

Assignees
No one—assign yourself

Labels
None yet

Milestone
No milestone

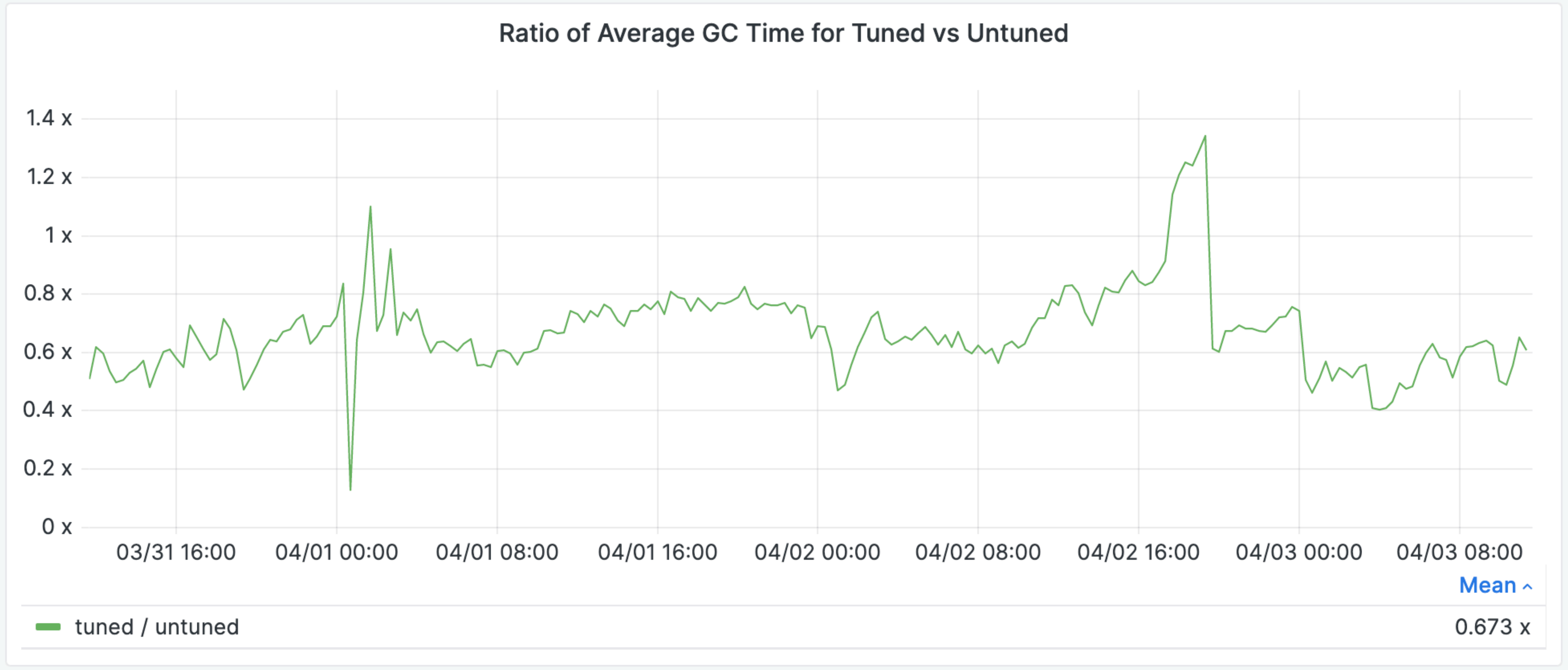
Notifications — Customize

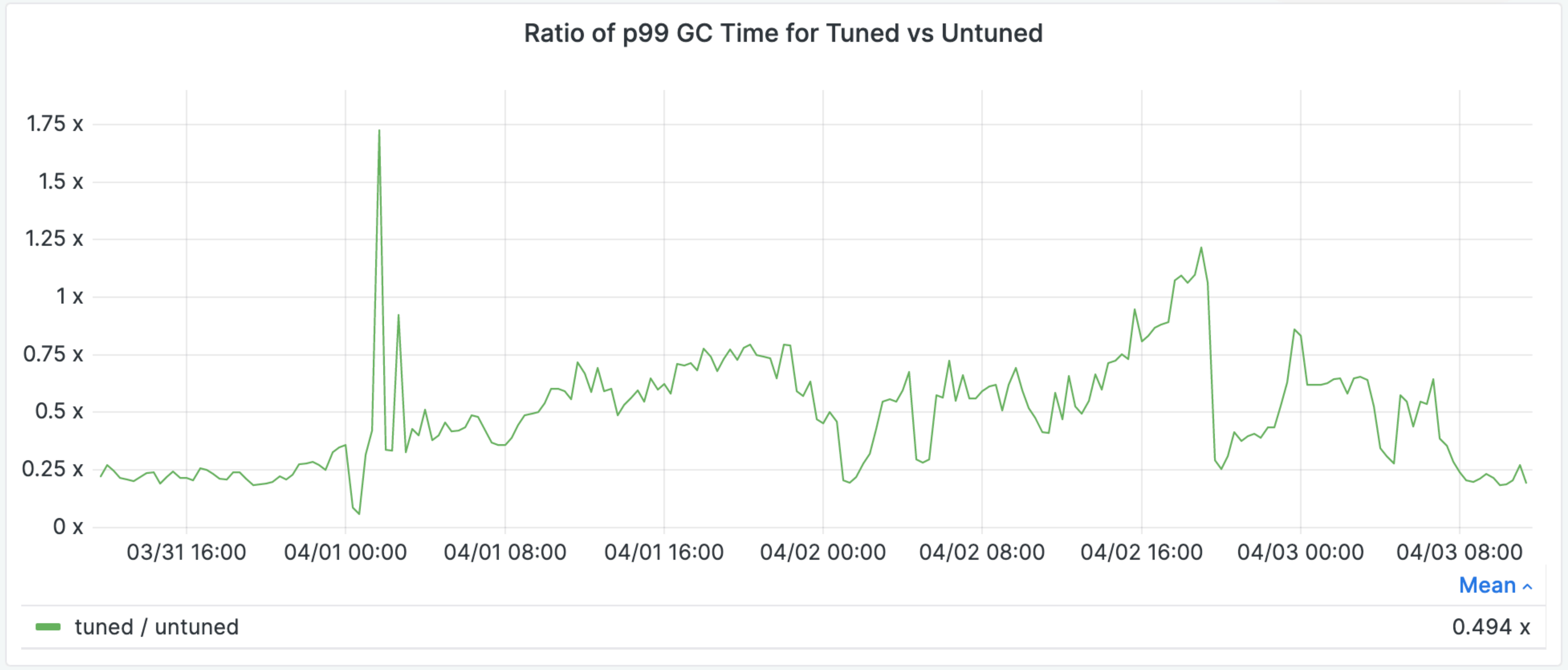
Unsubscribe

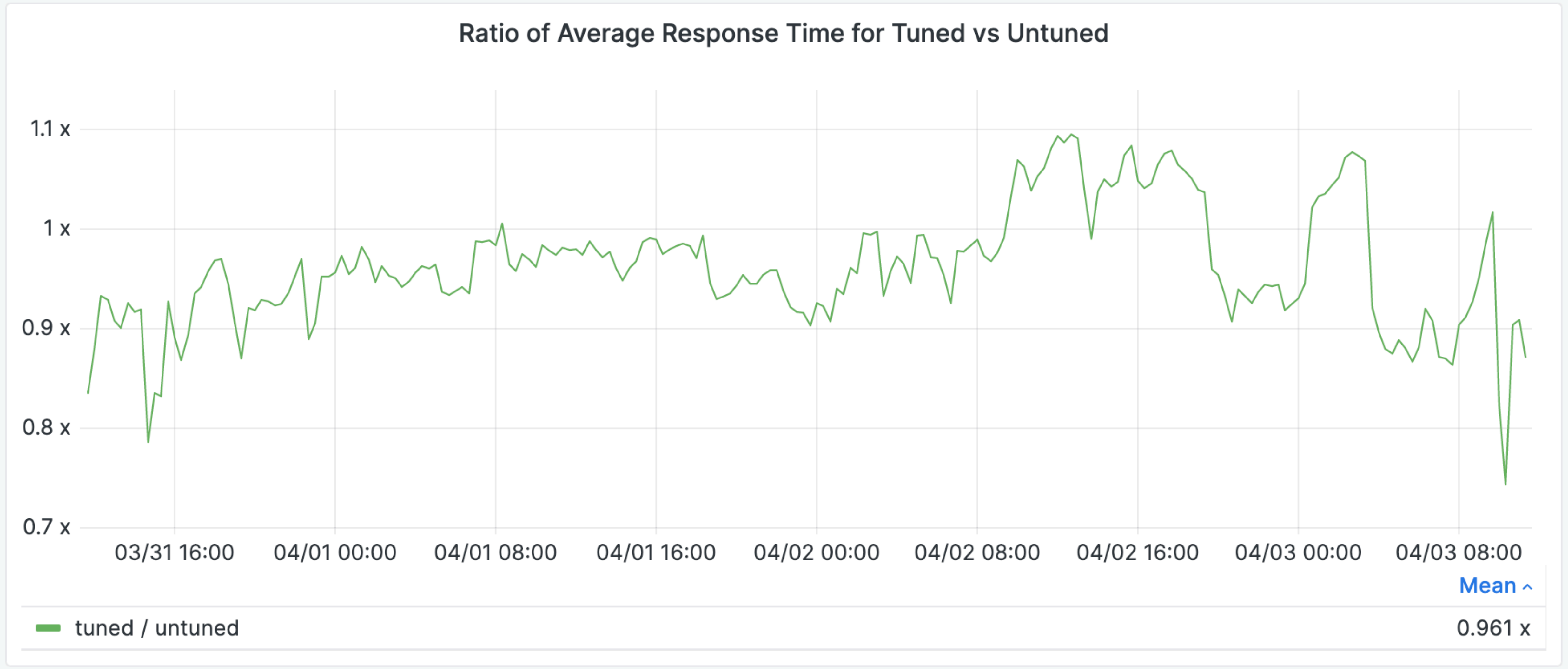
You're receiving notifications because you modified the open/close state.

