Heaping on the Complexity

Adventures in Ruby's GC Compactor

Matt Valentine-House
Shopify - Ruby Infrastructure team
@eightbitraptor

Outline

- Background

Outline

- Background
- The Problem and Solutions

Outline

- Background
- The Problem and Solutions
- What happened next & Memory Allocation update

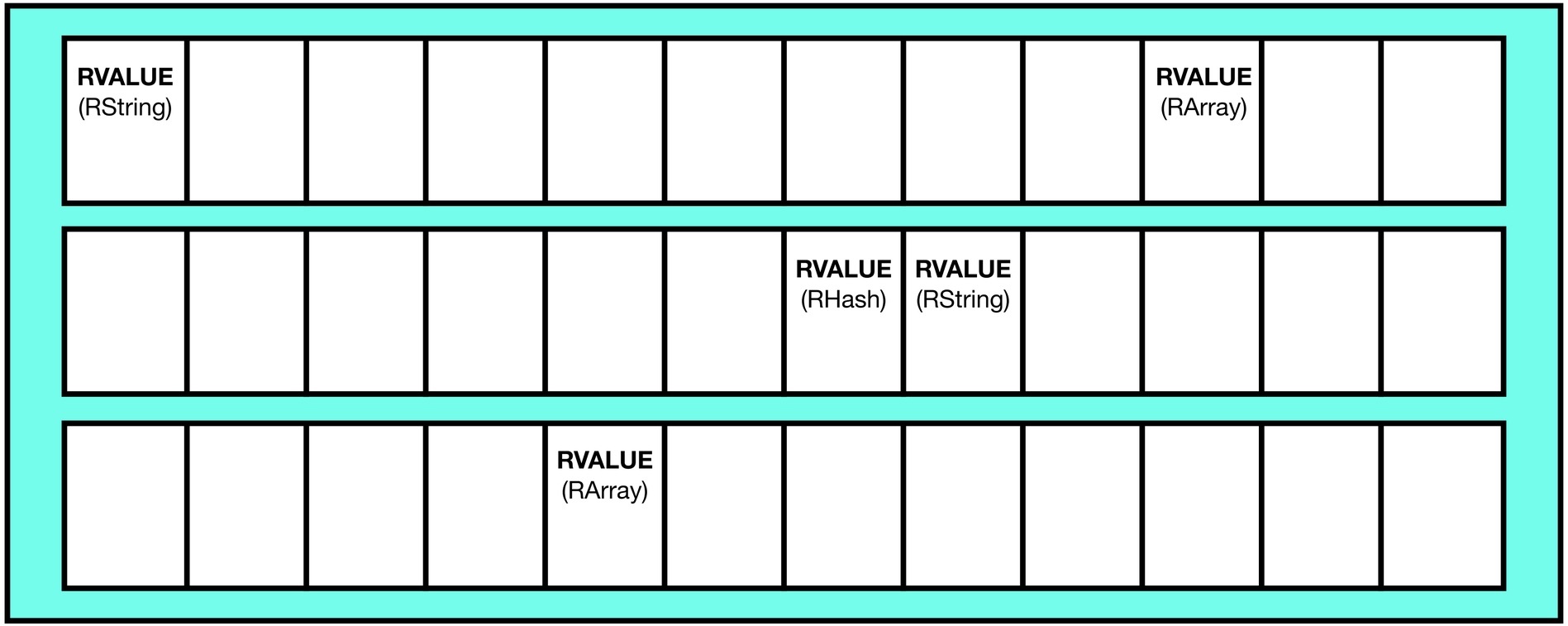
Outline

- Background
- The Problem and Solutions
- What happened next?
- Introduce Heapviz

