Understanding Garbage Collectors

in object-oriented programming

Lecture 4 - Tuning the Garbage Collector

Nahuel Palumbo

nahuel.palumbo@inria.fr

CERCIRAS 2023

Automatic Memory Management

Garbage Collectors

Compute the size
Allocate in the memory

```
data = malloc(size)
...
... use data ...
...
free(data)
```

Manually
Memory Management

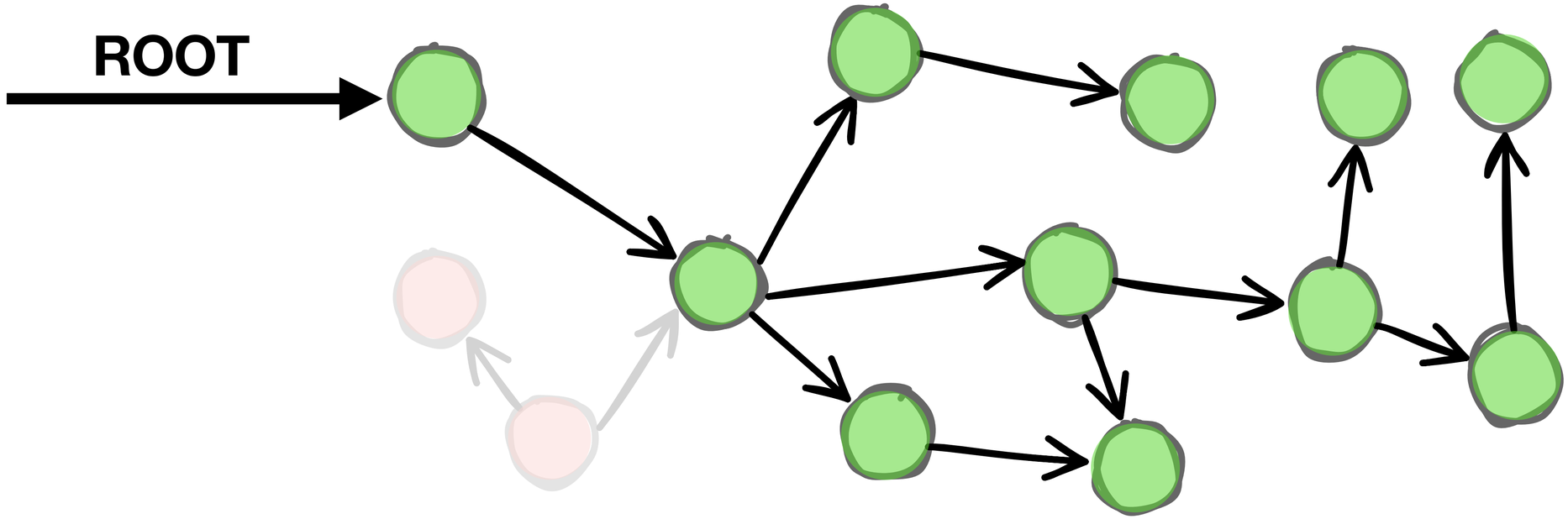
```
data = Data new
...
... use data ...
...
??????????
```

Maybe move the data for better use of the memory

Free the space when data is not used anymore

Application's Allocation Patterns

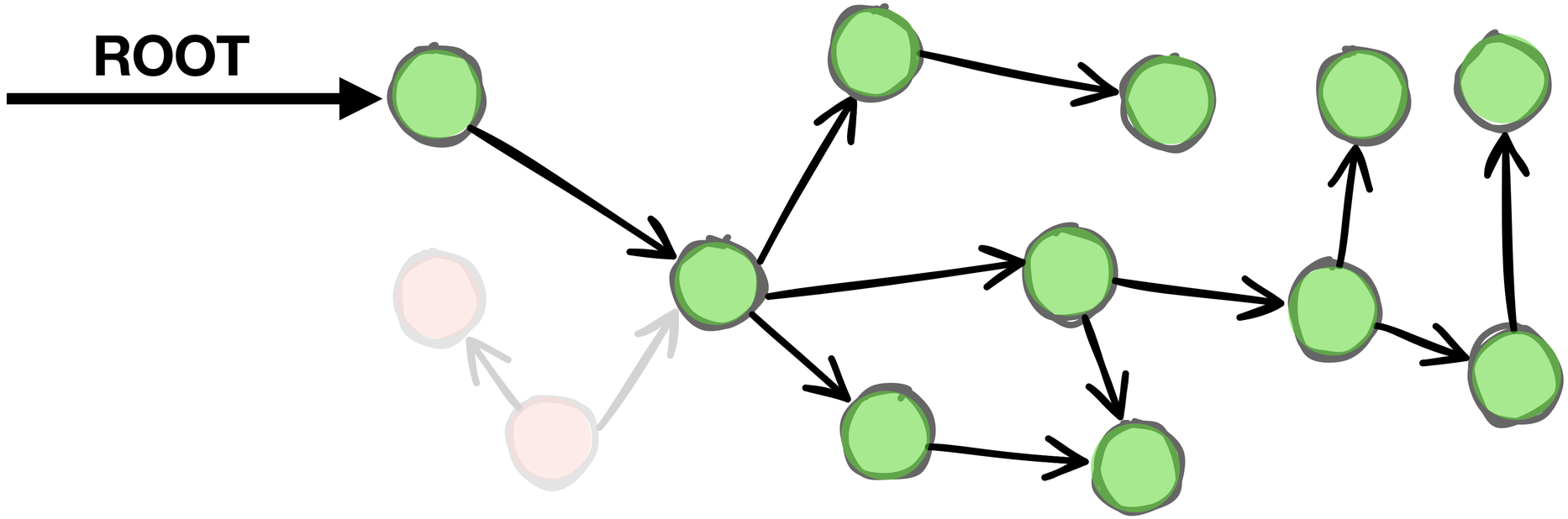
When an object dies?



"Must be accessible from the roots"

Application's Allocation Patterns

When an object dies?

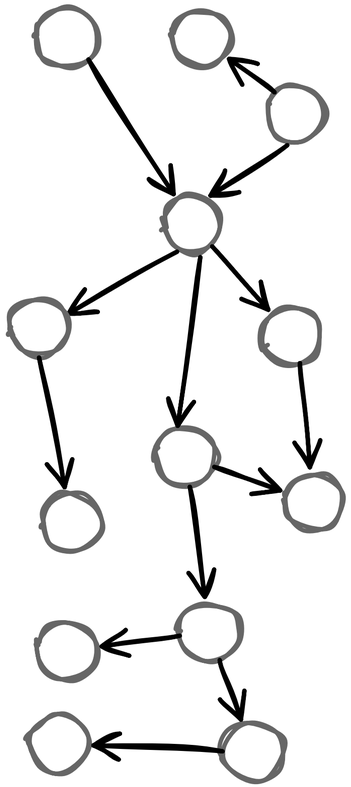


>1.5M objects

"Must be accessible from the roots"

Application's Allocation Patterns

How do the applications use the memory?

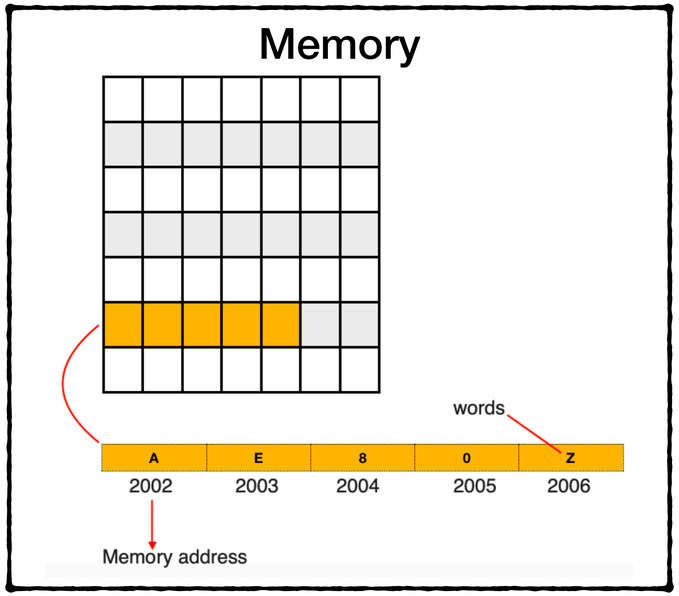


Allocations are particular for each application

→ **Hard to predict**

There are some general heuristics

→ **Easy to predict**



Application's Allocation Patterns

Weak generational hypothesis

```
data = Data new
... first use of data ...
??????????
```

Will you use it again?

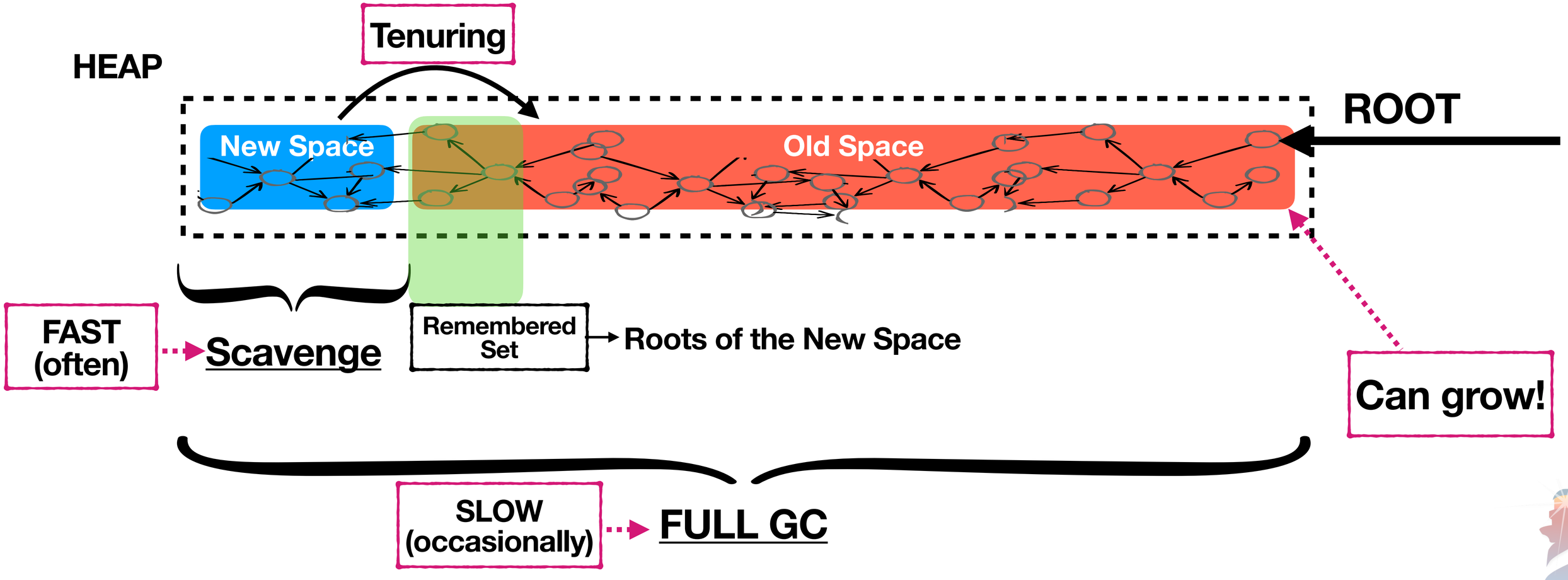
Mostly — No, you can free the memory

Sometimes — Yes, keep it!