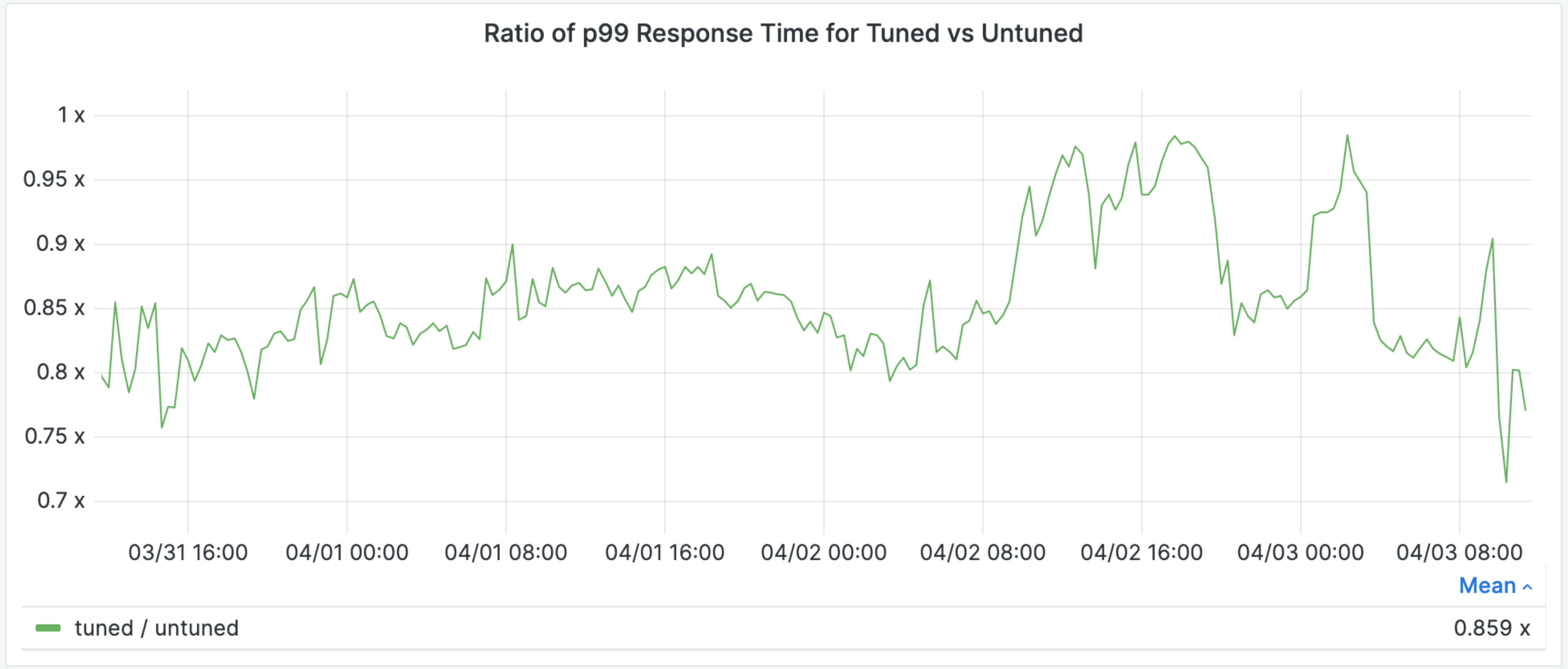
1 participant

https://github.com/ruby/ruby/pull/7577

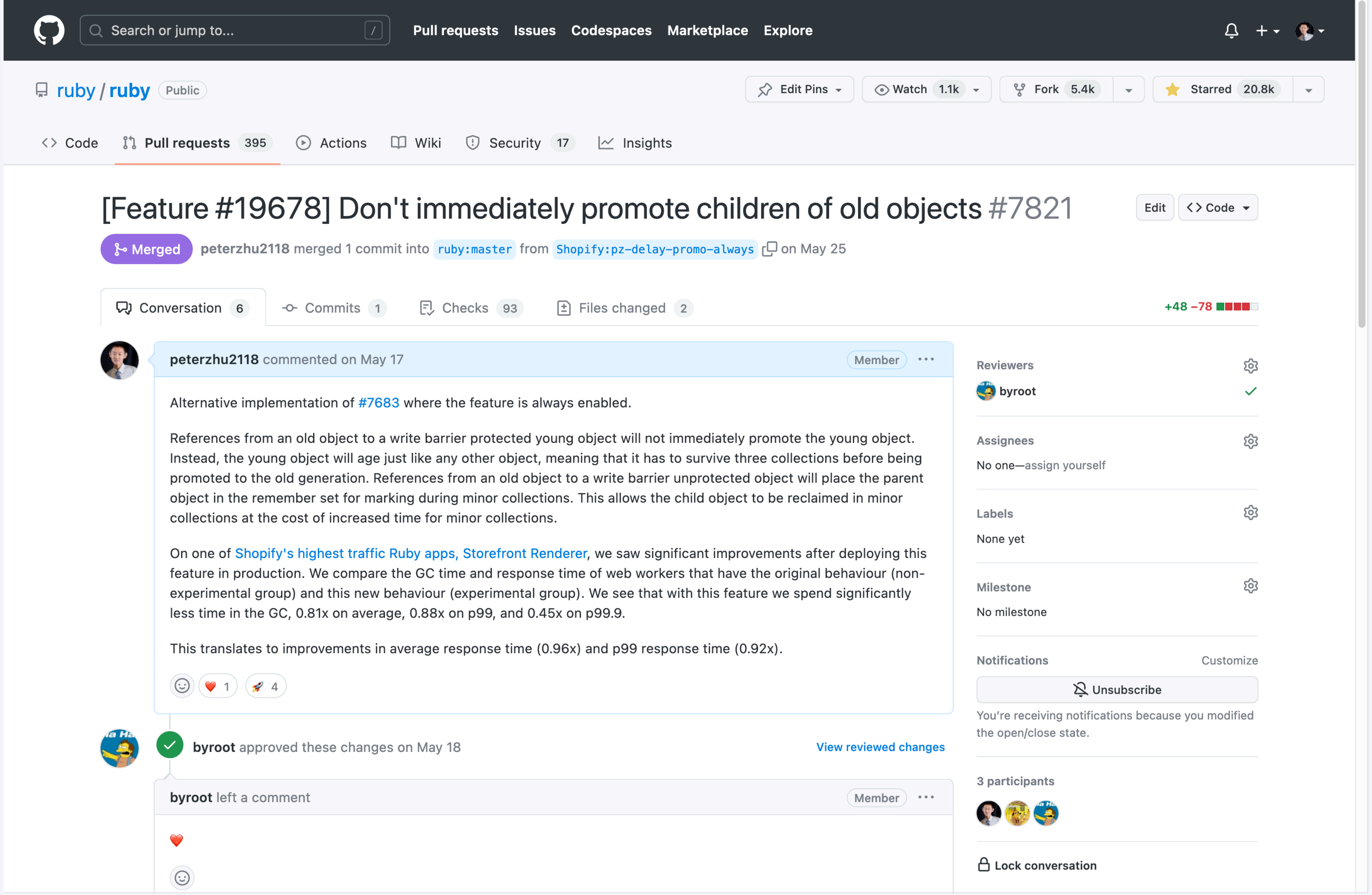








https://github.com/ruby/ruby/pull/7577



[Feature #19678] Don't immediately promote children of old objects #7821

Merged peterzhu2118 merged 1 commit into `ruby:master` from `Shopify:pz-delay-promo-always` on May 25

peterzhu2118 commented on May 17 — Member

Alternative implementation of #7683 where the feature is always enabled.

References from an old object to a write barrier protected young object will not immediately promote the young object. Instead, the young object will age just like any other object, meaning that it has to survive three collections before being promoted to the old generation. References from an old object to a write barrier unprotected object will place the parent object in the remember set for marking during minor collections. This allows the child object to be reclaimed in minor collections at the cost of increased time for minor collections.

On one of Shopify's highest traffic Ruby apps, Storefront Renderer, we saw significant improvements after deploying this feature in production. We compare the GC time and response time of web workers that have the original behaviour (non-experimental group) and this new behaviour (experimental group). We see that with this feature we spend significantly less time in the GC, 0.81x on average, 0.88x on p99, and 0.45x on p99.9.

This translates to improvements in average response time (0.96x) and p99 response time (0.92x).