Background

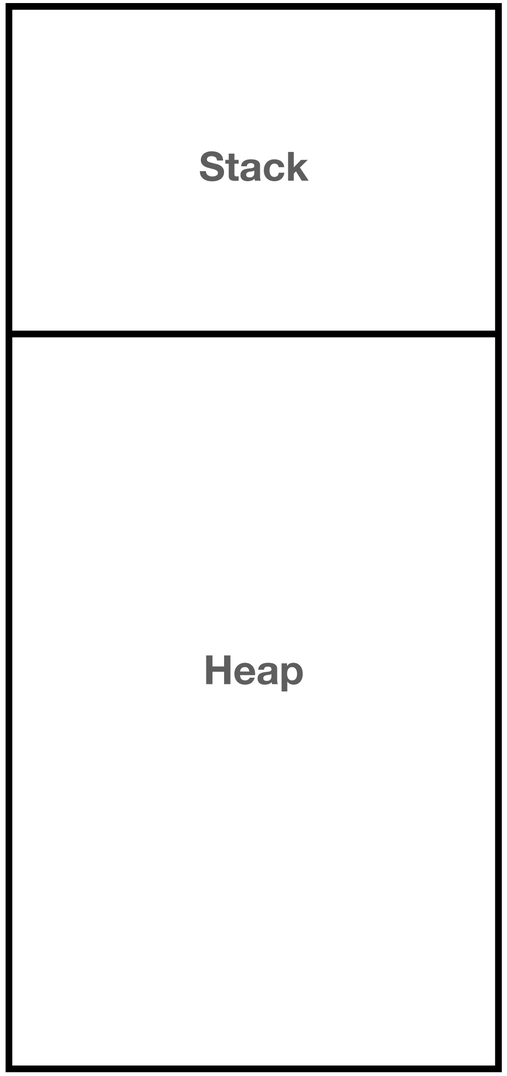
Memory

Background - The Heap



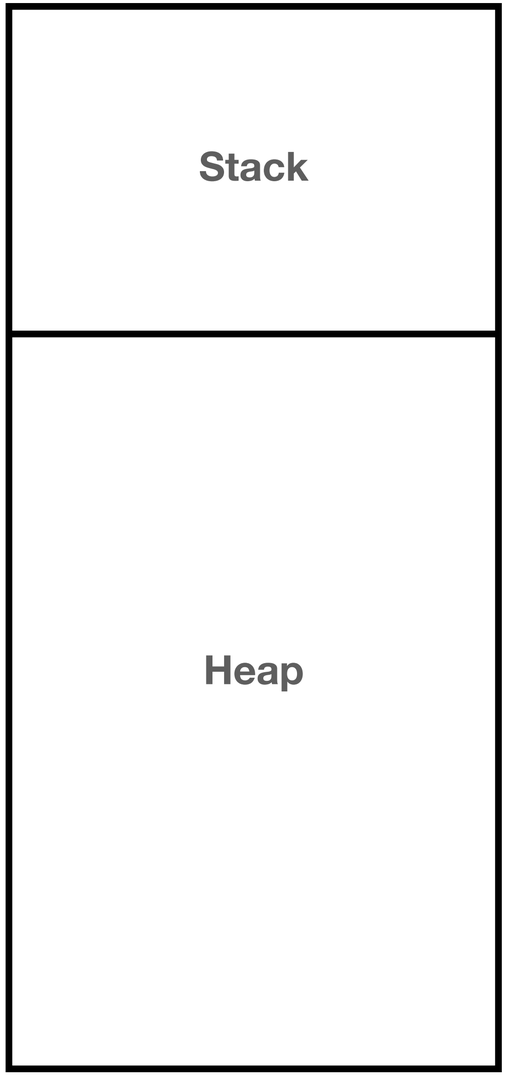
Background - Heap & Embedded Allocation

Background - Memory, an OS perspective



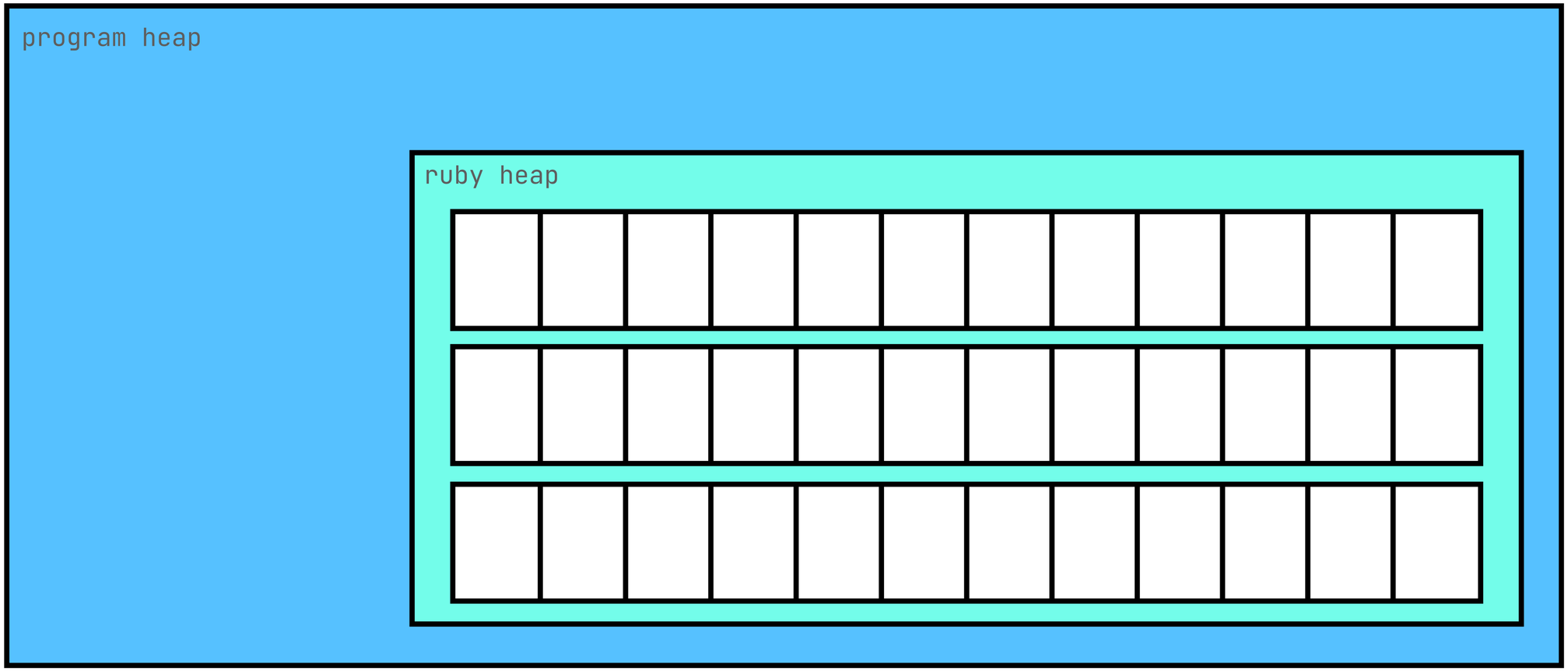

```
#include <stdlib.h>

int
main(int argc, char *argv)
{



}
```

Background - Memory, an OS perspective



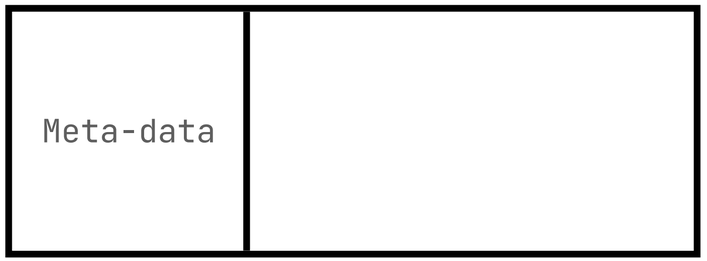

```c
#include <stdlib.h>

int
main(int argc, char *argv)
{
   // allocate space for a string of 100 characters in the heap
   char * dynamic_string = (char *)malloc(100);

   // release the memory back to the OS when we're done
   free(dynamic_string);
}
```

Background - The Ruby Heap



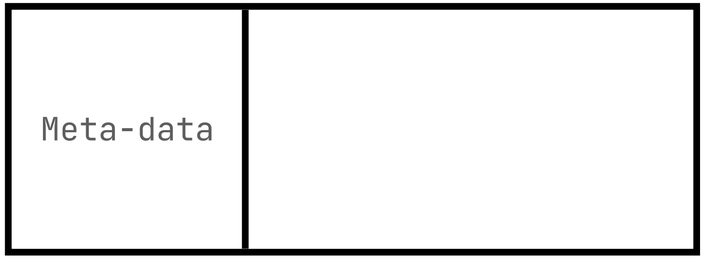
Background - Heap & Embedded Allocation

RString



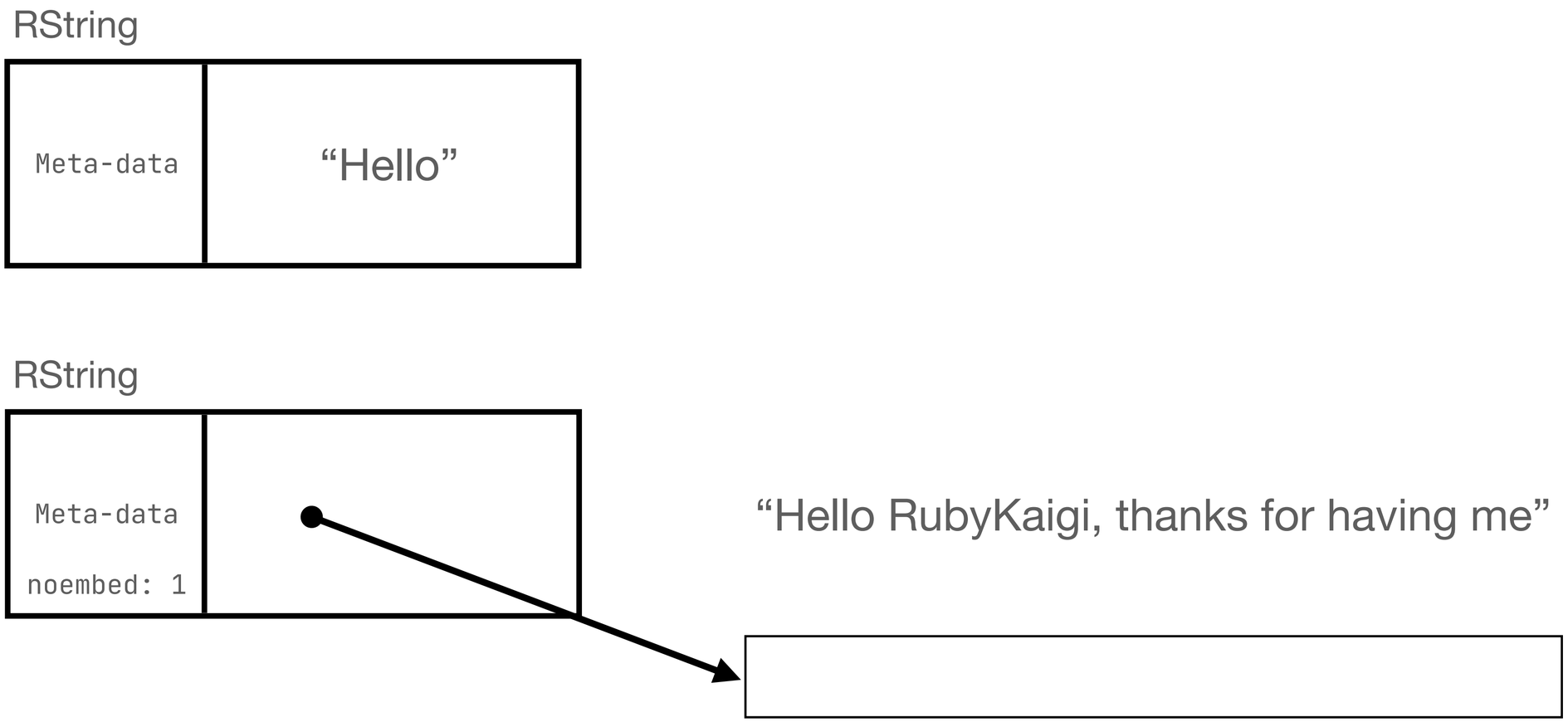
“Hello”