"Most of the objects die young"

Generational Garbage Collector

High-Performance Automatic Memory Management for OOP



D. Ungar, *Generation scavenging: A non-disruptive high performance storage reclamation algorithm*

Pathological Allocation Pattern

Generational Garbage Collectors' problem

Weak generational

Stable memory use

Few Full GCs

Fast Scavengers

Long lifetimes

Memory-starved

Many Full GCs

Scavenger overhead

DataFrame

“Pharo is slow”

DataFrame

DataFrame

“~~Pharo~~ The GC is slow”

Data size	Total time (sec)	GC time (sec)	GC overhead
529 MB	43	7	16%
1.6 GB	150	38	25%
3.1 GB	5599	5158	**92%**

DataFrame

DataFrame

“~~Pharo~~ The GC is slow”

Data size	Total time (sec)	GC time (sec)	GC overhead
529 MB	43	7	16%
1.6 GB	150	38	25%
3.1 GB	5599 >1h30m	5158	92% !!!!

~1h25m

DataFrame

Pathological Allocation Pattern

Garbage Collectors' problem

Weak generational

Stable memory use

Few Full GCs

Fast Scavengers

Long lifetimes

Memory-starved

Many Full GCs

Scavenger overhead

Pathological Allocation Pattern

Garbage Collectors' problem

Weak generational

Stable memory use

Few Full GCs

Fast Scavengers

Long lifetimes

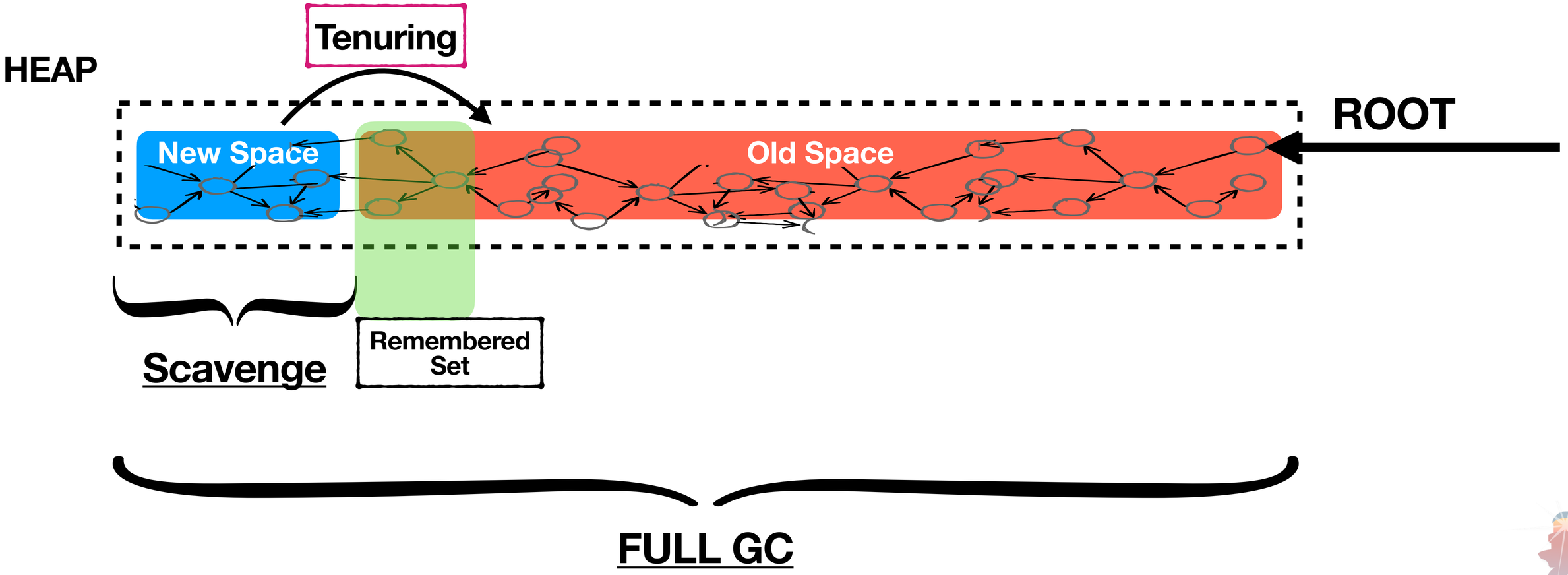
Memory-starved

Many Full GCs

Scavenger overhead

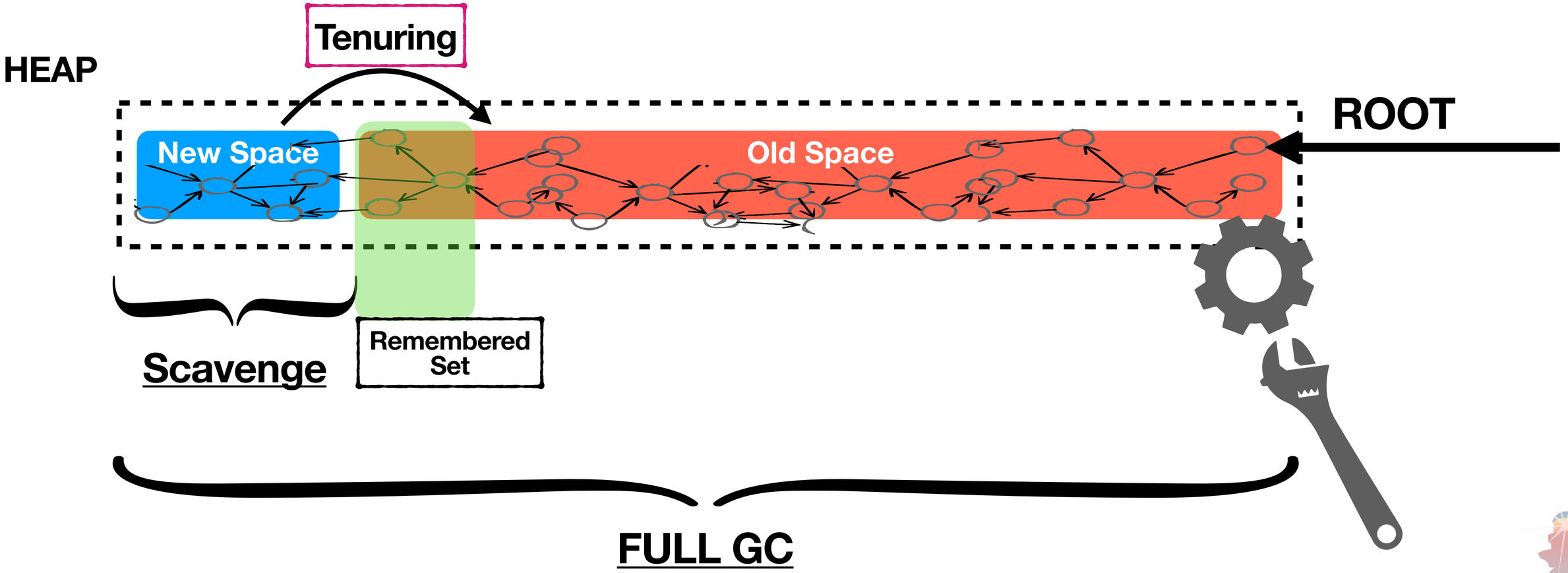
Pathological Allocation Pattern

Tuning the Garbage Collector



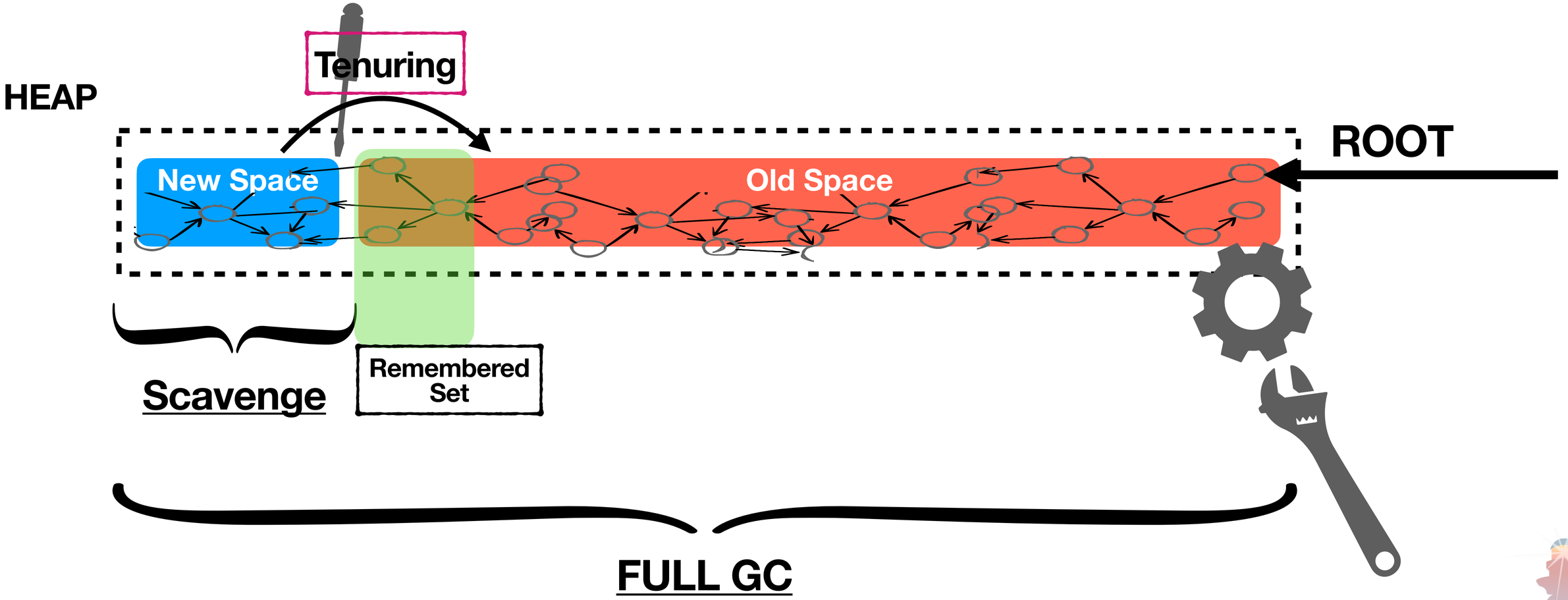
Pathological Allocation Pattern

Tuning the Garbage Collector



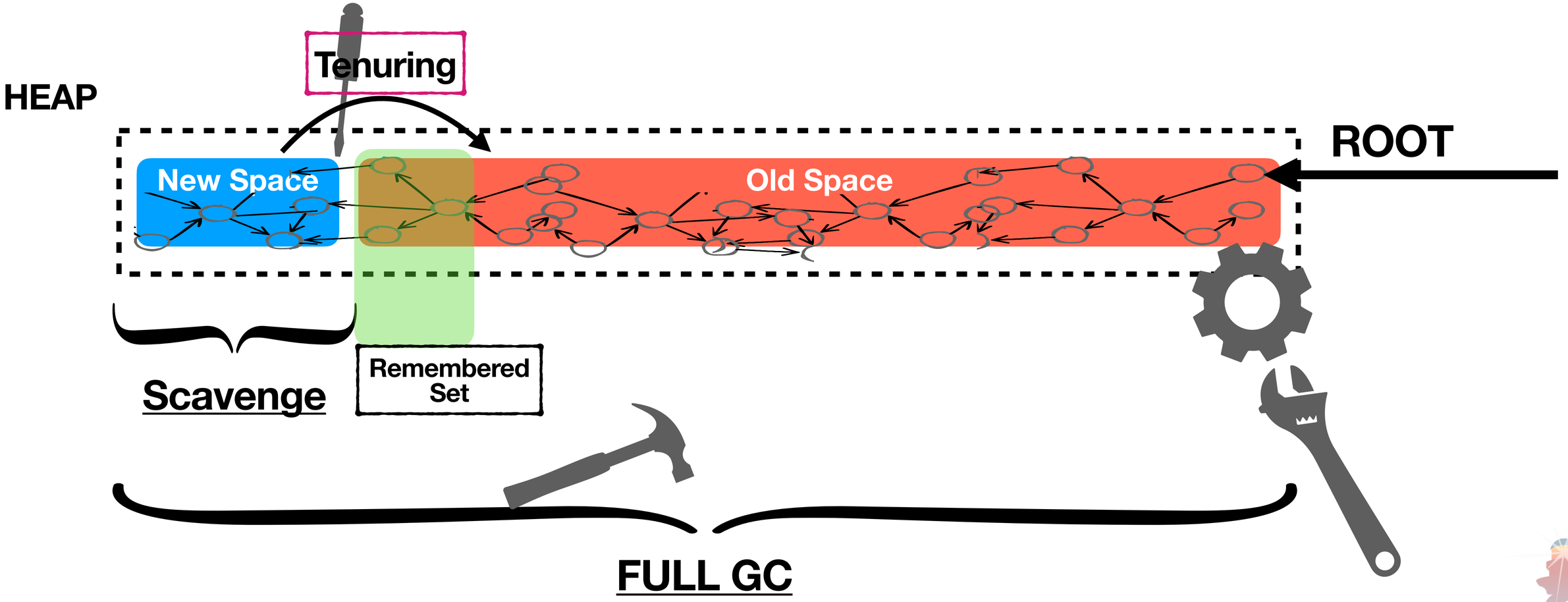
Pathological Allocation Pattern

Tuning the Garbage Collector



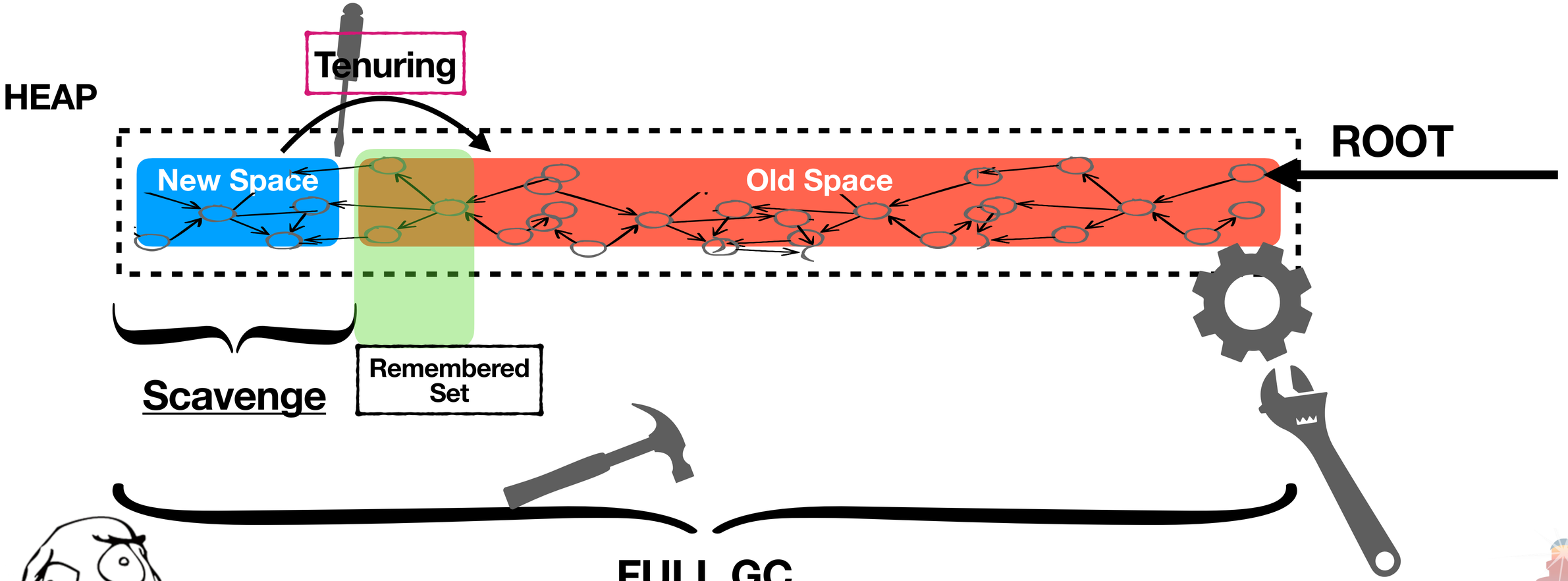
Pathological Allocation Pattern

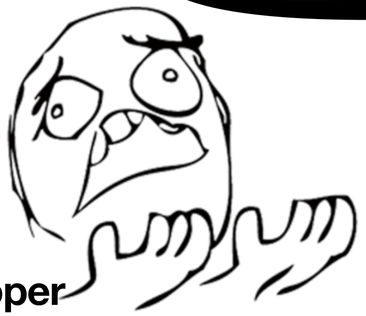
Tuning the Garbage Collector



Pathological Allocation Pattern

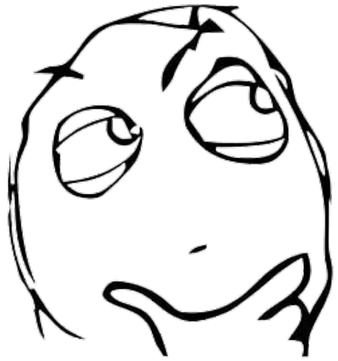
Tuning the Garbage Collector





Developer

How should I tune the GC parameters for my application?



Garbage Collector Tuning in Pathological Allocation Pattern Applications

Nahuel Palumbo[1], Sebastian Jordan Montaño[1], Guillermo Polito[1], Pablo Tesone[1] and Stéphane Ducasse[1]

[1]*Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL F-59000 Lille, France*

Abstract