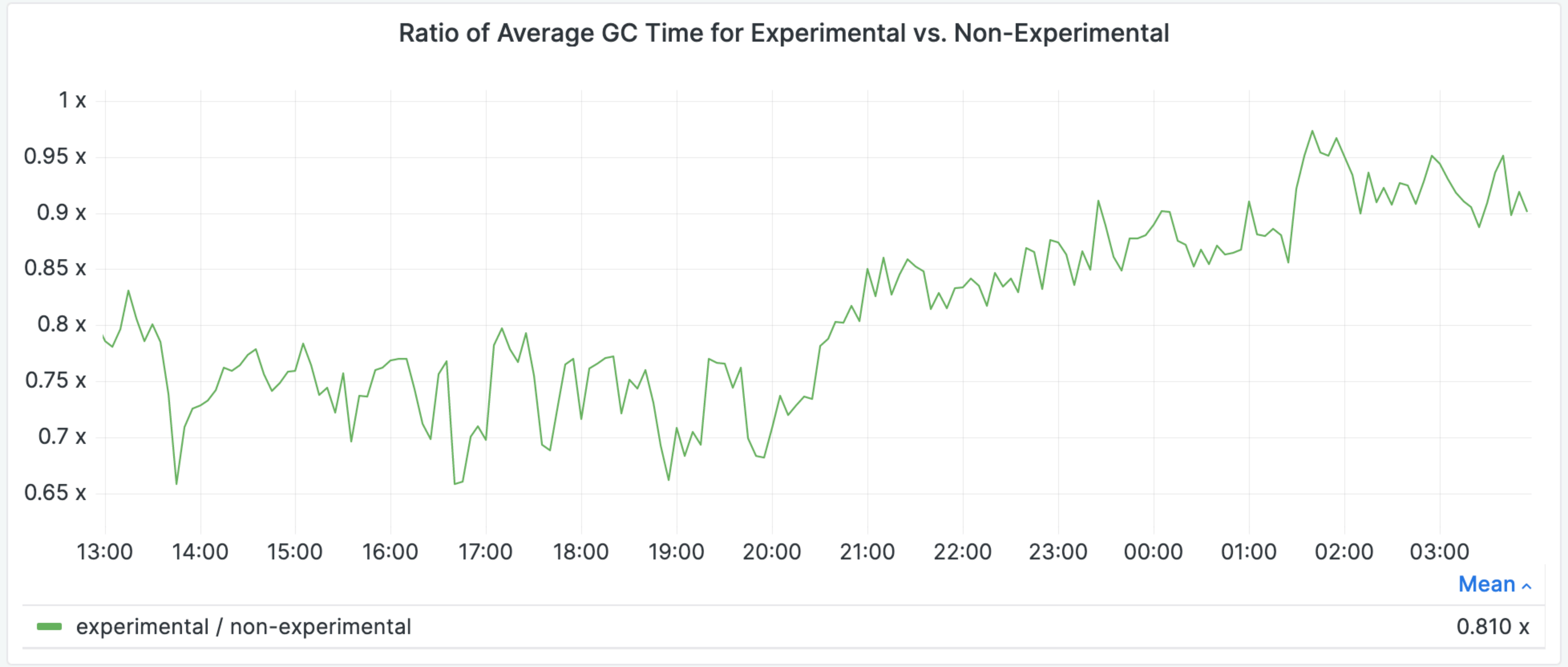
❤️ 1 🚀 4

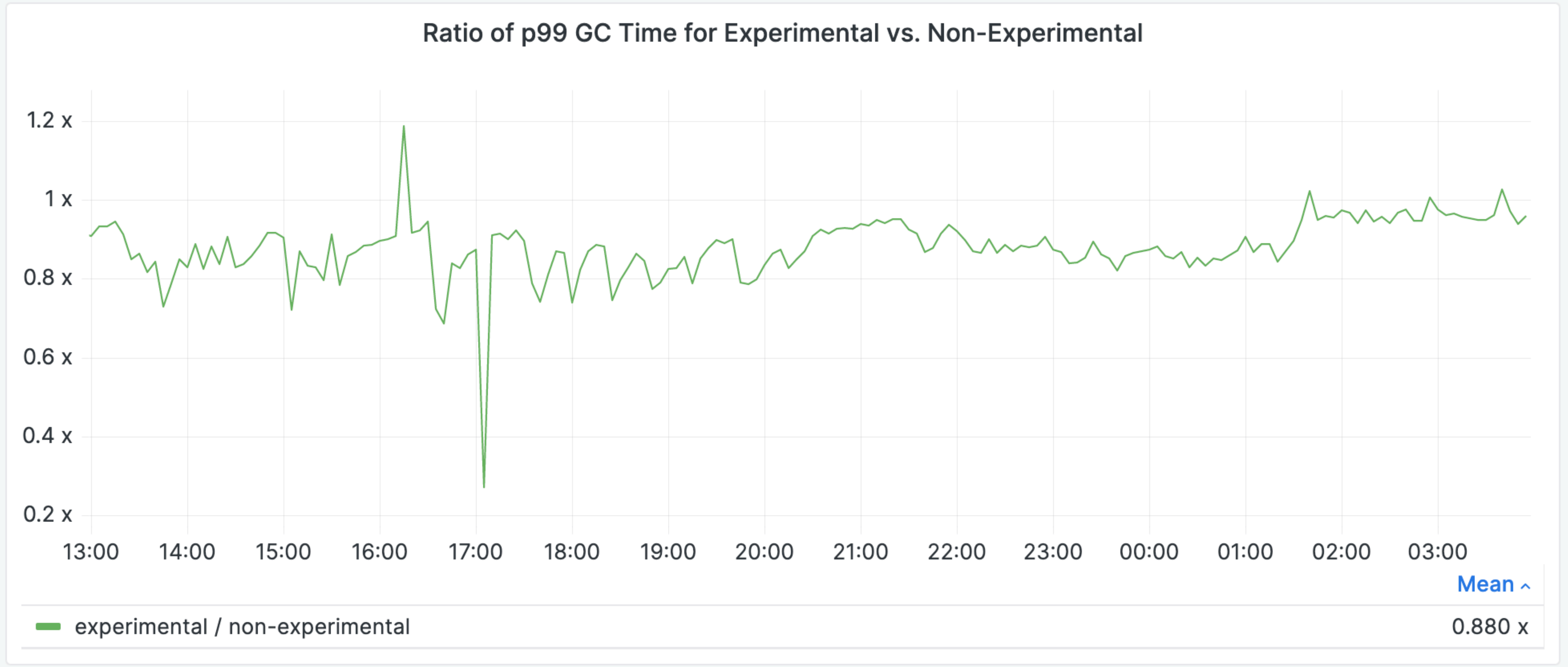
byroot approved these changes on May 18 — View reviewed changes