RString

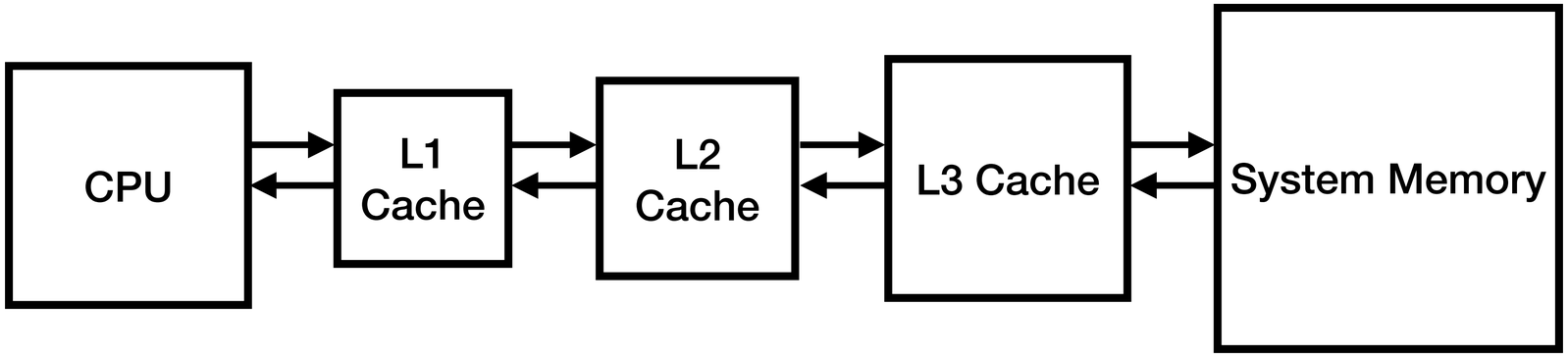


“Hello RubyKaigi, thanks for having me”