Automatic memory management is often supported by Garbage Collectors (GC). GC usually impacts running application performance. For tuning properly, they expose some parameters to support the adaptation of their algorithms to specific applications' scenarios. In some cases, the developers should modify the GC parameter values to achieve high performance.

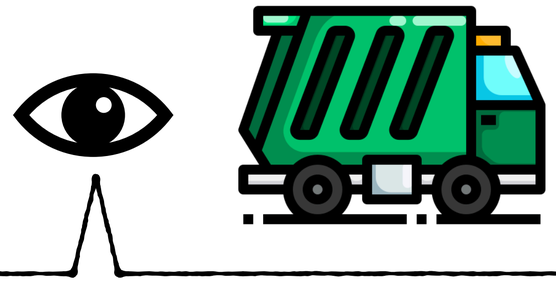
However, many application developers cannot be expected to perform expert analysis to determine which parameter values are the best for their application. There are techniques to find "good enough" parameter values. But, even if the overhead was reduced, it is still unknown the cause of the problem and how the GC tuning managed it.

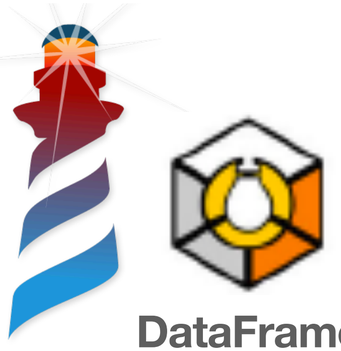
In this paper, we present a methodology to identify the causes of GC overhead in Pharo applications for tuning GC parameters. We describe the GC inside the PharoVM and its parameters, looking at how their variations change the allocation behaviour. We were able to analyse, identify and understand the GC performance issues present in one real application and suggest specific GC tuning actions. Using the suggested parameter values, we improved its performance by up to 12x and reduced GC overhead by up to 3.8x.

During the experiments, we also found: 1) a bug in the production PharoVM concerning the tenuring policy, 2) a misconception about one GC parameter even for the VM developers, and 3) some possible improvements for the current GC implementation.

Our methodology for GC tuning

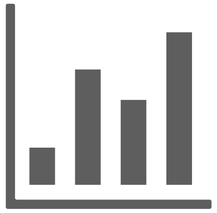
Profile GC events





DataFrame

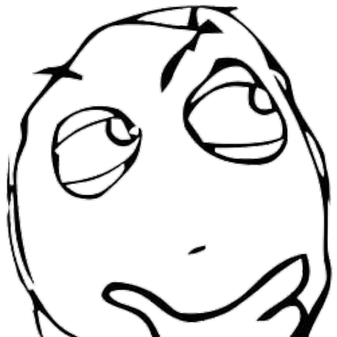
Logs



Plots

From Scavenges:
- Amount of memory used (before and after).
- Size of the Remembered Set (before and after).
- Tenuring info (amount of data - threshold).
- Executed time.

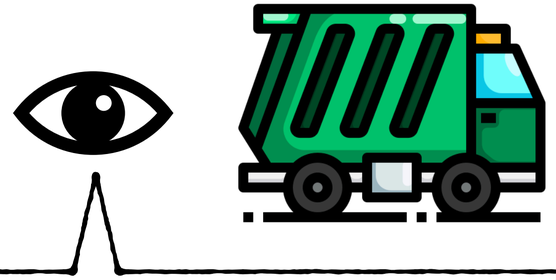
From FullGC:
- Time spent marking/sweeping/compacting.
- Executed time.



Developer

Our methodology for GC tuning

Profile GC events



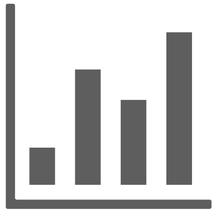
DataFrame

Logs

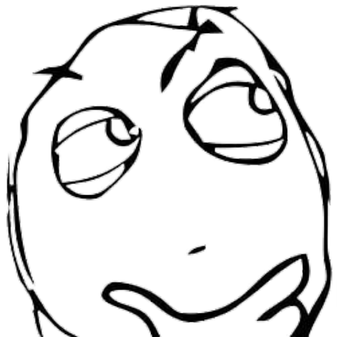
From Scavenges:
- Amount of memory used (before and after).
- Size of the Remembered Set (before and after).
- Tenuring info (amount of data - threshold).
- Executed time.