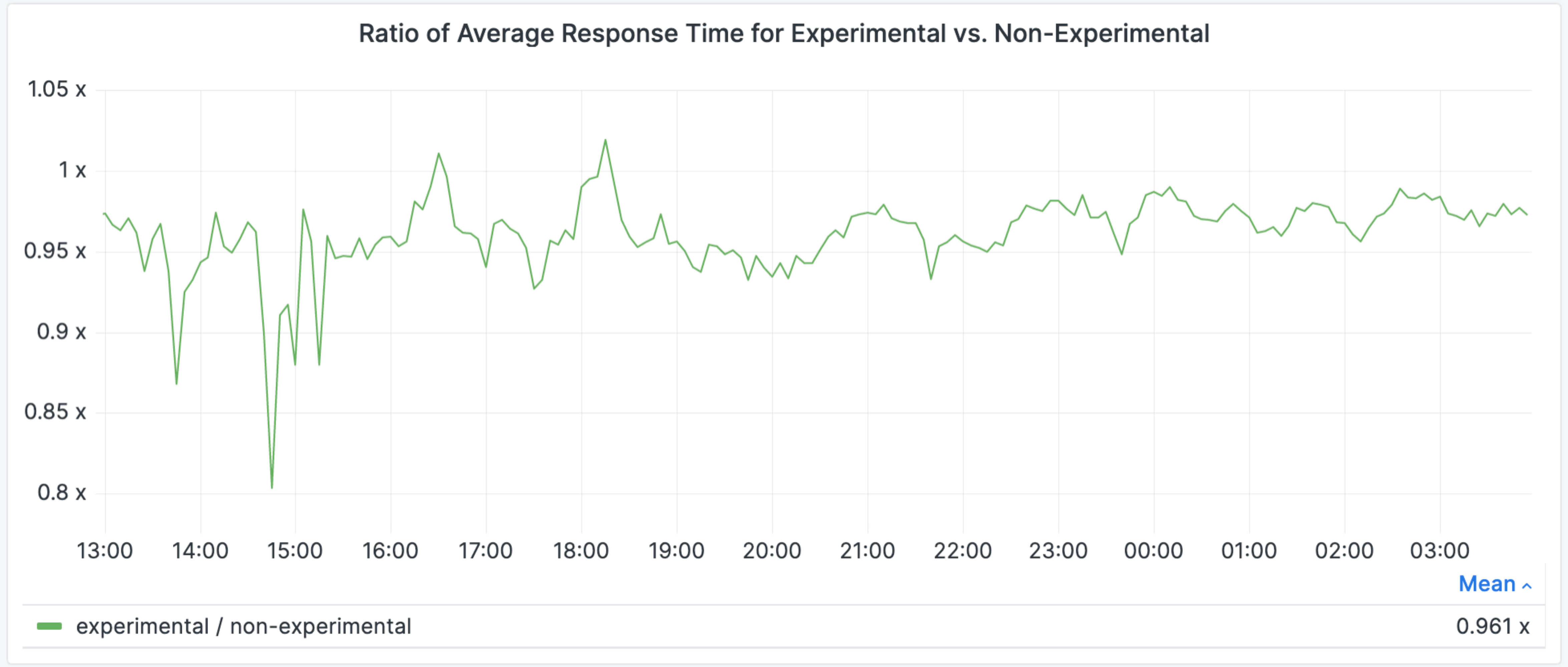
byroot left a comment — Member

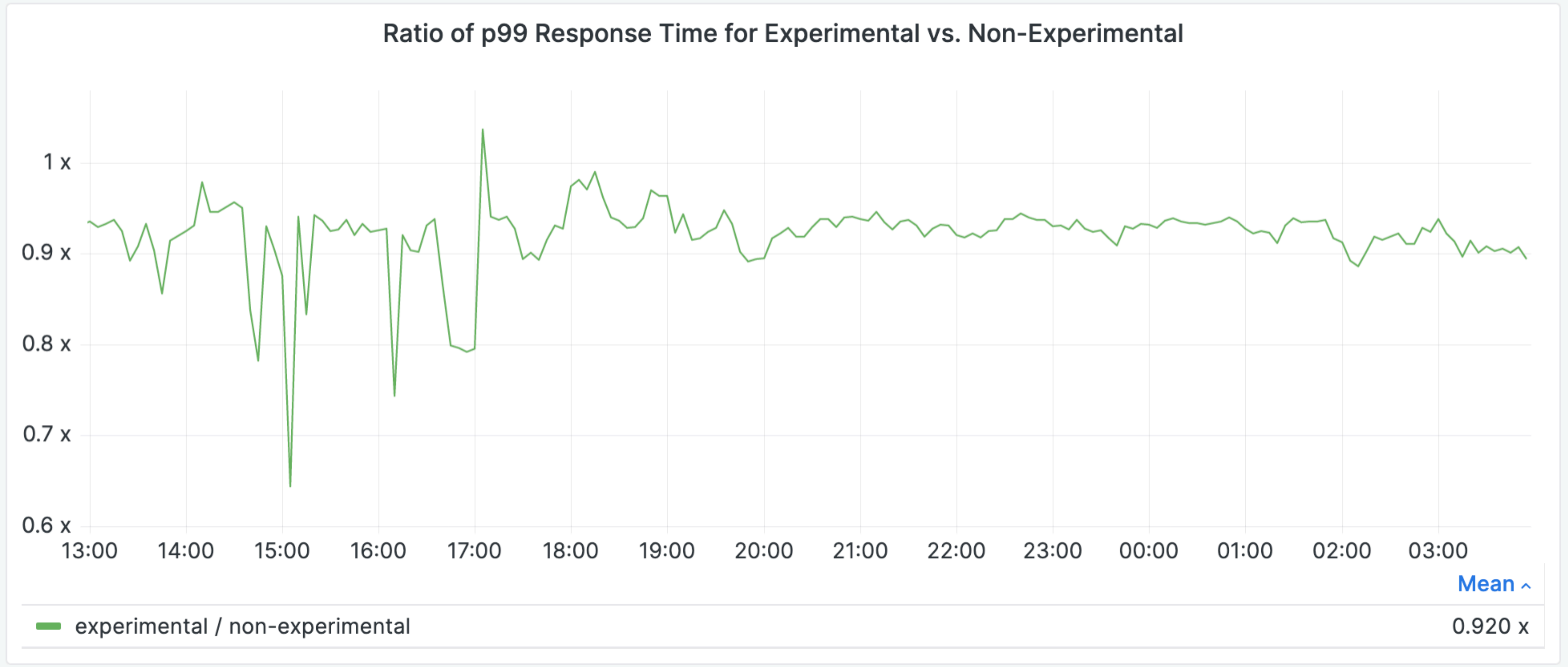
❤️

https://github.com/ruby/ruby/pull/7821







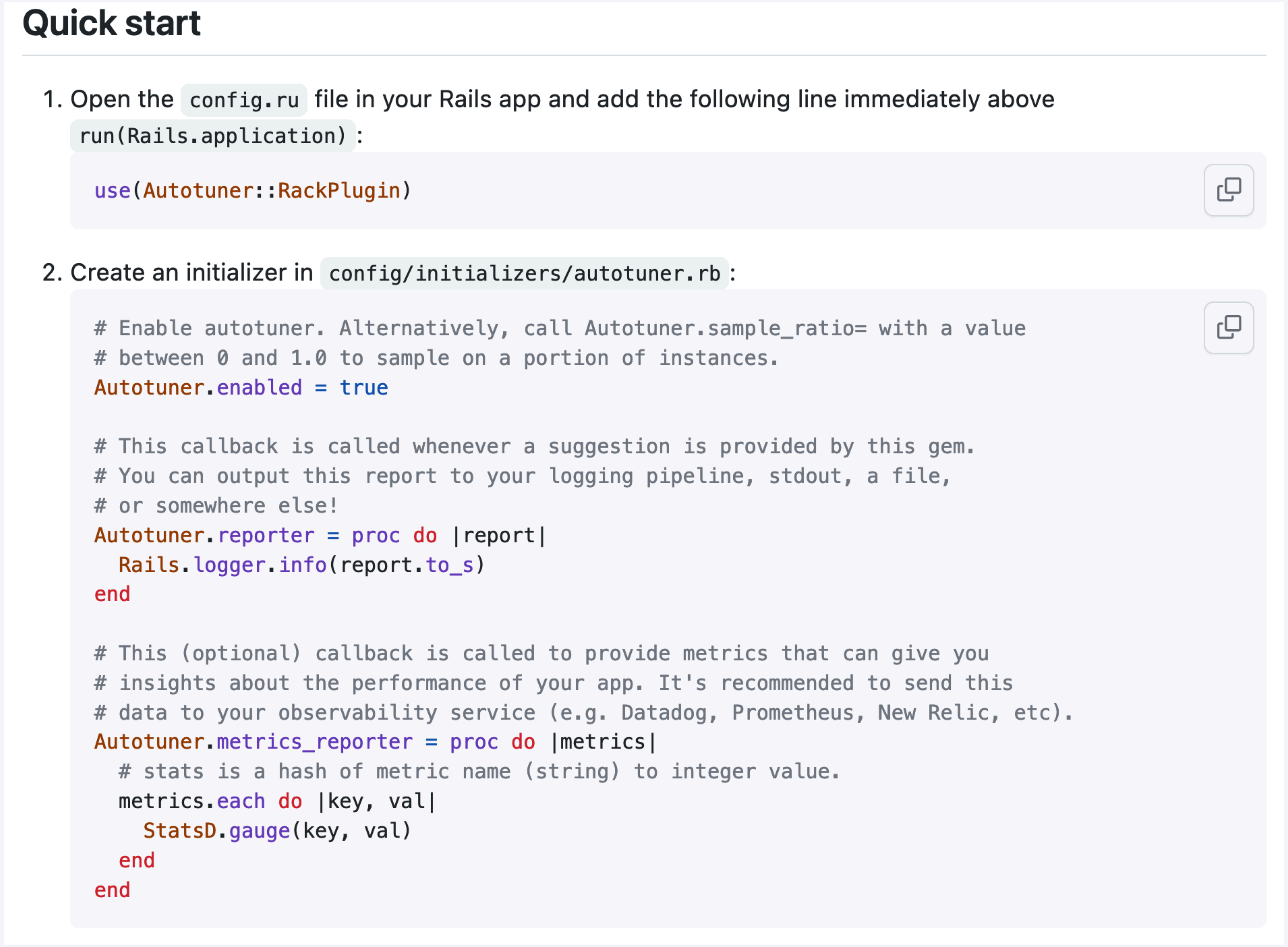


https://github.com/ruby/ruby/pull/7821

railsatscale.com/2023-08-08-two-garbage-collection-improvements-made-our-storefronts-8-faster/