Background - Heap & Embedded Allocation

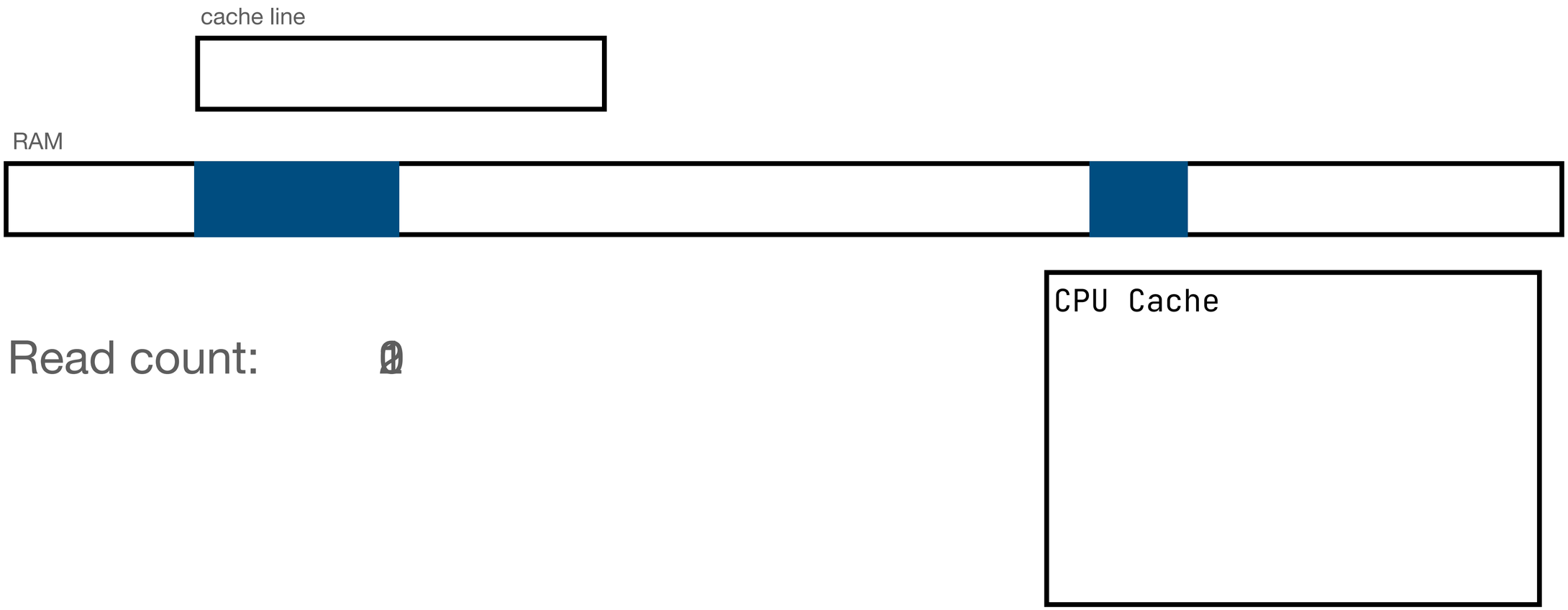


Background - CPU caches



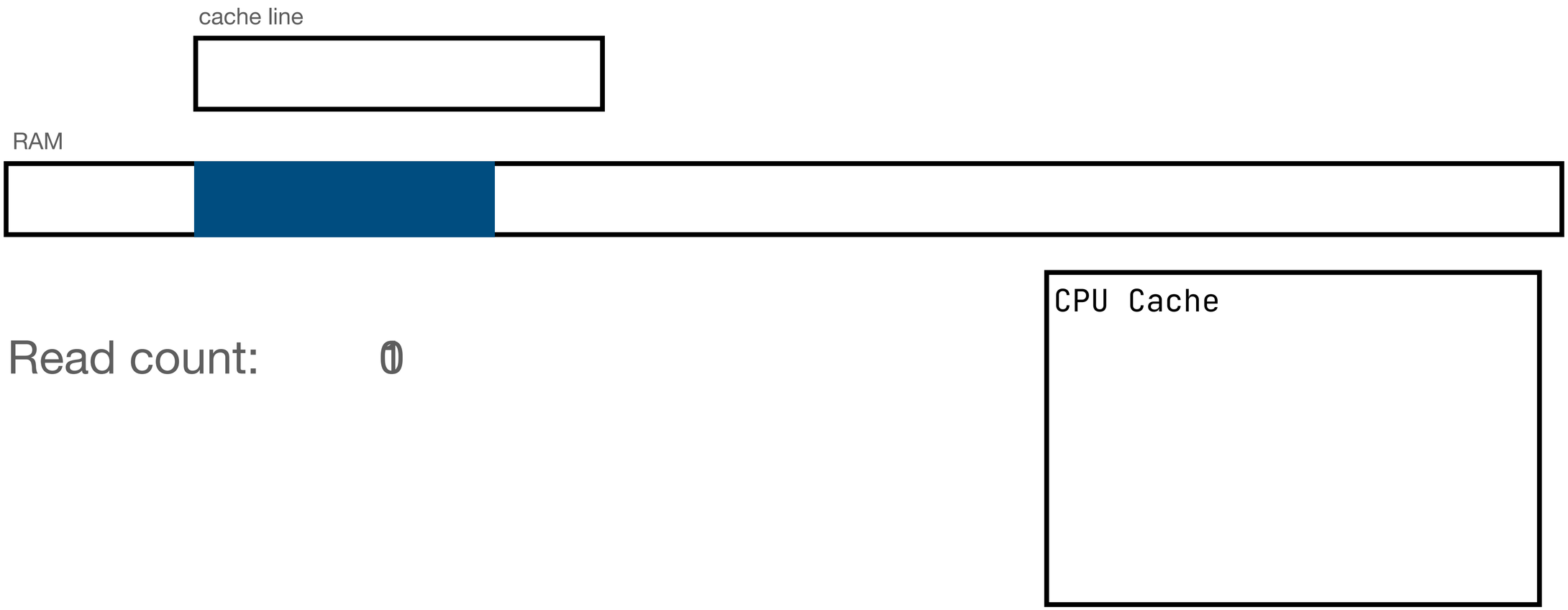
- **CPU's have multiple levels of cache**
- **The CPU cores read data from the closest cache**
- **A "cache miss" is when the data requested isn't in the desired cache.**
- **Poor cache performance can impact the performance of our programs**

Background - CPU Caches



Read count: 0

Background - CPU Caches

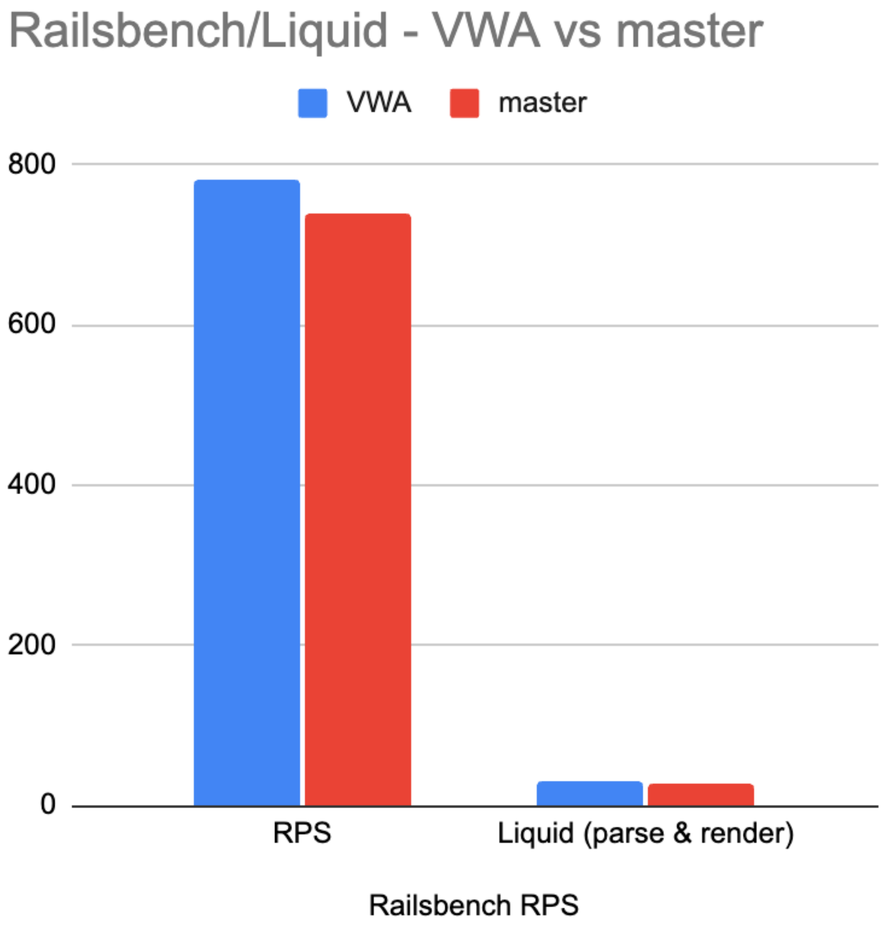


Read count: 0

Background - CPU Caches

- **Heap allocated objects have poor locality**.
  - Objects with poor locality require **more reads when processing**.
  - Therefore poor locality has a **performance impact** on our programs.
- **Shopify are trying to solve this problem using Variable Width Allocation**.
  - VWA changes the heap structure to **enable more objects to be embedded**.
  - Embedded objects have **better data locality**, and are more **performant**

Background - Variable Width Allocation



VWA Results so far:

- **~5% improvement in Railsbench RPS**
- **~2-3% improvement in Liquid performance**

Background

Garbage Collection (briefly)