From FullGC:
- Time spent marking/sweeping/compacting.
- Executed time.

Tuning



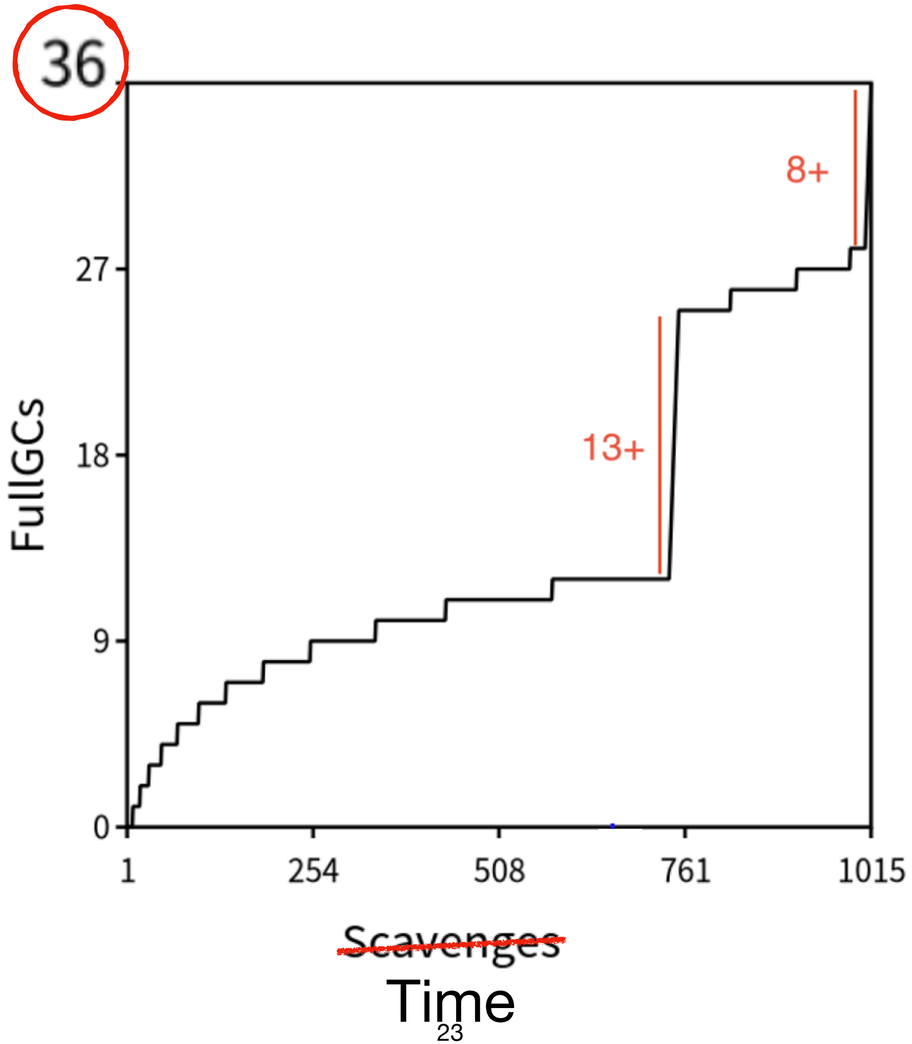
Plots



Developer

How Memory Grows

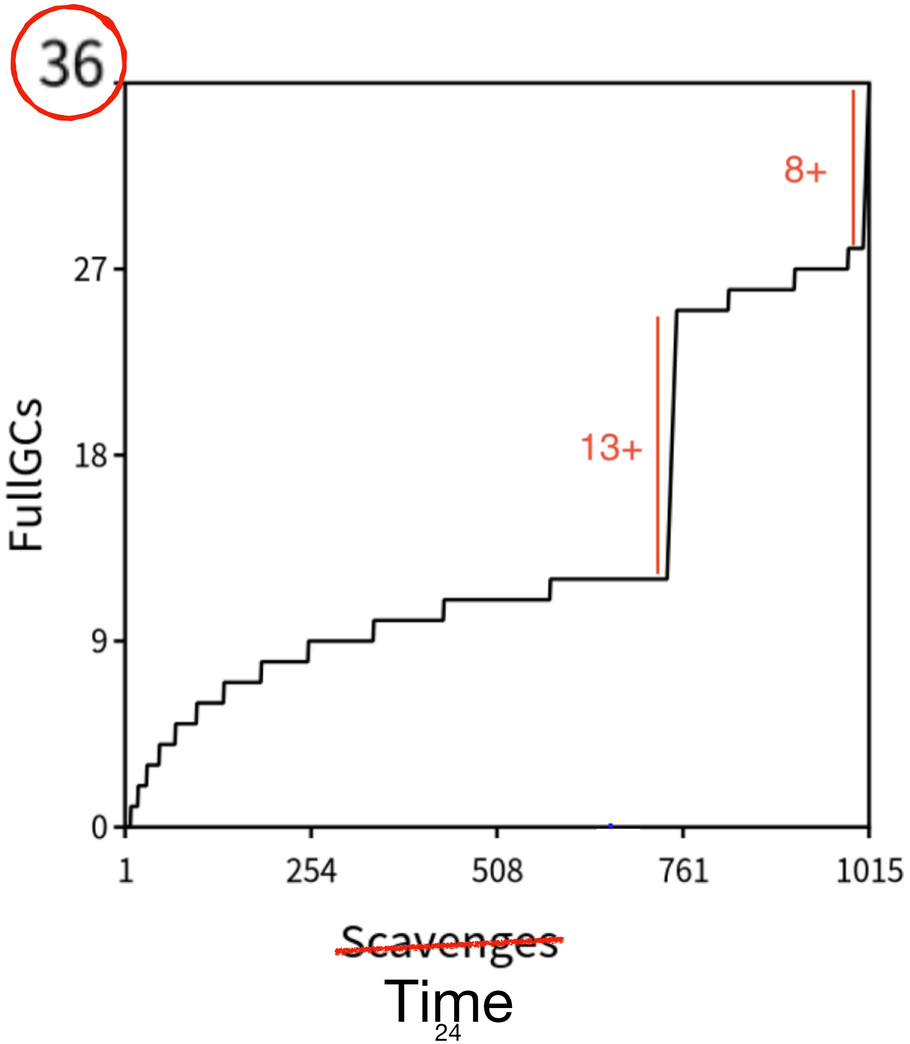
The overhead



How Memory Grows

The overhead

I run some FullGCs when memory grows so much



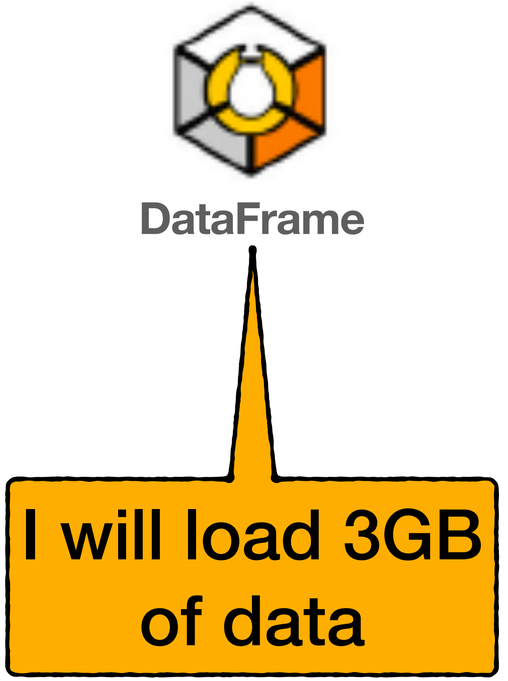
Don’t do that

How Memory Grows

The tuning solution

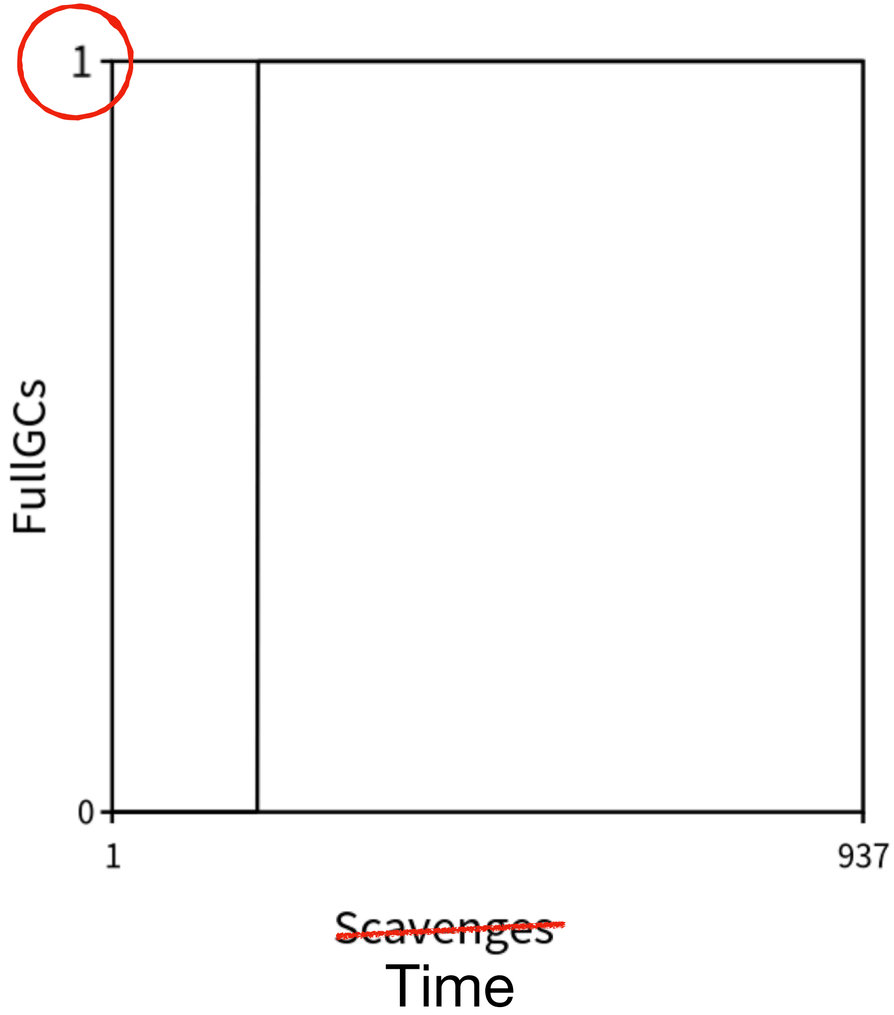
FullGC Ratio - Threshold for triggering a FullGC when the old space grows more than expected