The autotuner gem

github.com/Shopify/autotuner



Quick start

1. Open the `config.ru` file in your Rails app and add the following line immediately above `run(Rails.application)`:

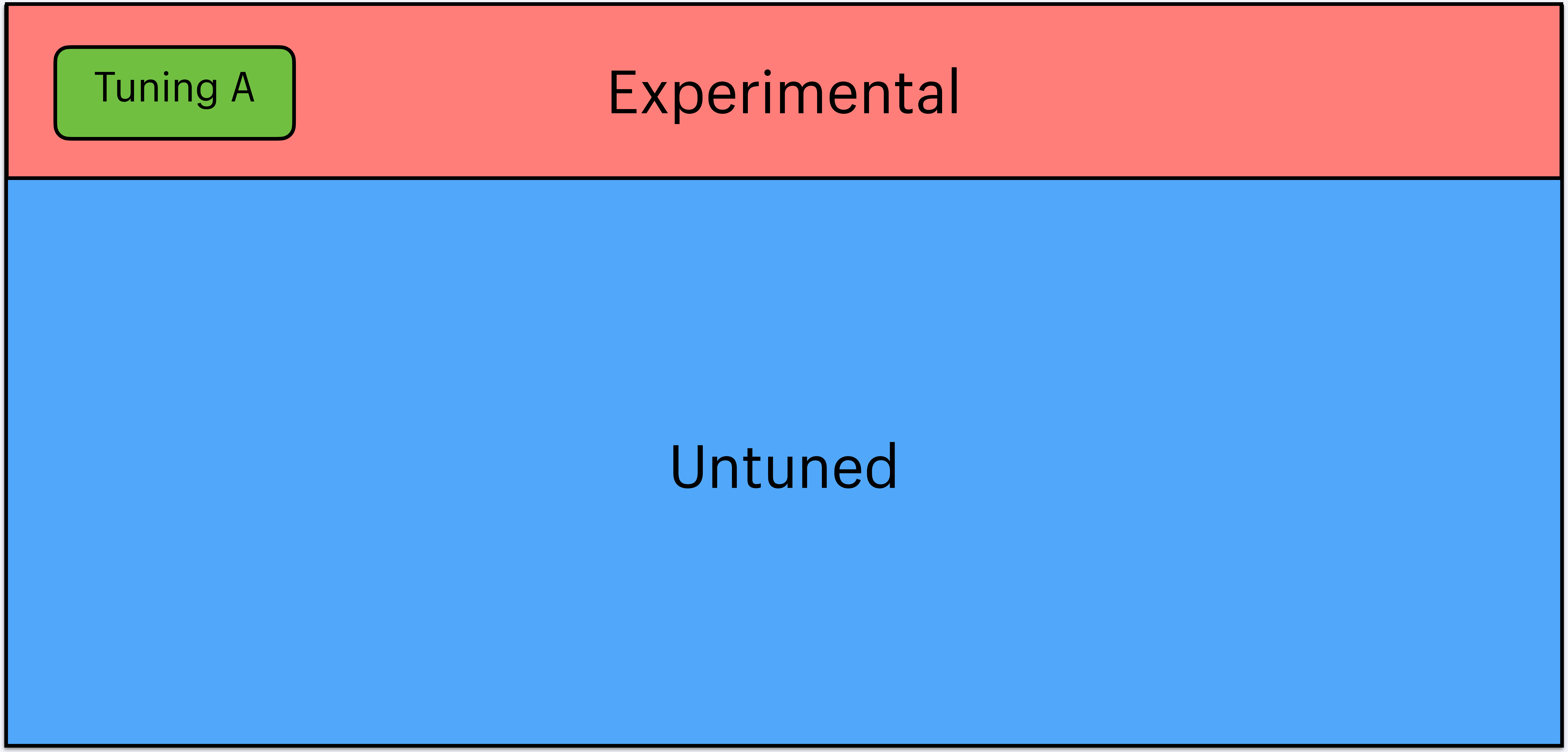
```ruby
use(Autotuner::RackPlugin)
```

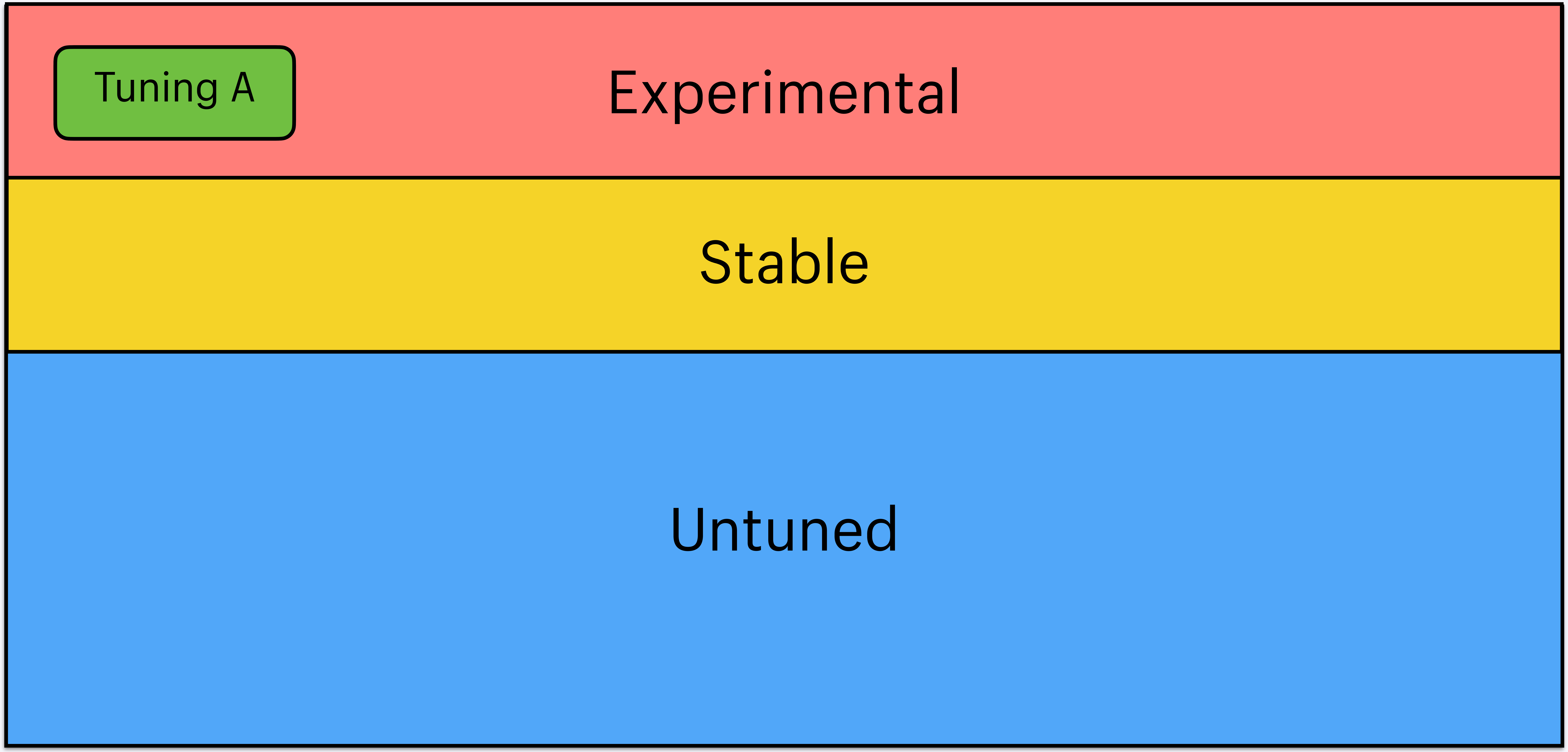
2. Create an initializer in `config/initializers/autotuner.rb`:

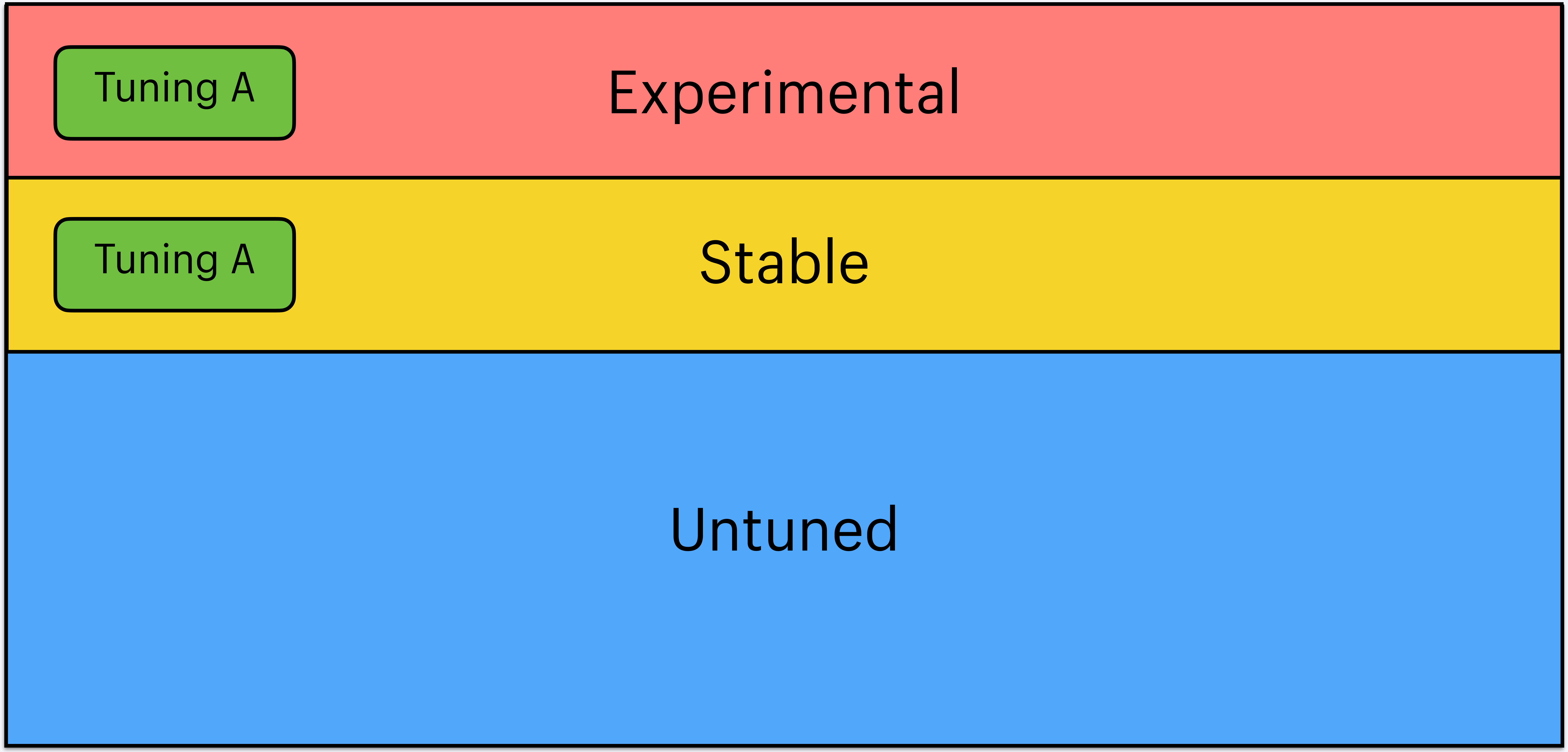
```ruby
# Enable autotuner. Alternatively, call Autotuner.sample_ratio= with a value
# between 0 and 1.0 to sample on a portion of instances.
Autotuner.enabled = true

# This callback is called whenever a suggestion is provided by this gem.
# You can output this report to your logging pipeline, stdout, a file,
# or somewhere else!
Autotuner.reporter = proc do |report|
  Rails.logger.info(report.to_s)
end

# This (optional) callback is called to provide metrics that can give you
# insights about the performance of your app. It's recommended to send this
# data to your observability service (e.g. Datadog, Prometheus, New Relic, etc).
Autotuner.metrics_reporter = proc do |metrics|
  # stats is a hash of metric name (string) to integer value.
  metrics.each do |key, val|
    StatsD.gauge(key, val)
  end
end
```