Background - Garbage Collection

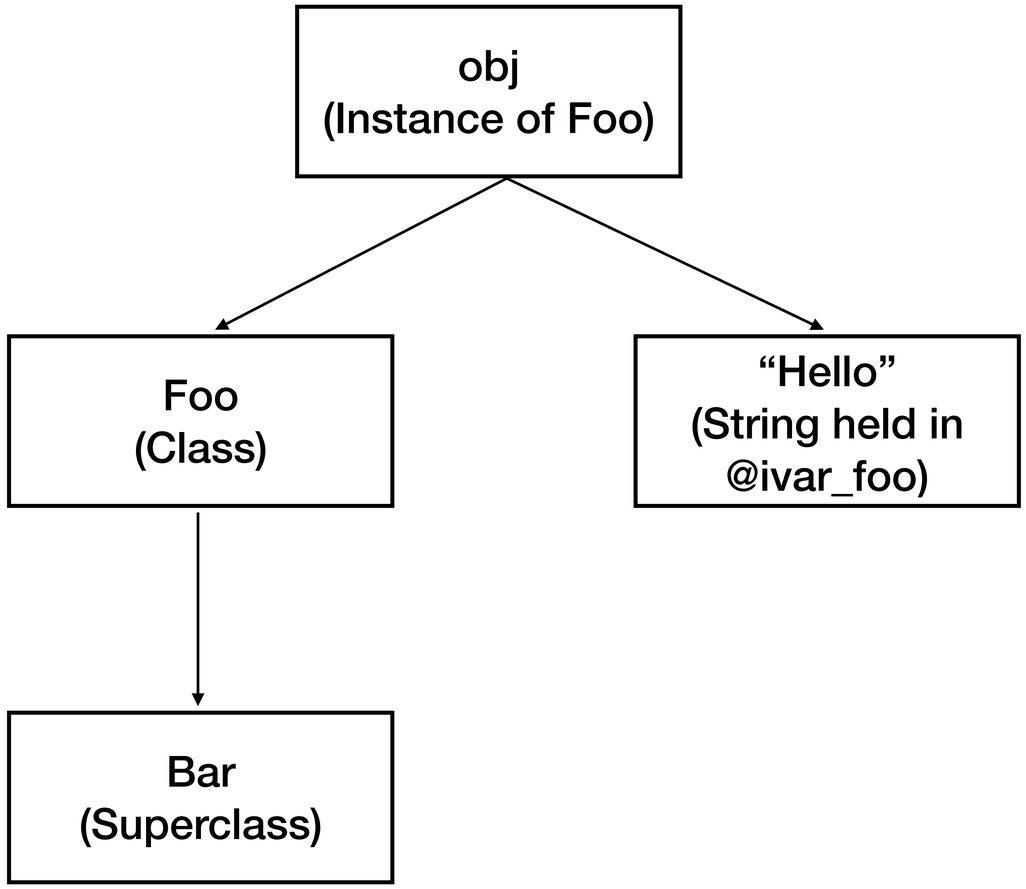
*"**Garbage collection (GC)** is a form of **automatic memory management**.*

The garbage collector attempts to reclaim memory which was allocated by the program, but is no longer referenced"

—*Wikipedia*

Background - Garbage Collection

```
class Foo < Bar
  def initialize(my_foo)
    @ivar_foo = my_foo
  end
end

obj = Foo.new("Hello")
```



Background - Garbage Collection

Ruby's GC is:

- **Stop the World** - Execution of our program is paused while GC happens

Background - Garbage Collection

Ruby's GC is:

- **Stop the World** - Execution of our program is paused while GC happens
- **Mark-Sweep** - reachable objects are marked, unmarked objects are swept

Background - Garbage Collection

Ruby's GC is:

- **Stop the World** - Execution of our program is paused while GC happens
- **Mark-Sweep** - reachable objects are marked, unmarked objects are swept
- **Generational** - Young objects are collected more frequently than Old objects

Background - Garbage Collection

Ruby's GC is:

- **Stop the World** - Execution of our program is paused while GC happens
- **Mark-Sweep** - reachable objects are marked, unmarked objects are swept
- **Generational** - Young objects are collected more frequently than Old objects
- **Incremental** - GC pauses are spread over the runtime of the program

Background - Garbage Collection

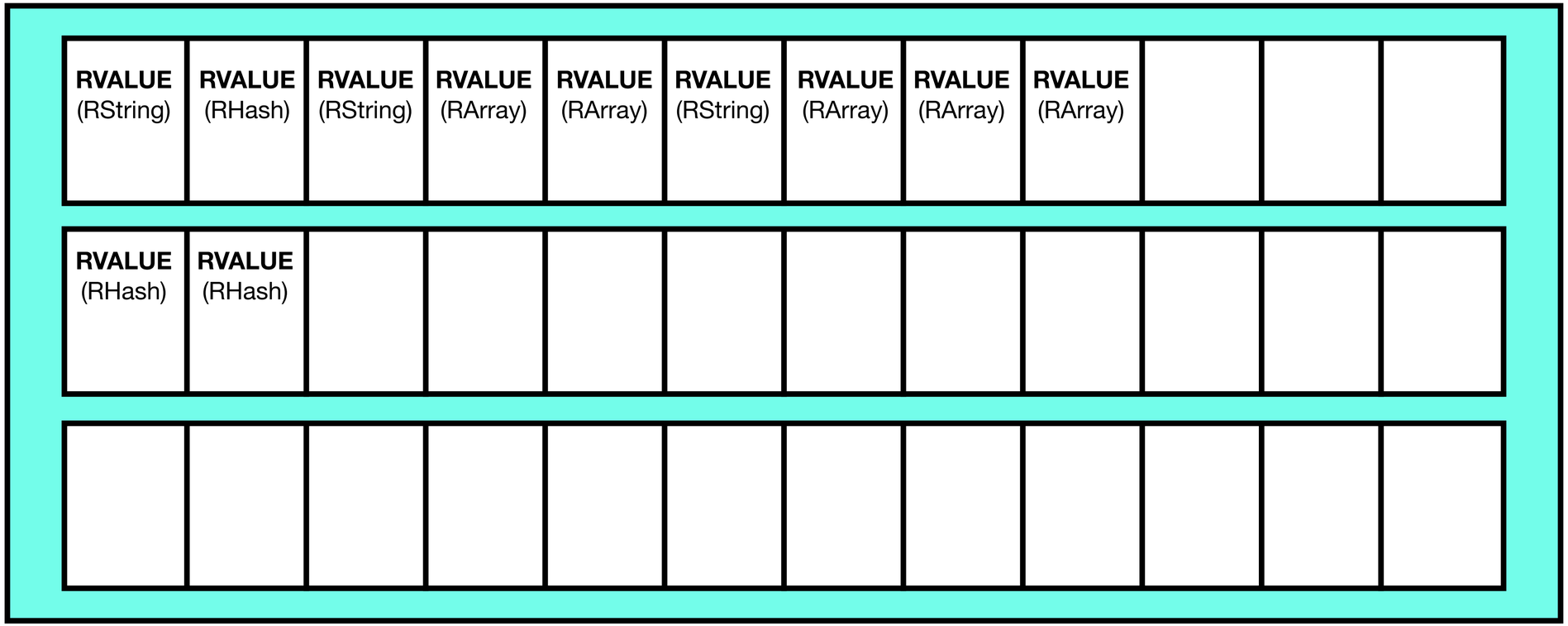
Ruby's GC is:

- **Stop the World** - Execution of our program is paused while GC happens
- **Mark-Sweep** - reachable objects are marked, unmarked objects are swept
- **Generational** - Young objects are collected more frequently than Old objects
- **Incremental** - GC pauses are spread over the runtime of the program
- **Compacting** - Objects are moved closer together to avoid heap fragmentation