Grow Headroom - Minimum amount of memory that the GC will order from the OS



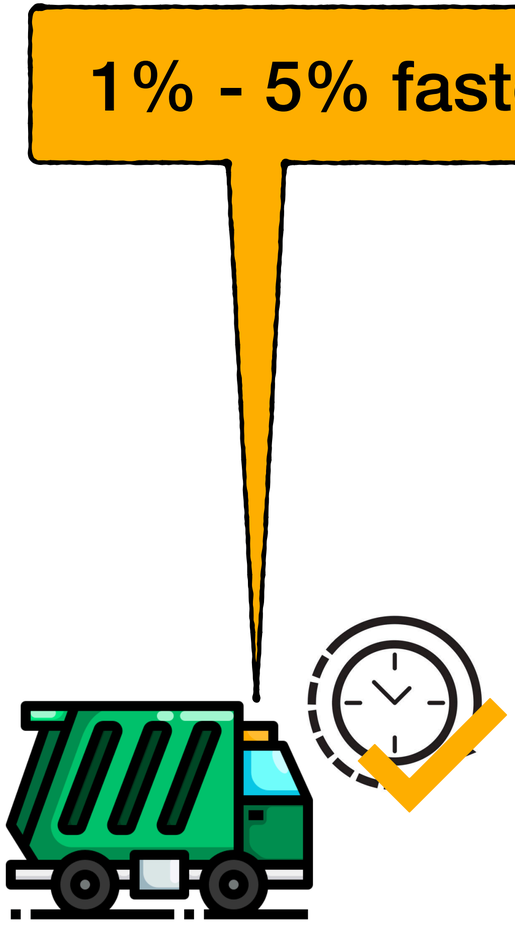
How Memory Grows

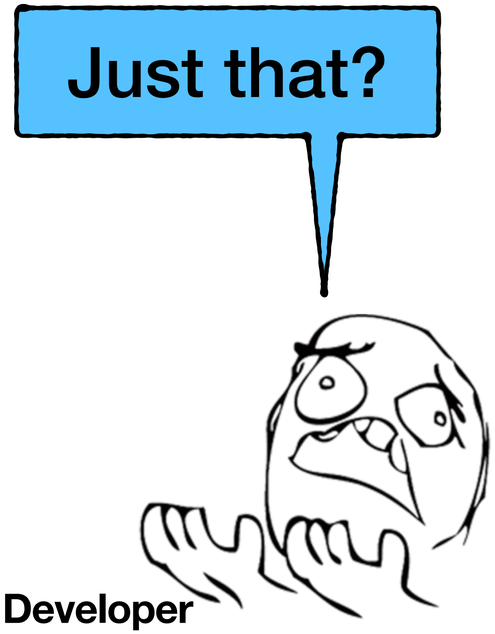
The tuning solution

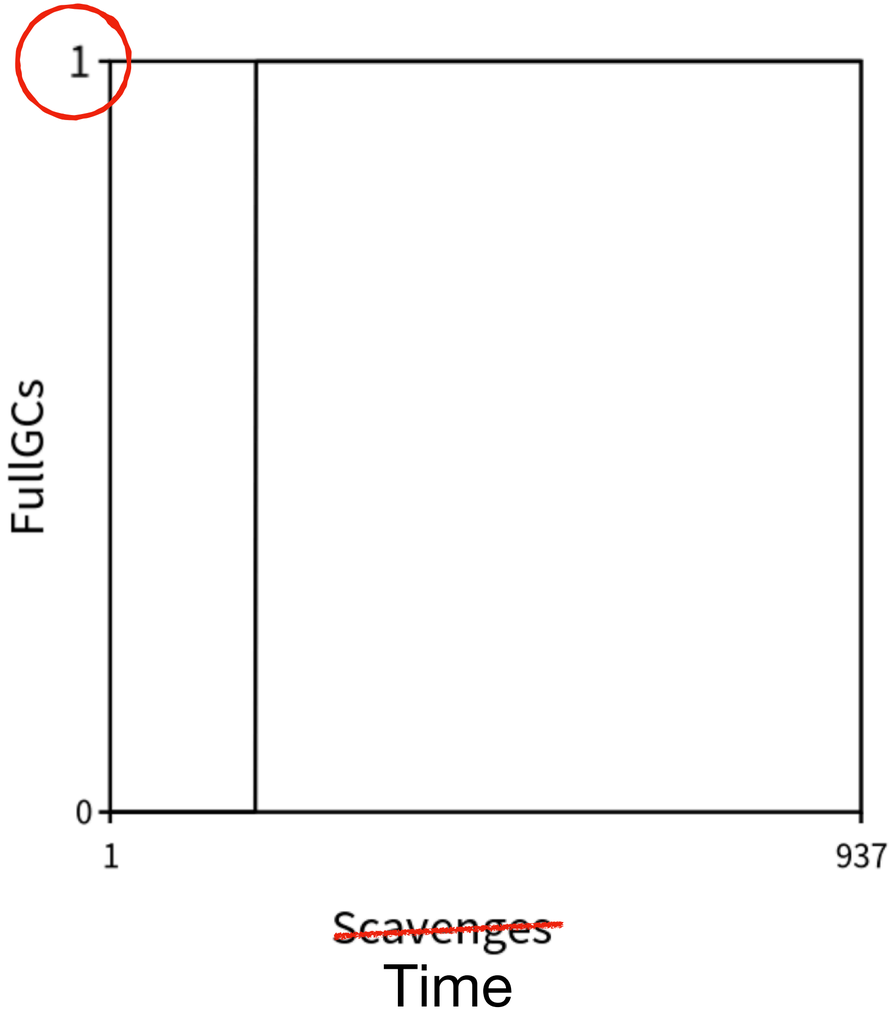


How Memory Grows

The tuning solution

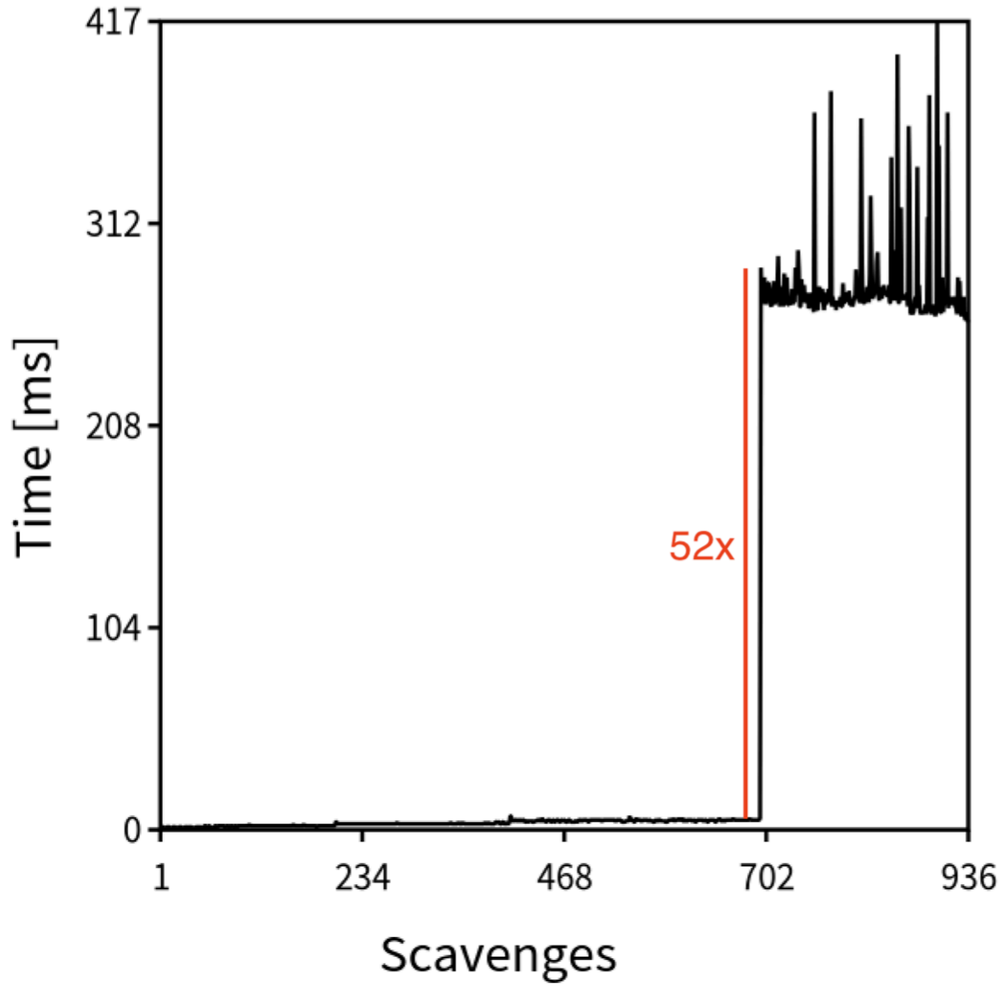






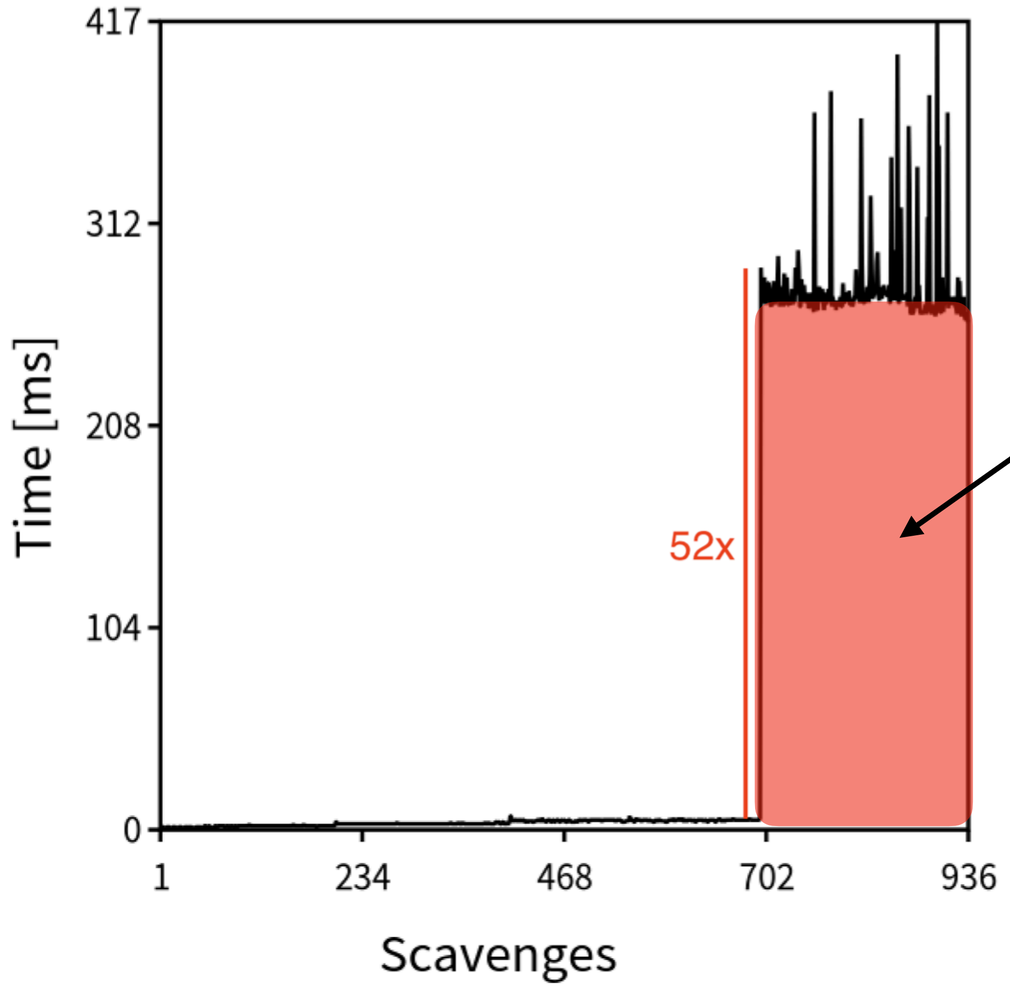
Deeper in the allocation pattern

Generational clash



Deeper in the allocation pattern

Generational clash

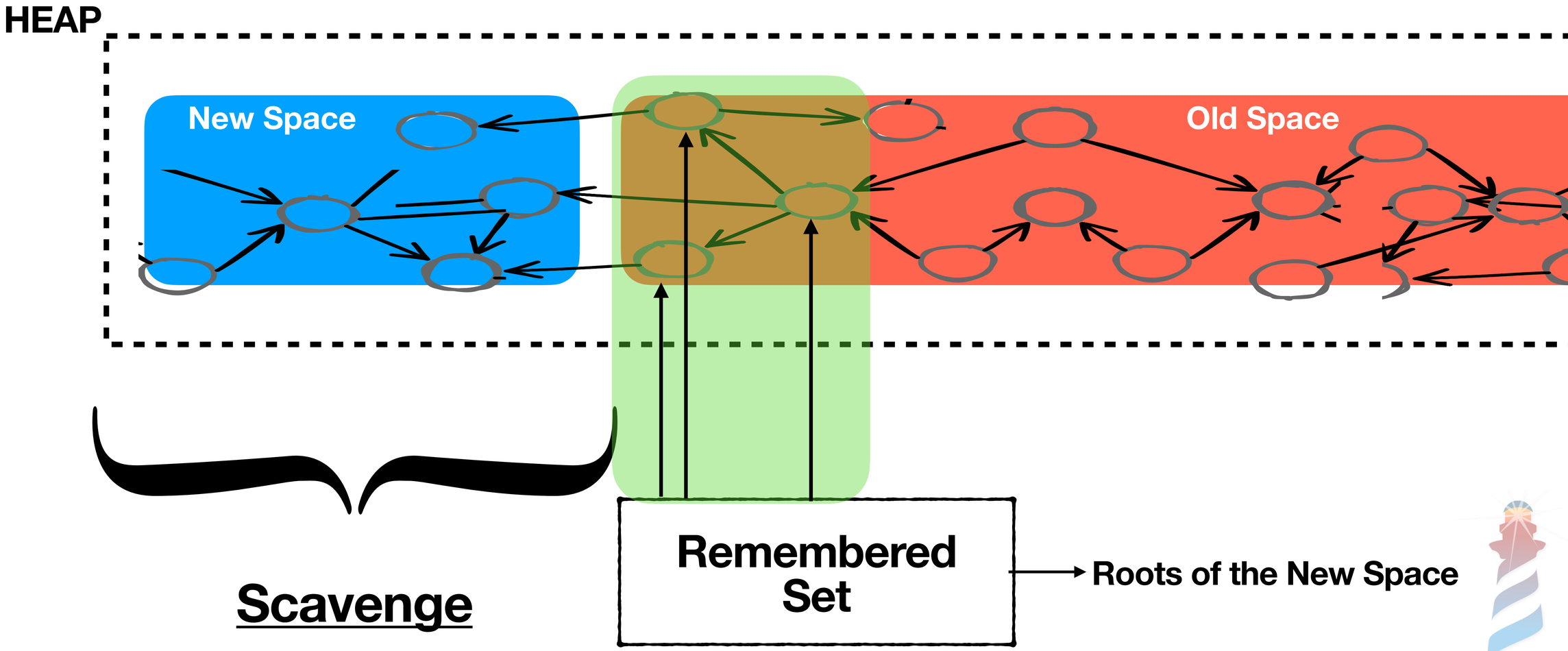


BIG

OVERHEAD