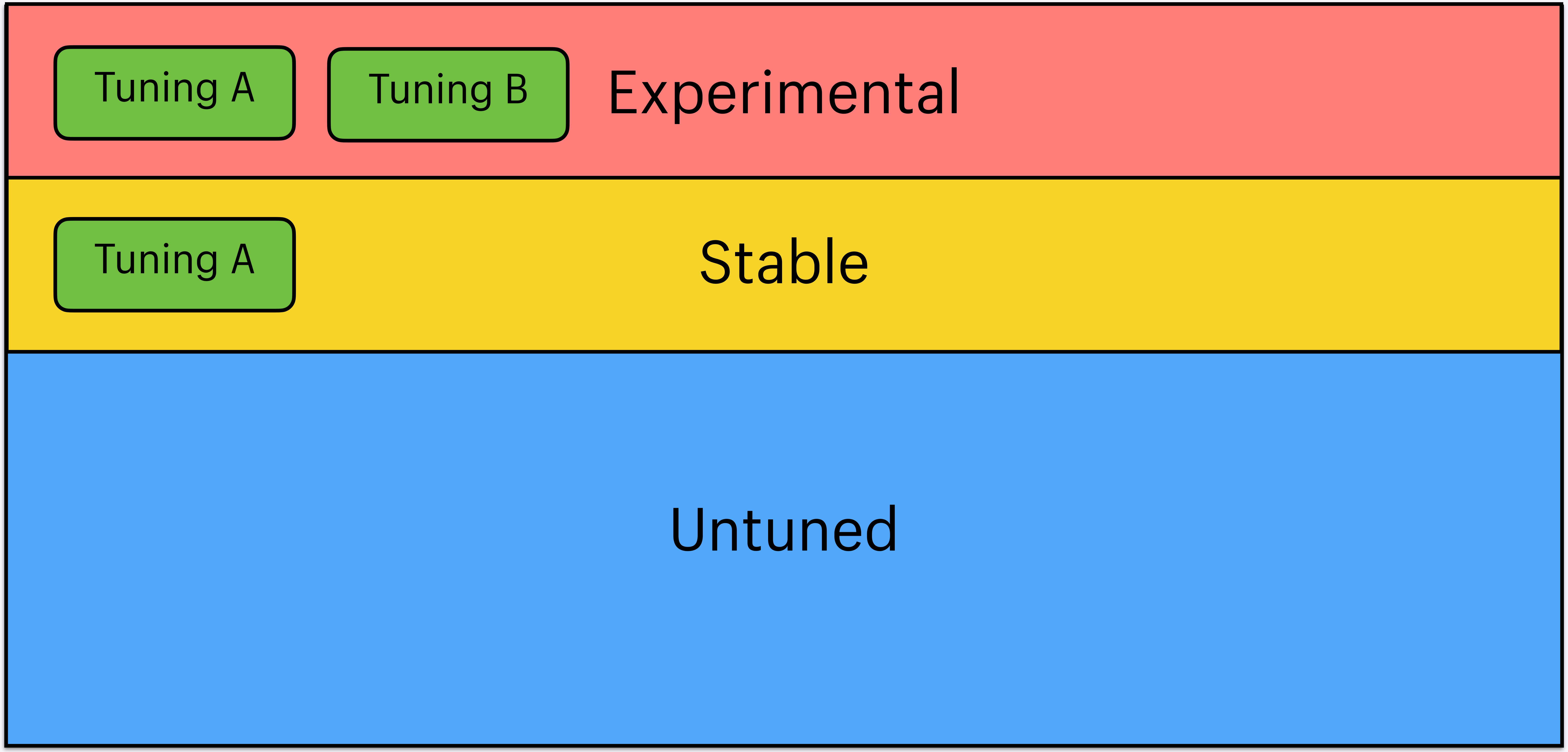
```
     autotuner: The following suggestions reduces the number of minor garbage collection cycles, specifically
a cycle called "malloc". Your app runs malloc cycles in approximately 62.50% of all minor garbage collection
cycles.

Reducing minor garbage collection cycles can help reduce response times. The following tuning values aims to
reduce malloc garbage collection cycles by setting it to a higher value. This may cause a slight increase in
memory usage. You should monitor memory usage carefully to ensure your app is not running out of memory.

Suggested tuning values:
  RUBY_GC_MALLOC_LIMIT=67108864 (configured value: 33554432)
  RUBY_GC_MALLOC_LIMIT_MAX=134217728 (configured value: 67108864)

It is always recommended to experiment with these suggestions as some suggestions may not always yield
positive performance improvements. The recommended method is to perform A/B testing where a portion of
traffic does not have the these suggested values and a portion of traffic with these suggested values.
```