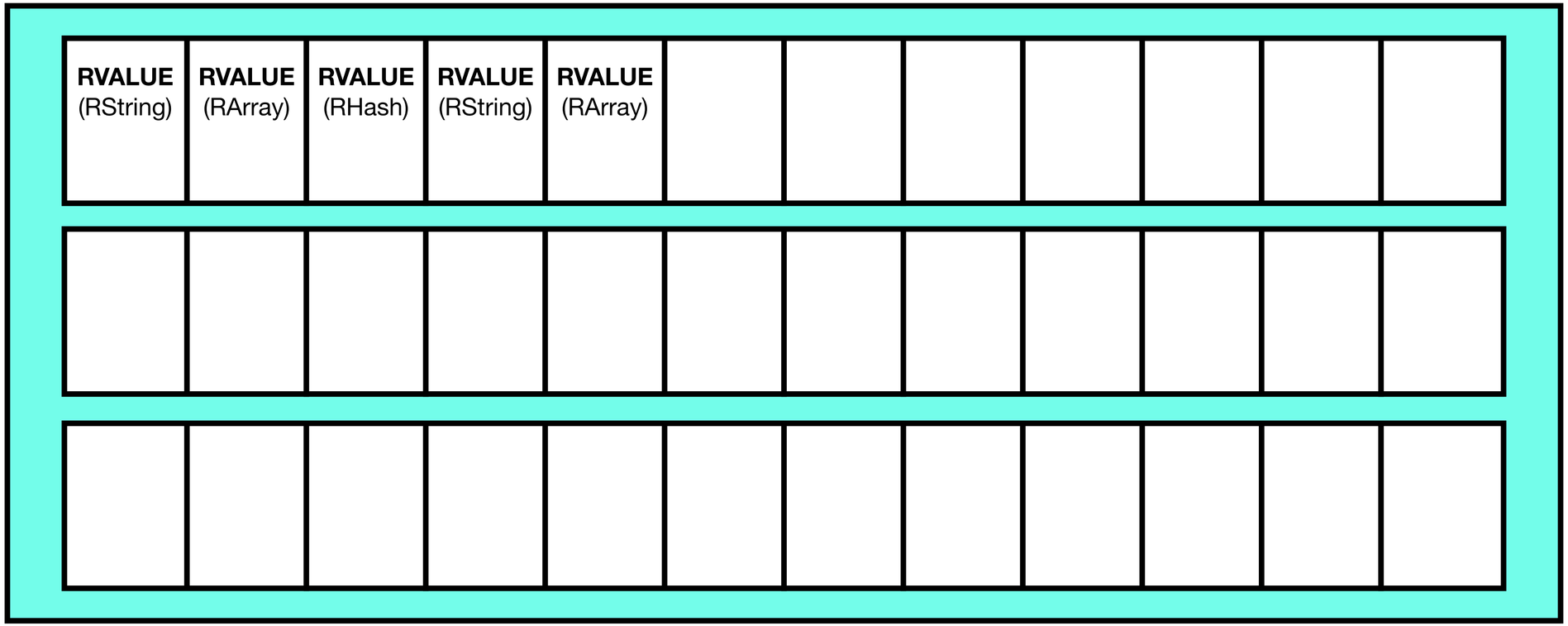
Background

GC Compaction (less briefly)