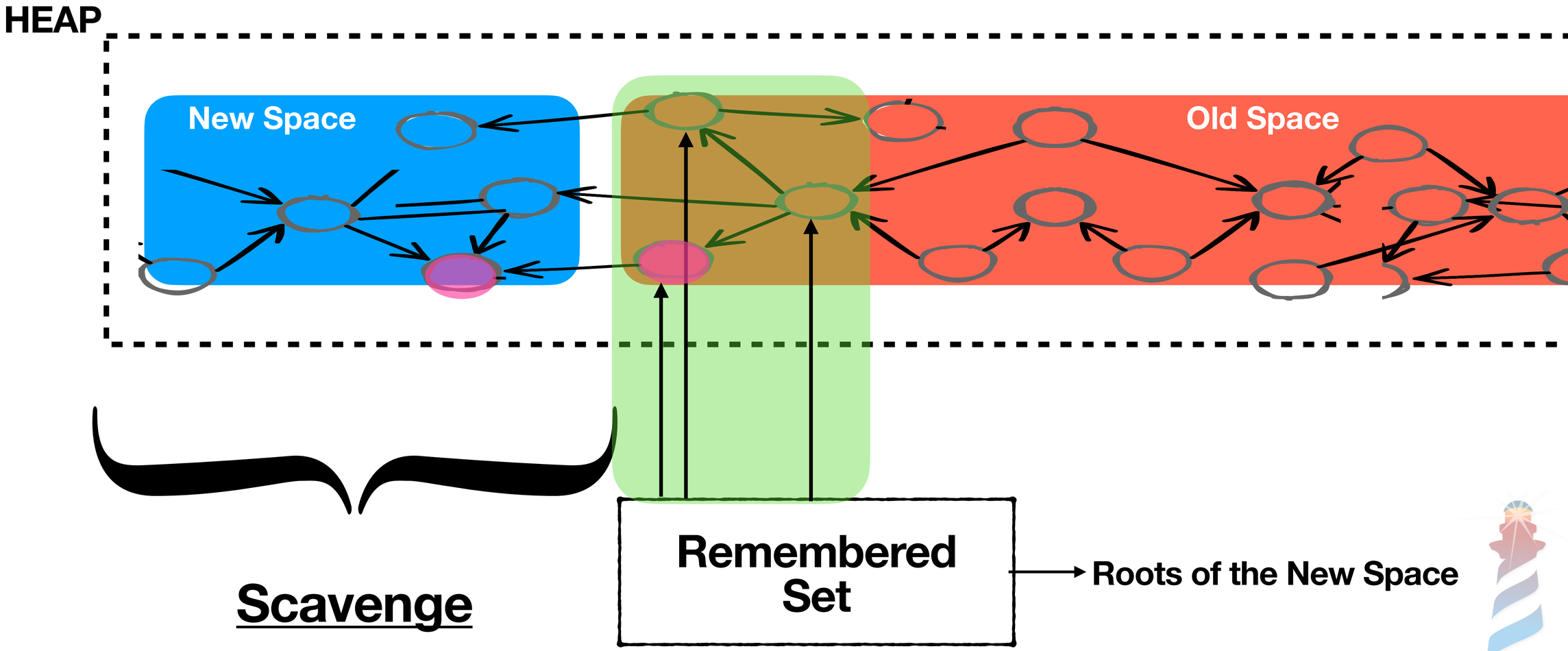
Deeper in the allocation pattern

Generational clash



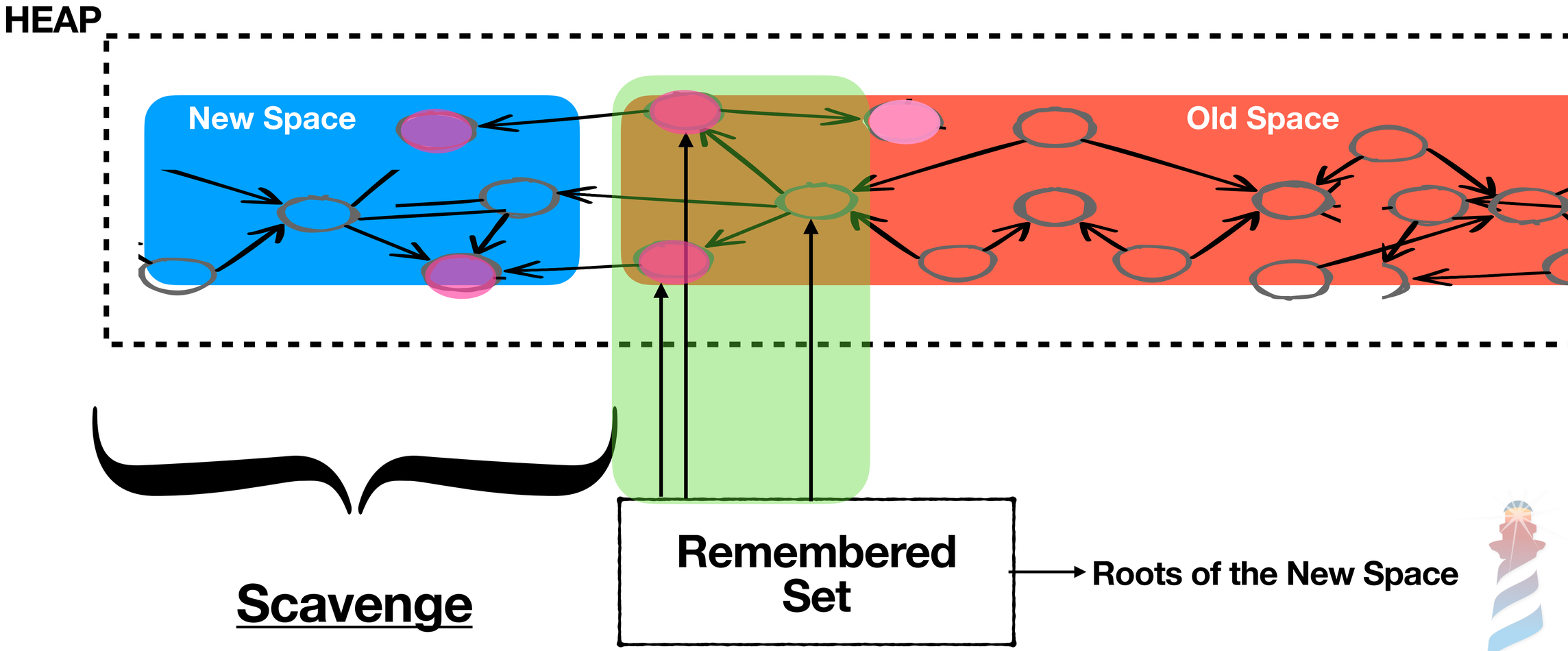
Deeper in the allocation pattern

Generational clash



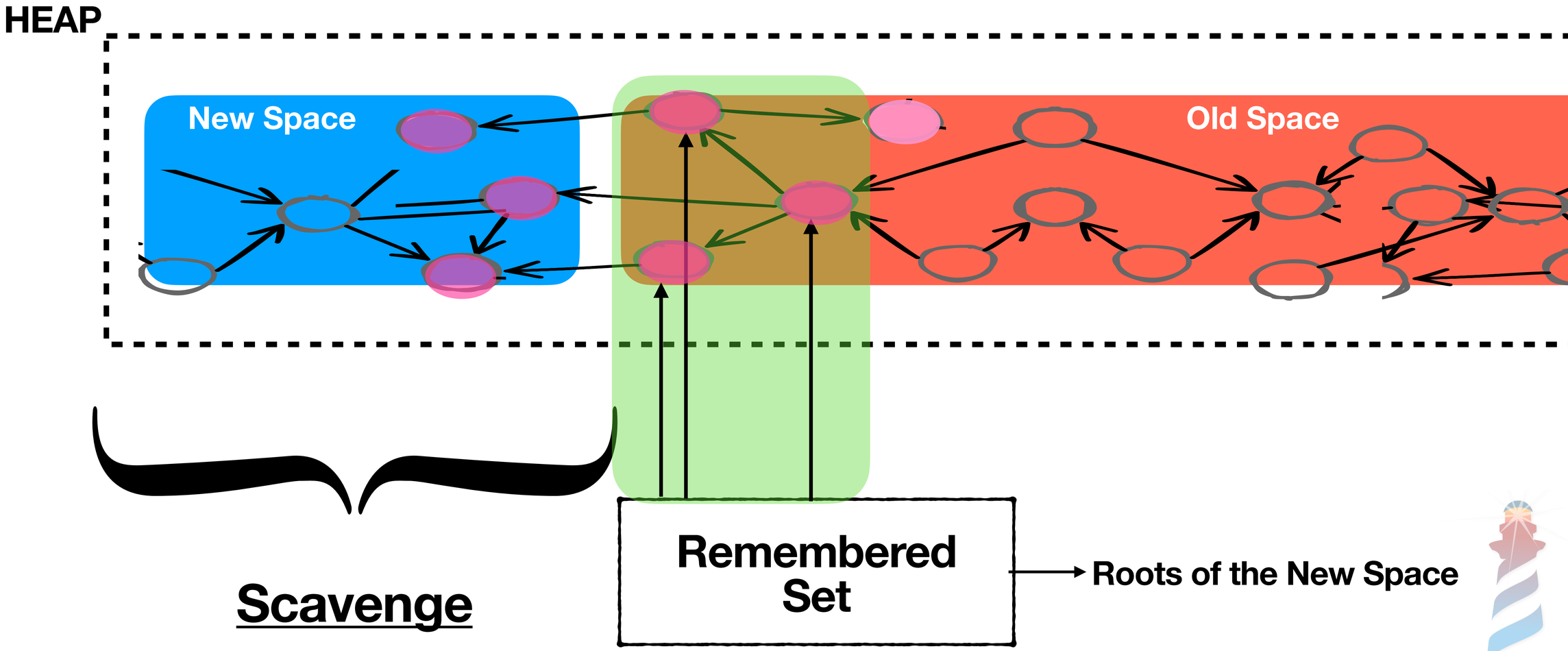
Deeper in the allocation pattern

Generational clash



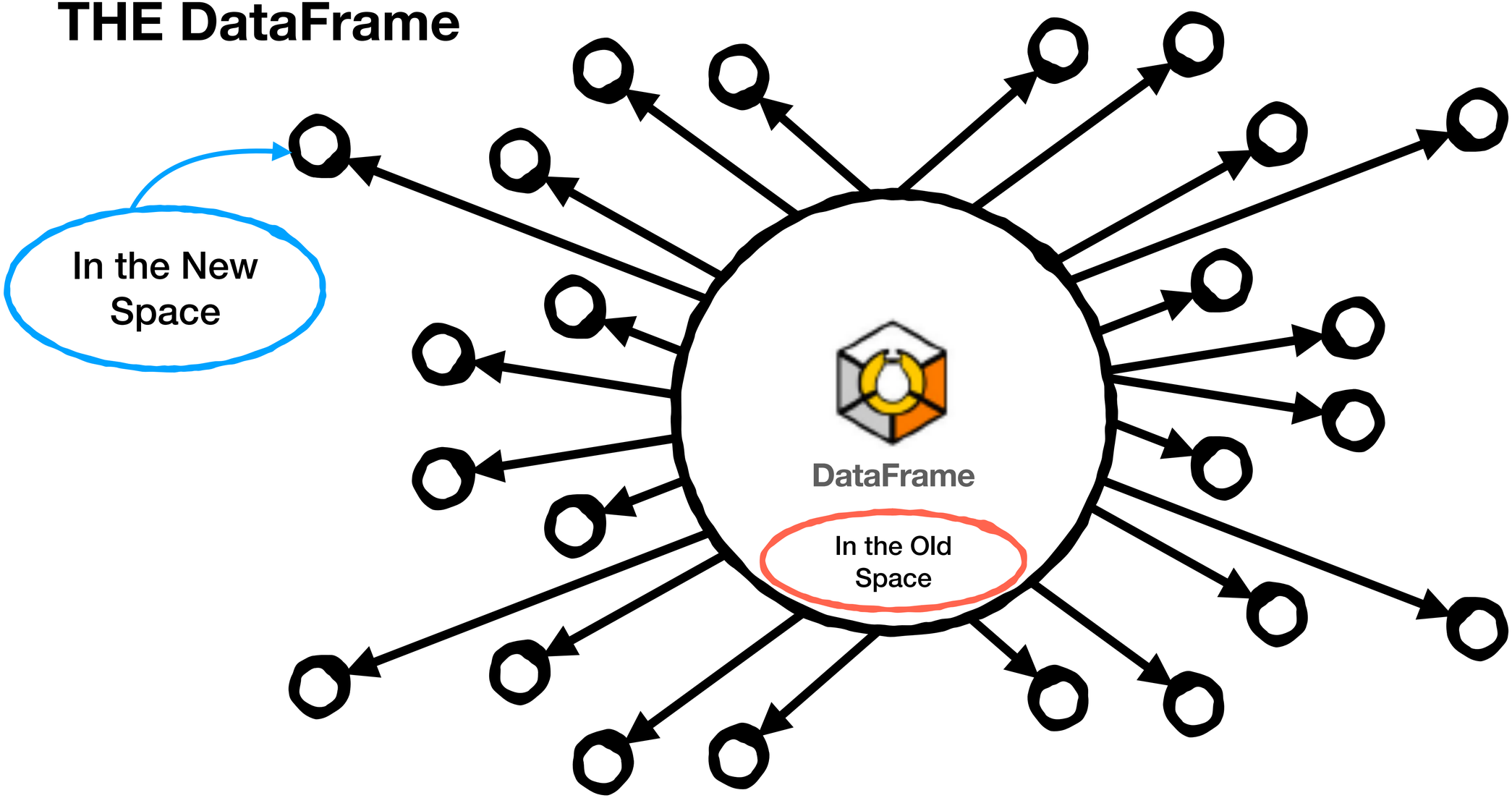
Deeper in the allocation pattern

Generational clash



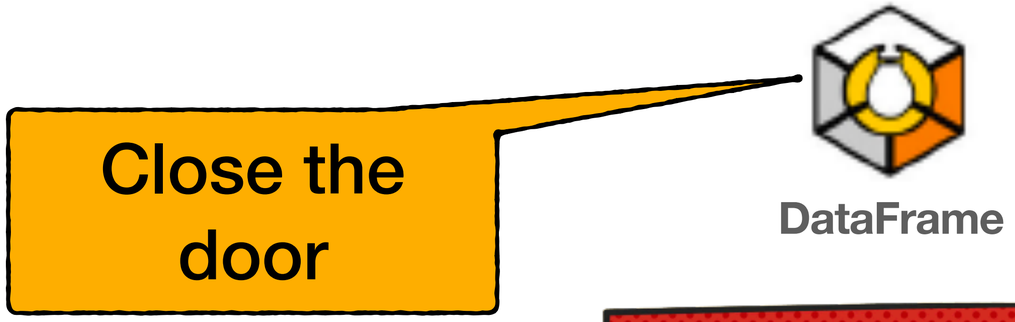
Deeper in the allocation pattern

THE DataFrame



Long Scavenges

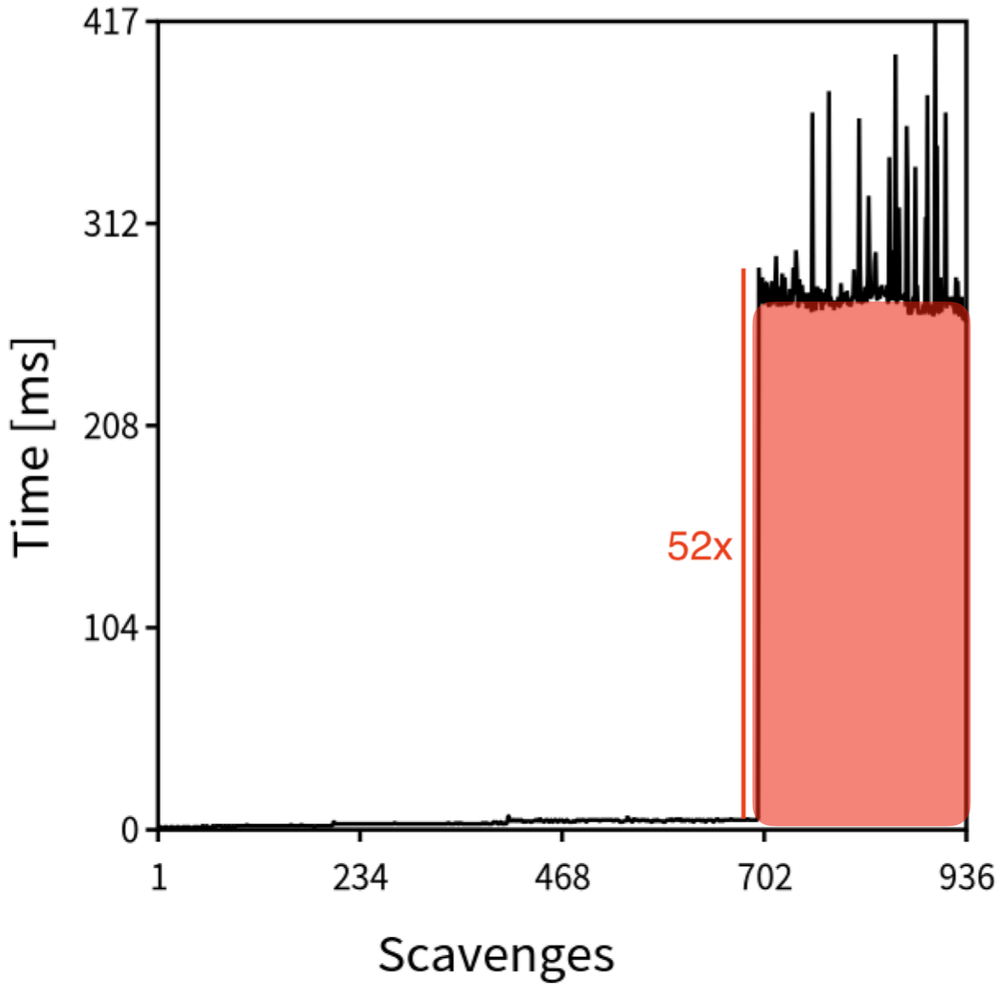