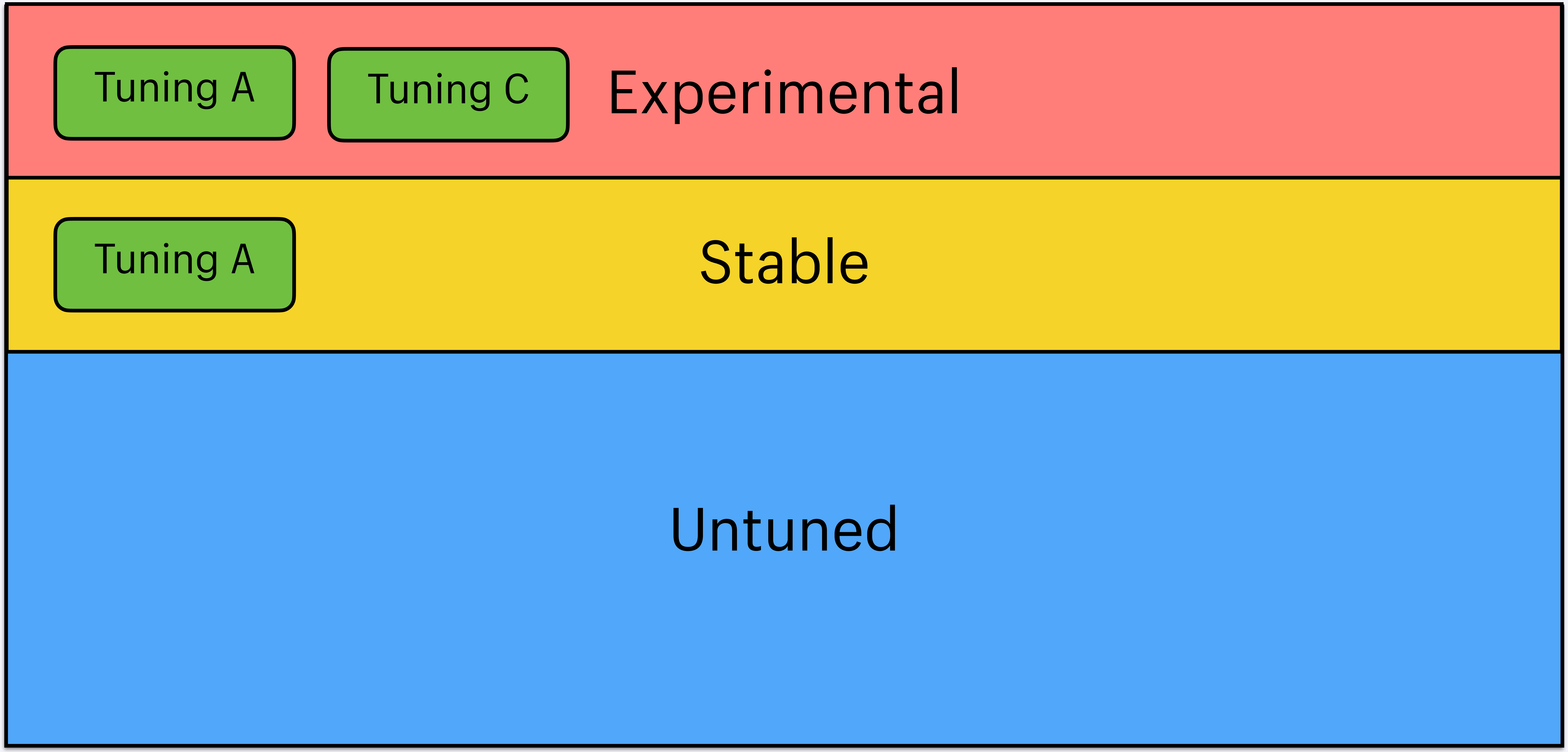
Untuned

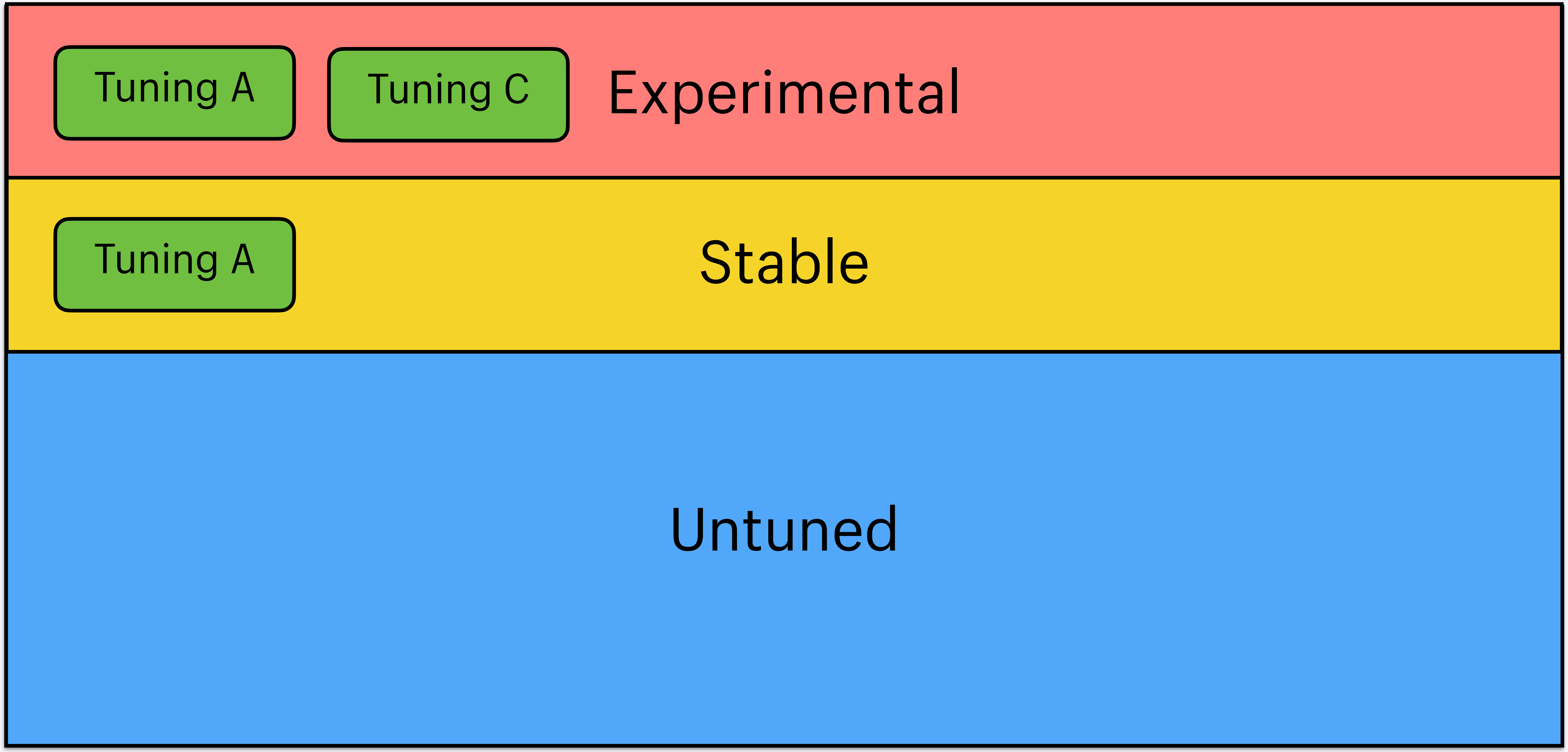

Tuning A
Experimental
Untuned

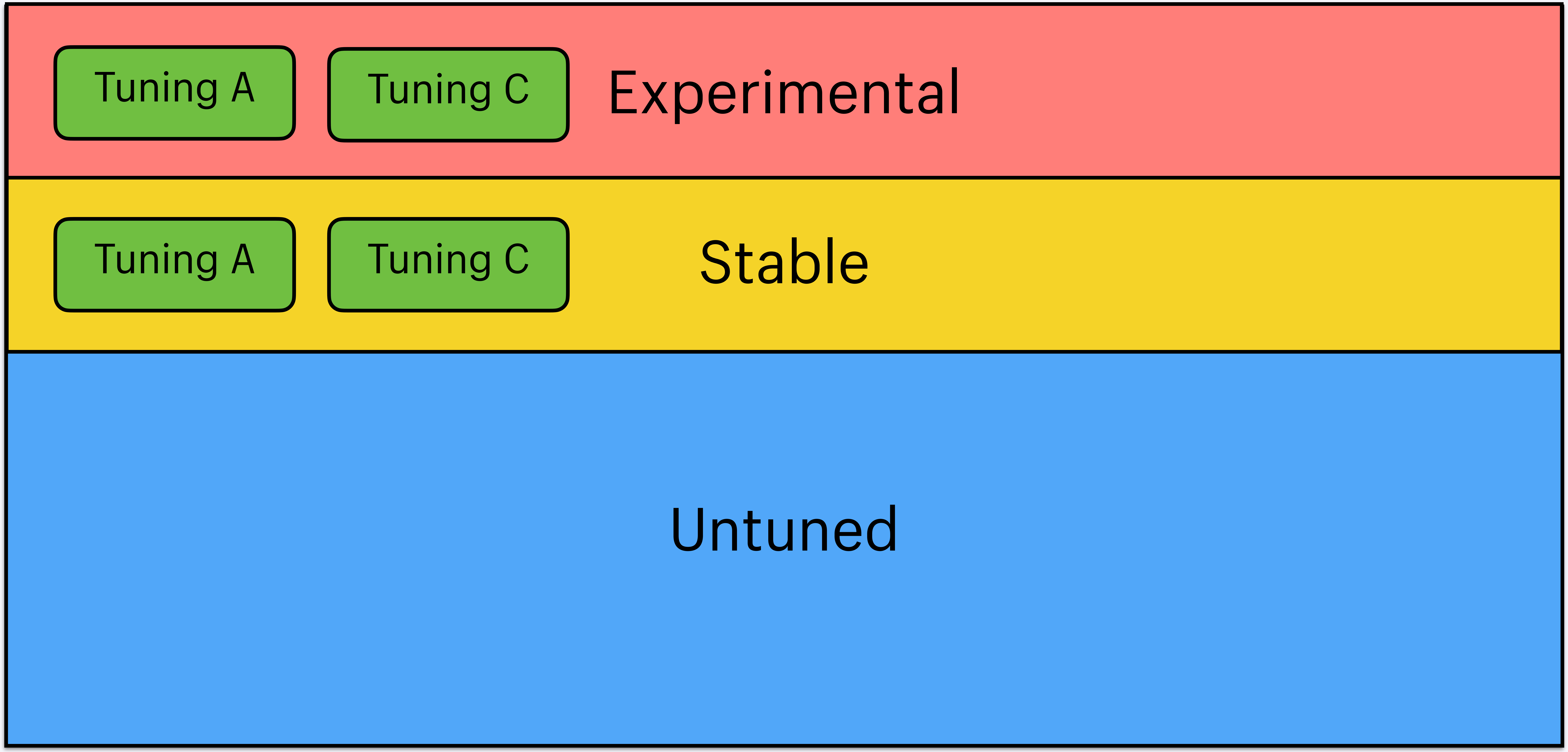

Tuning A
Experimental
Stable
Untuned

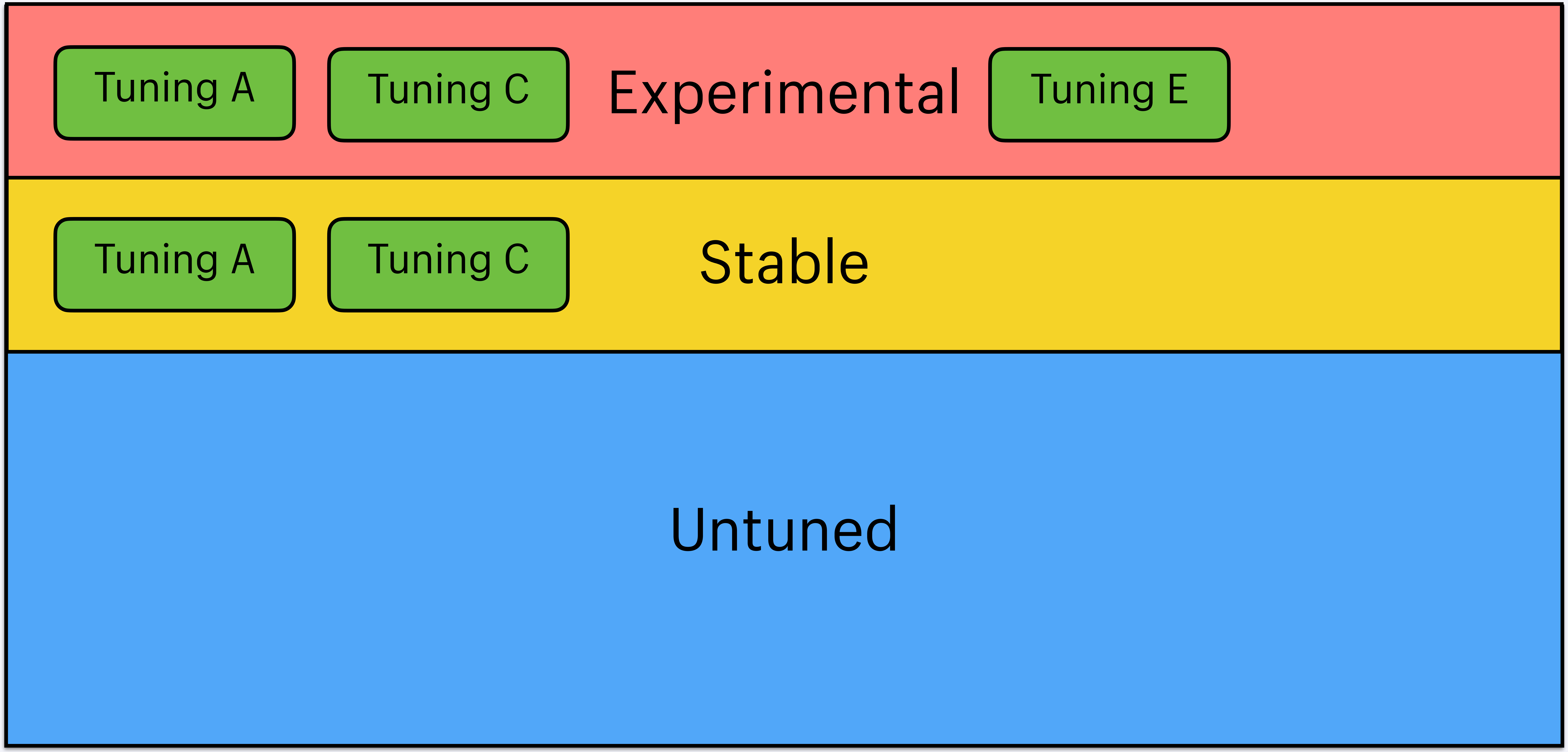

Tuning A
Experimental
Tuning A
Stable
Untuned

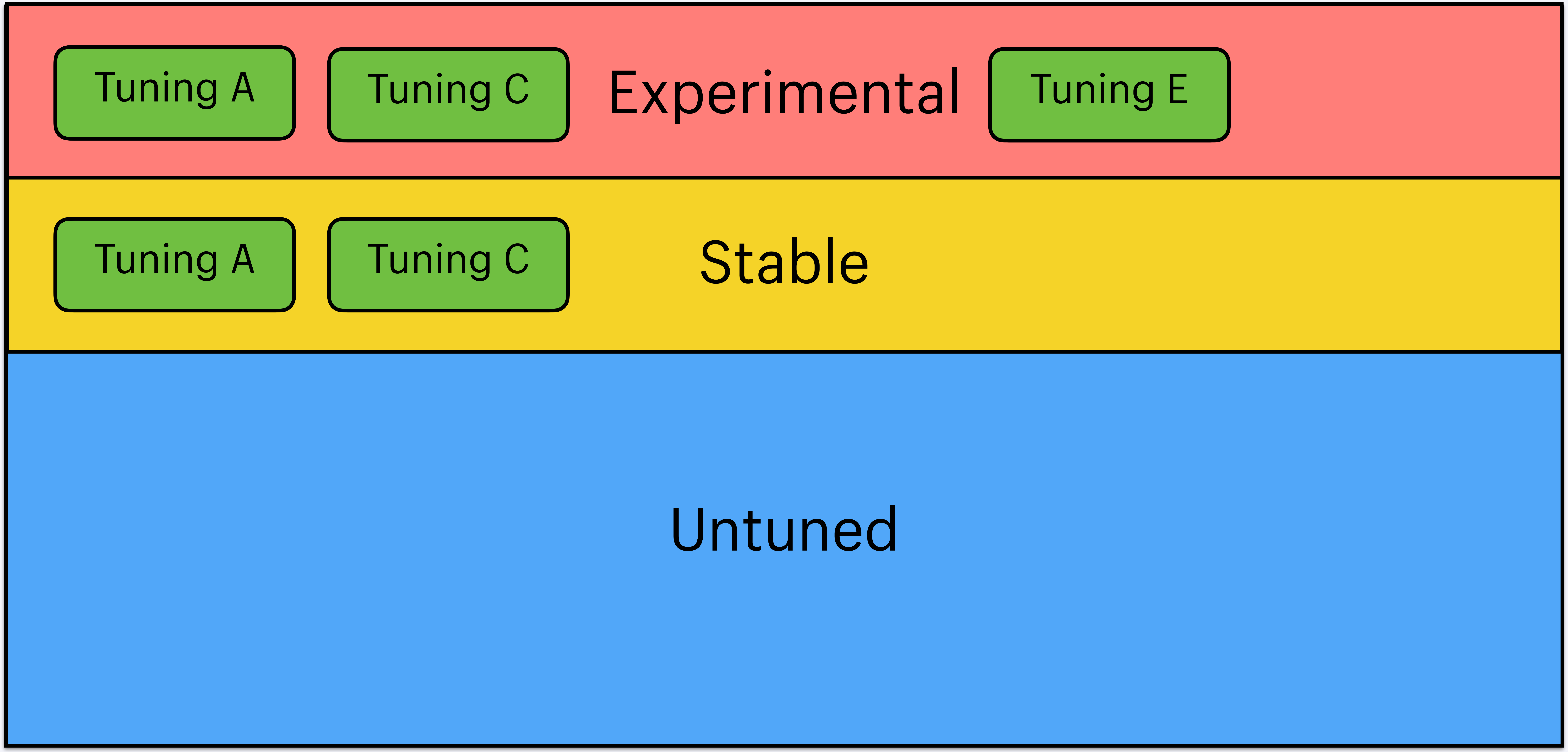

Tuning A
Tuning B
Experimental
Tuning A
Stable
Untuned

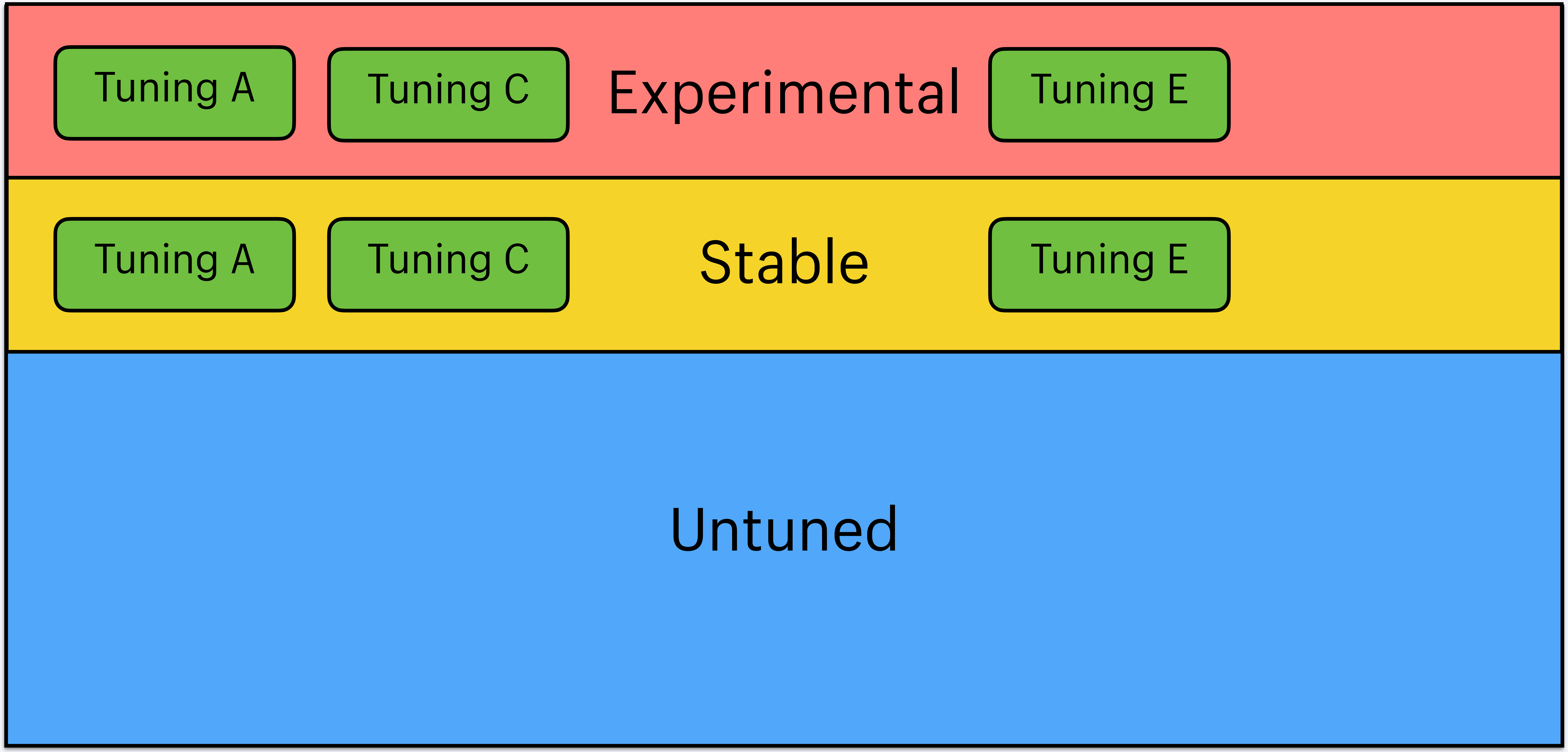

Tuning A
Tuning C
Experimental
Tuning A
Stable
Untuned