Background - GC Compaction

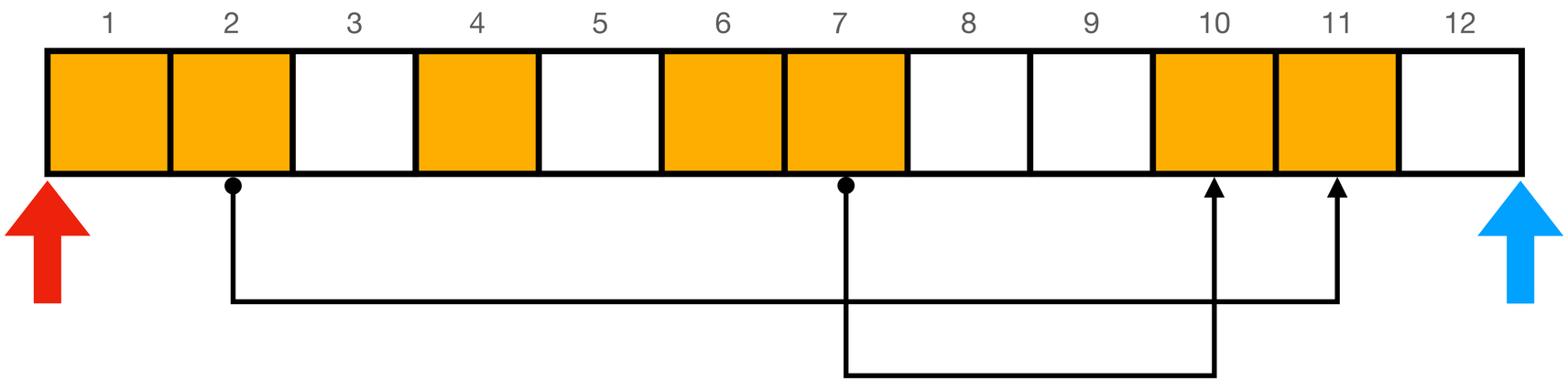


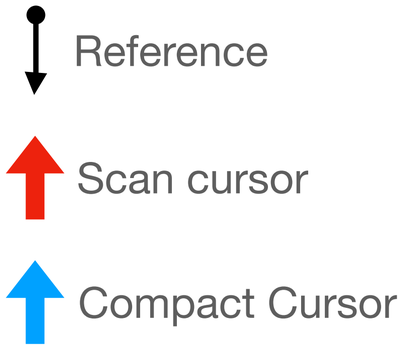
Background - GC Compaction



Background - The Two Finger Algorithm

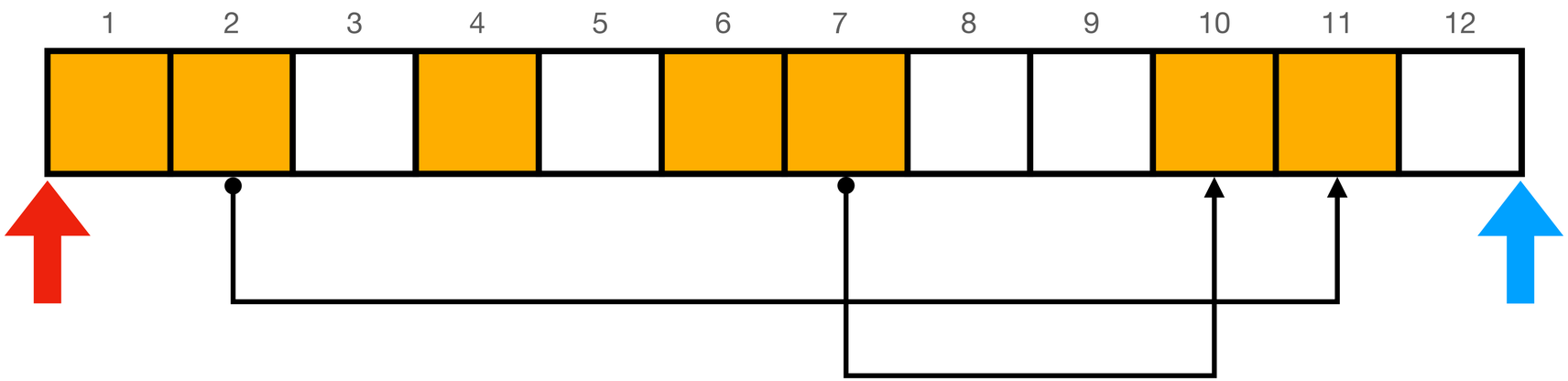
Phase 1: Object Movement

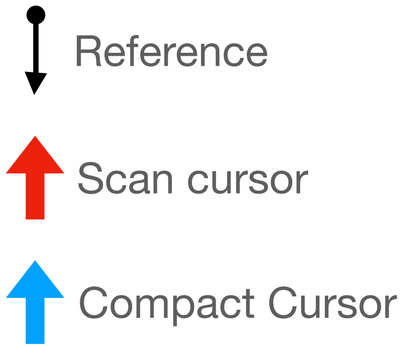




Background - The Two Finger Algorithm

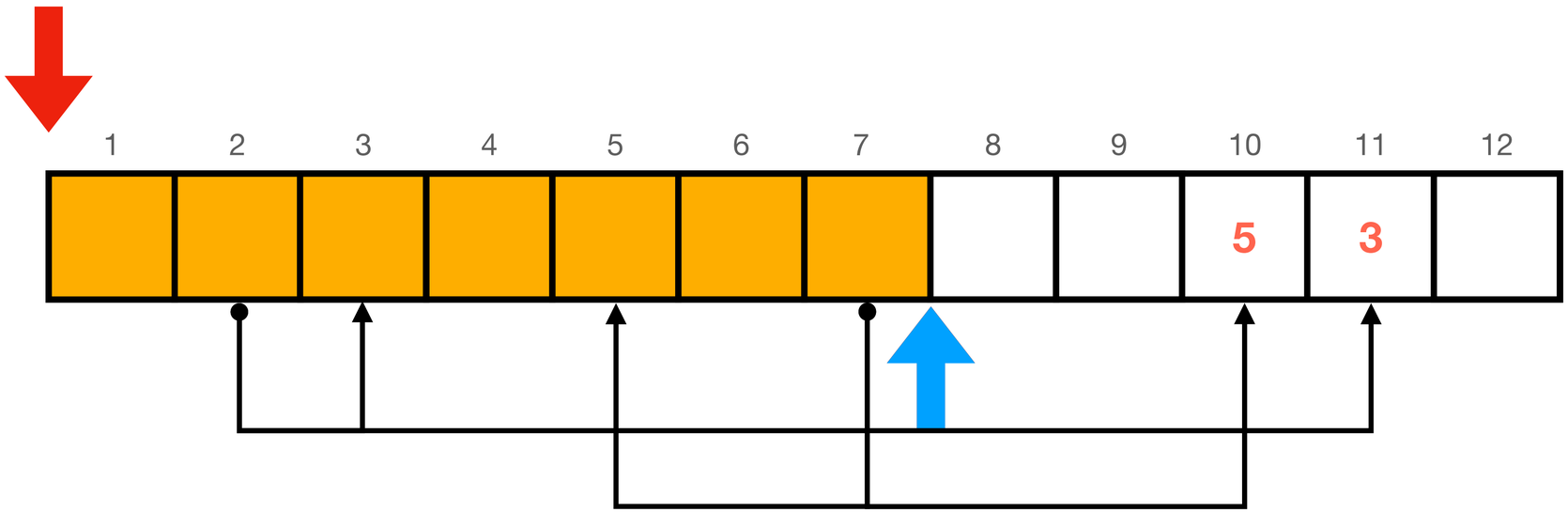
Phase 1: Object Movement





Background - The Two Finger Algorithm

Phase 2: Update References



Reference

Scan cursor

Compact Cursor

Background - GC Compaction

- Objects that cannot be moved are pinned
- `GC.compact` was introduced in Ruby 2.7
- Auto-compaction introduced in Ruby 3.0 but disabled by default
- Ruby 3.0 integrated sweep and compact

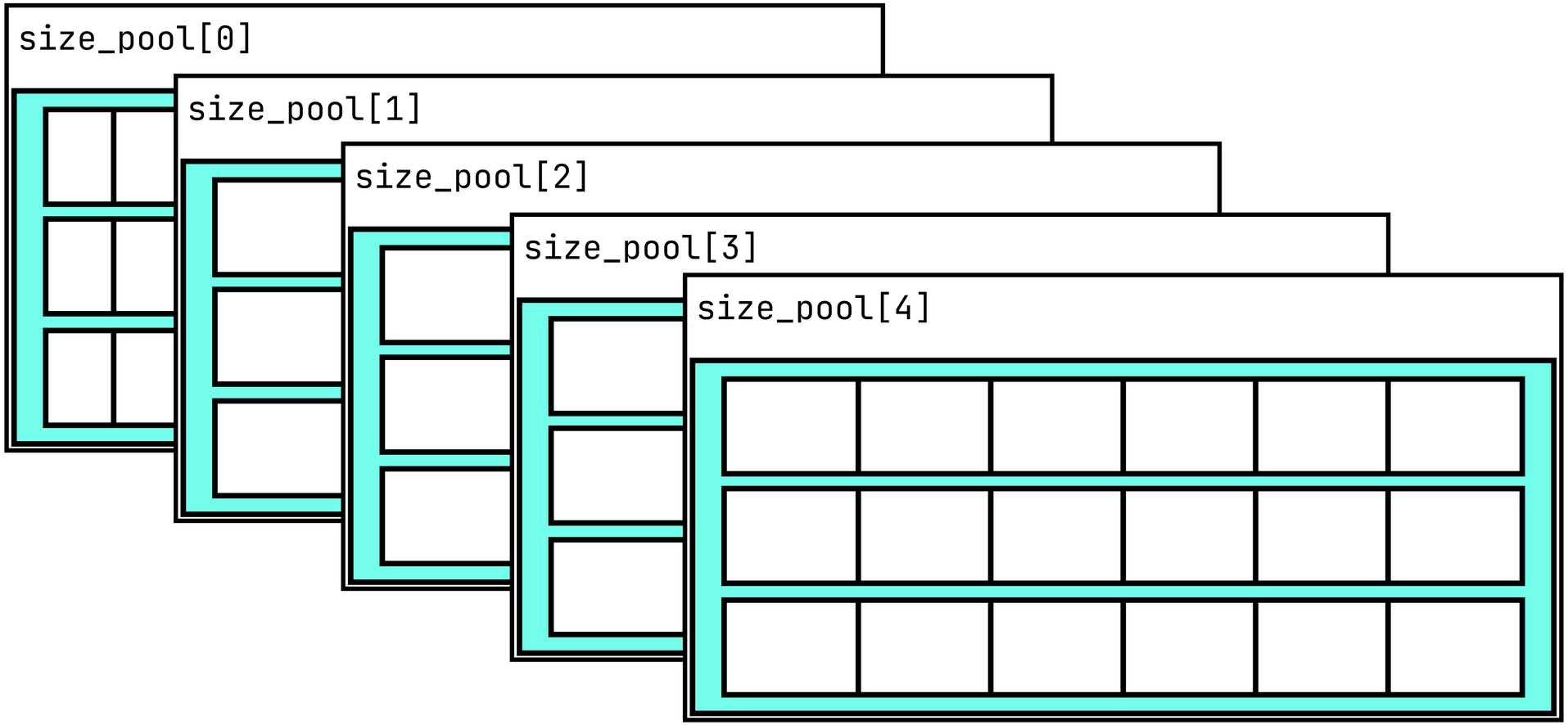
Background - Summary

- **Heap/Pages/RVALUES** - Ruby stores objects in fixed size slots in pages
- **Compacting GC** - Objects get moved during sweeping to improve fragmentation
- **Objects can have poor data locality** - related data is often far apart in memory
- **VWA is a project to improve data locality** - by restructuring Ruby's heap