The tuning solution

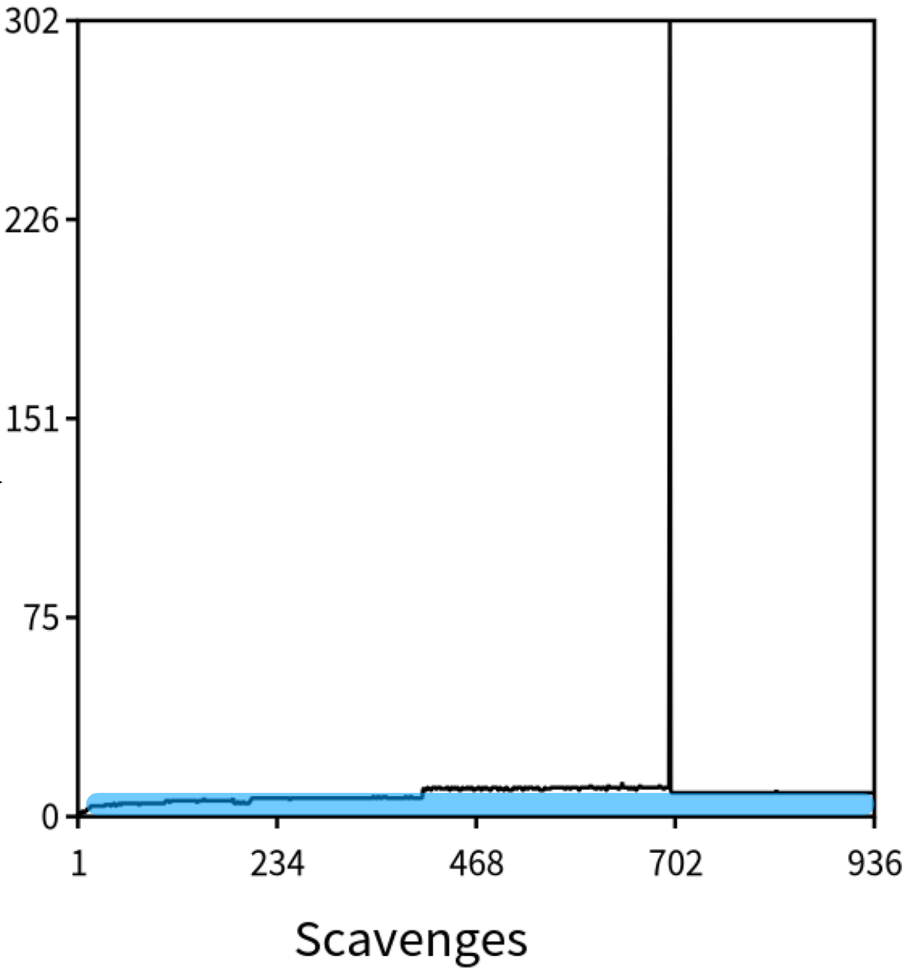


Tenuring threshold - Desired number of objects already in the New Space for tenuring to the Old Space

Long Scavenges

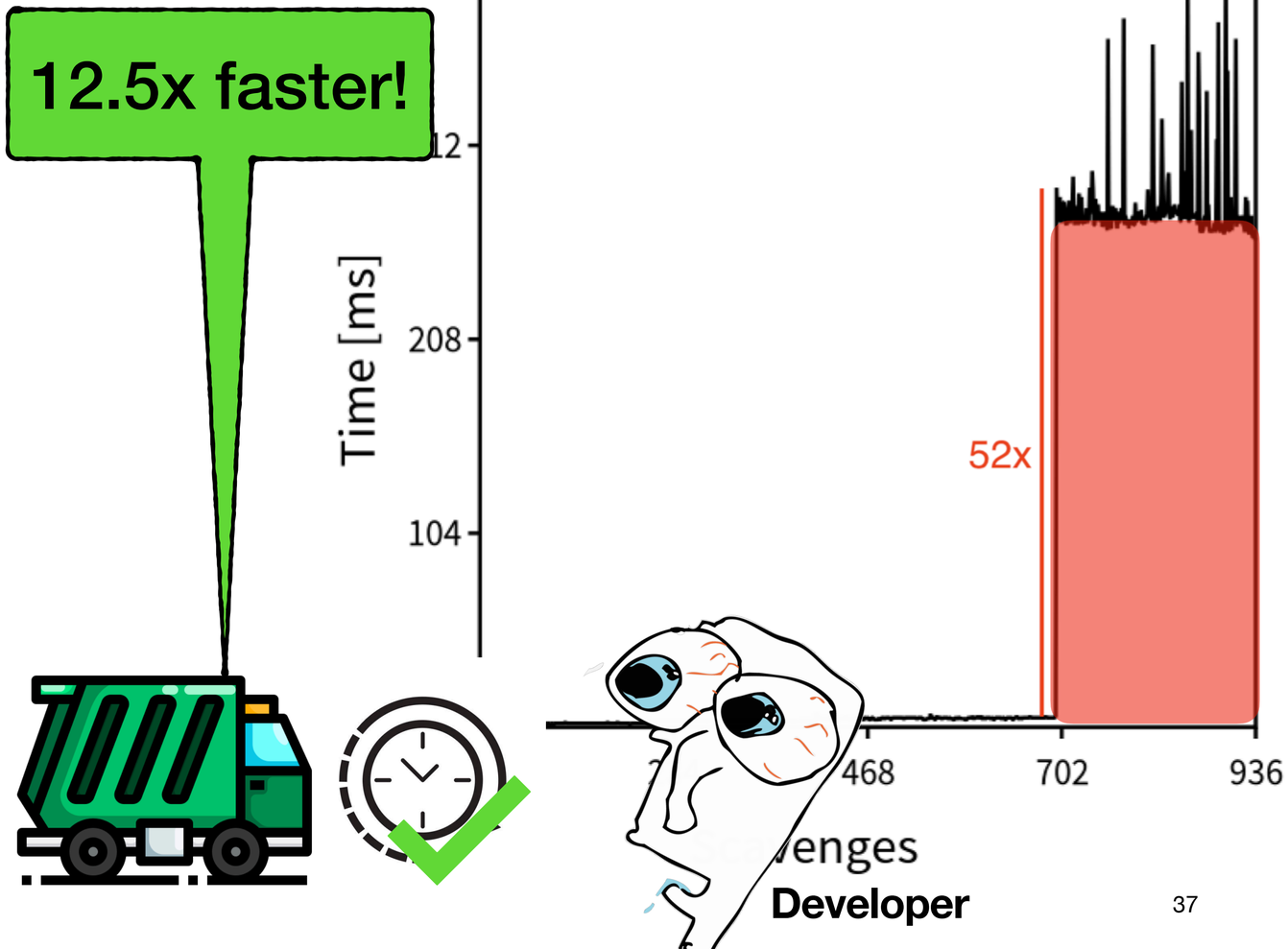
The tuning solution

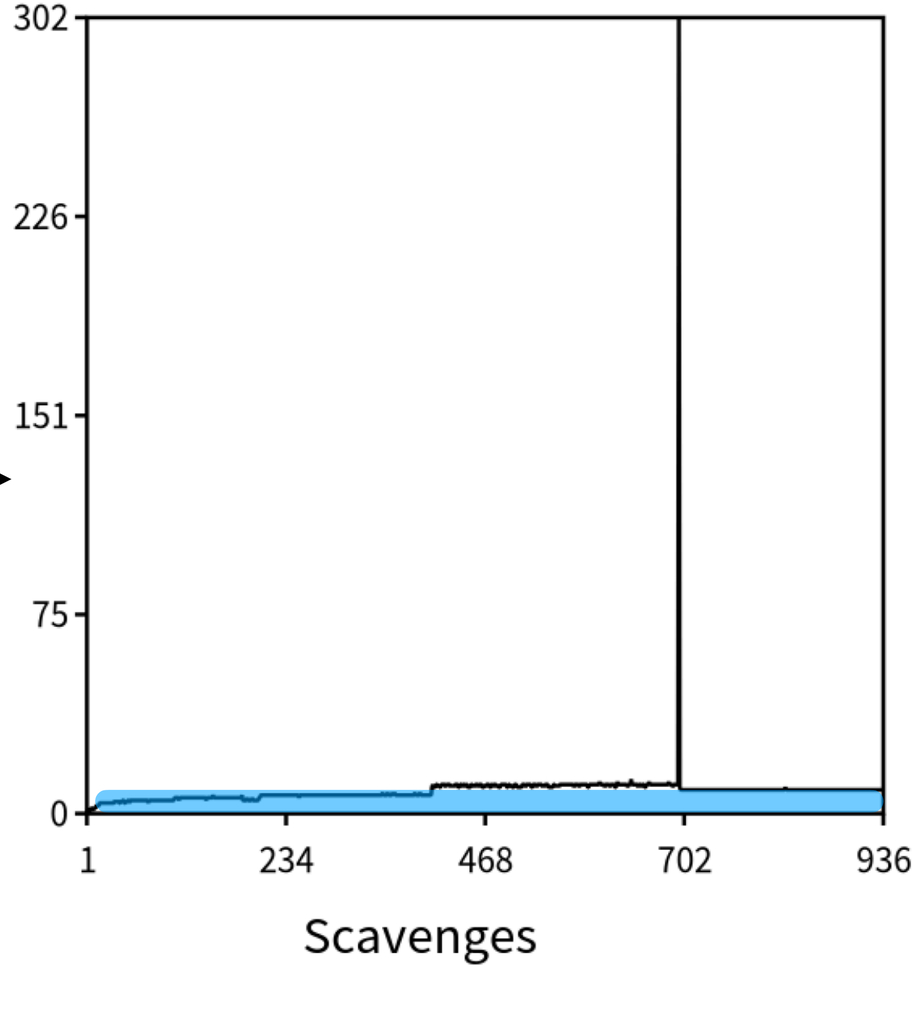




Long Scavenges

The tuning solution





Final result

1) Have an infinite FullGC ratio To reduce the number of FullGC when the Old Space grows.

2) Have a grow headroom equal to the loaded file To avoid many FullGC together.

3) Keep all survivors in the semi-space To tenure new objects to the Old Space quickly.

Data size	Total secs before	GC overhead before	Total secs after	GC overhead after
529 MB	43	16%	37 (1.1x)	5% (3.2x)
1.6 GB	150	25%	122 (1.2x)	7% (3.6x)
3.1 GB	5599 >1h30m	**92%**	440 (12.5x)	**24%** (3.8x)

~7mins

Understanding Garbage Collectors

in object-oriented programming

Nahuel Palumbo

PalumboN

nahuel.palumbo@inria.fr

CERCIRAS 2023

Inria · Evref · cnrs · Pharo Association · CRIStAL Centre de Recherche en Informatique, Signal et Automatique de Lille · Université de Lille