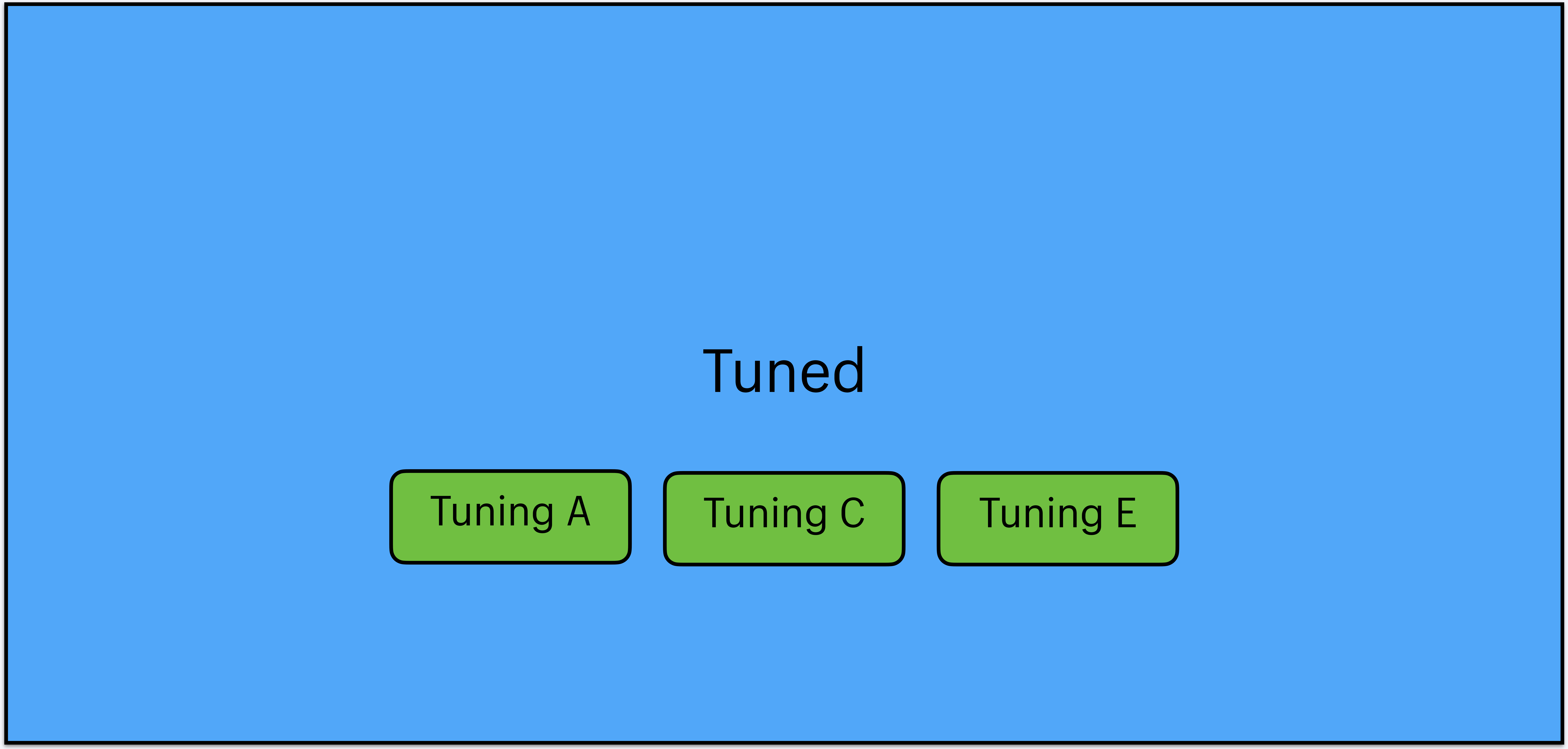

Tuning A
Tuning C
Experimental
Tuning A
Stable
Untuned


Tuning A
Tuning C
Experimental
Tuning A
Tuning C
Stable
Untuned


Tuning A
Tuning C
Experimental
Tuning E
Tuning A
Tuning C
Stable
Untuned


Tuning A
Tuning C
Experimental
Tuning E
Tuning A
Tuning C
Stable
Untuned


Tuning A
Tuning C
Experimental
Tuning E
Tuning A
Tuning C
Stable
Tuning E
Untuned


Tuned
Tuning A
Tuning C
Tuning E

Thank You

github.com/Shopify/autotuner

@peterzhu2118

peter@peterzhu.ca