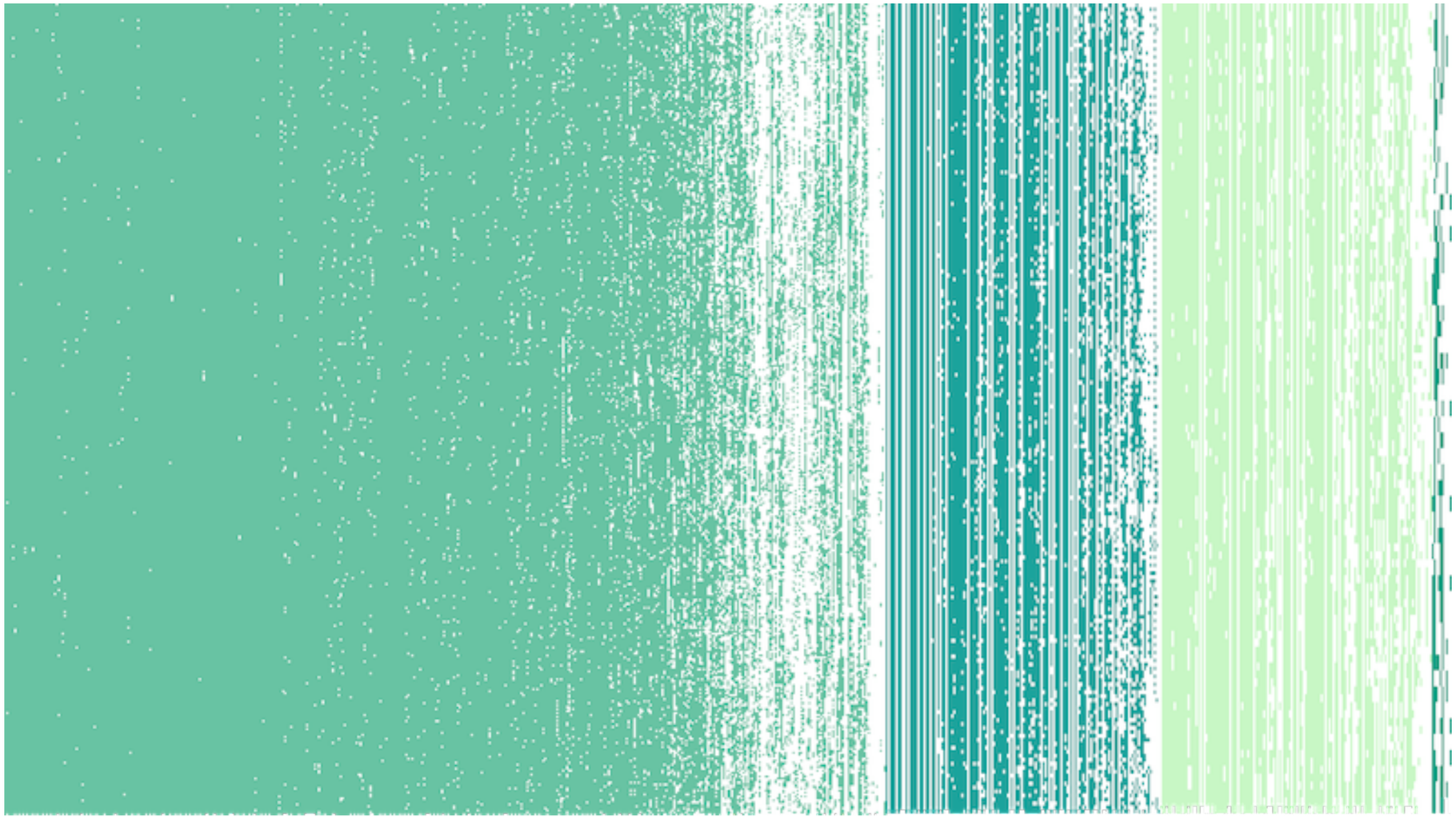
The Problem

Uh-Oh

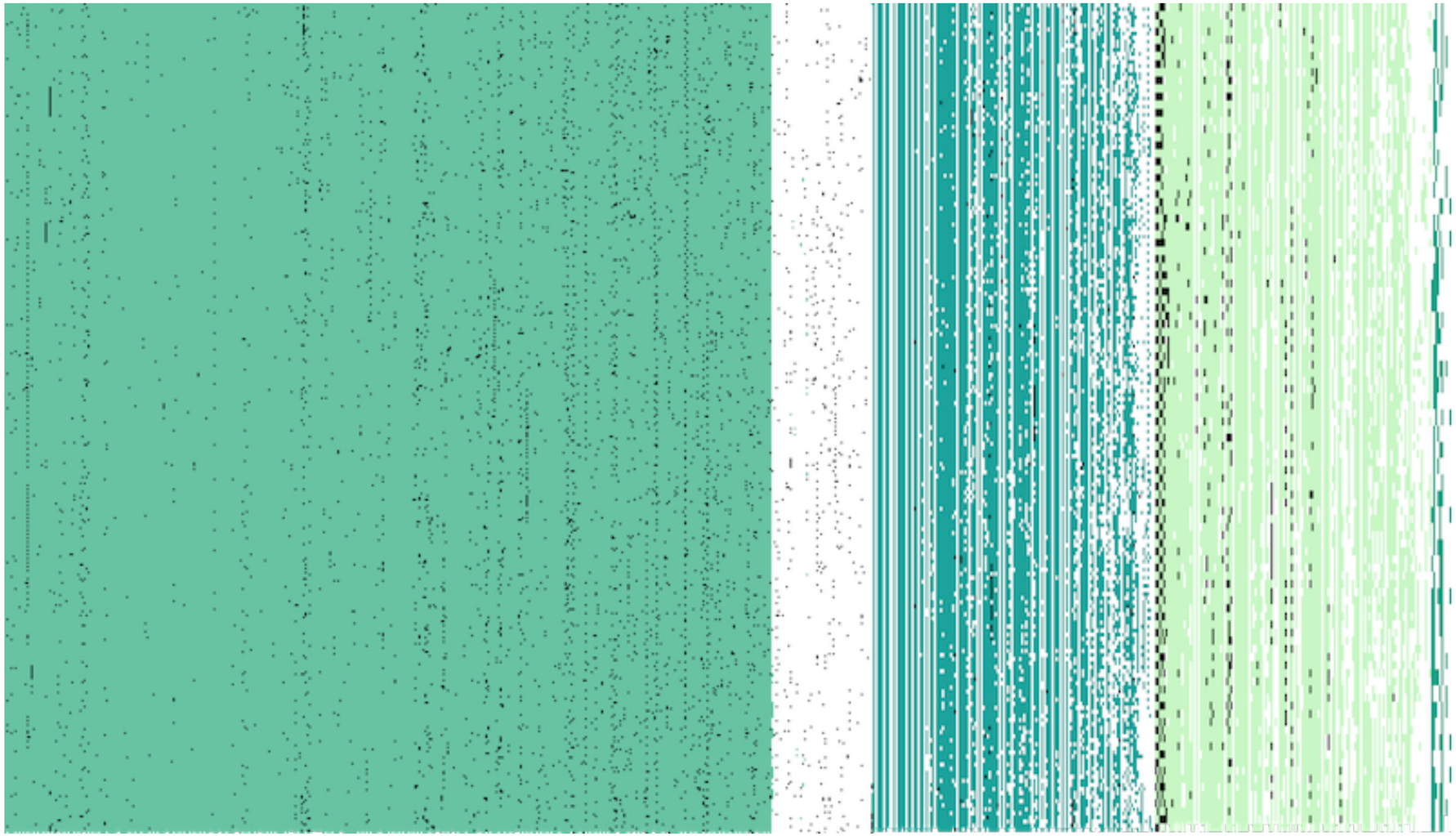
The Problem



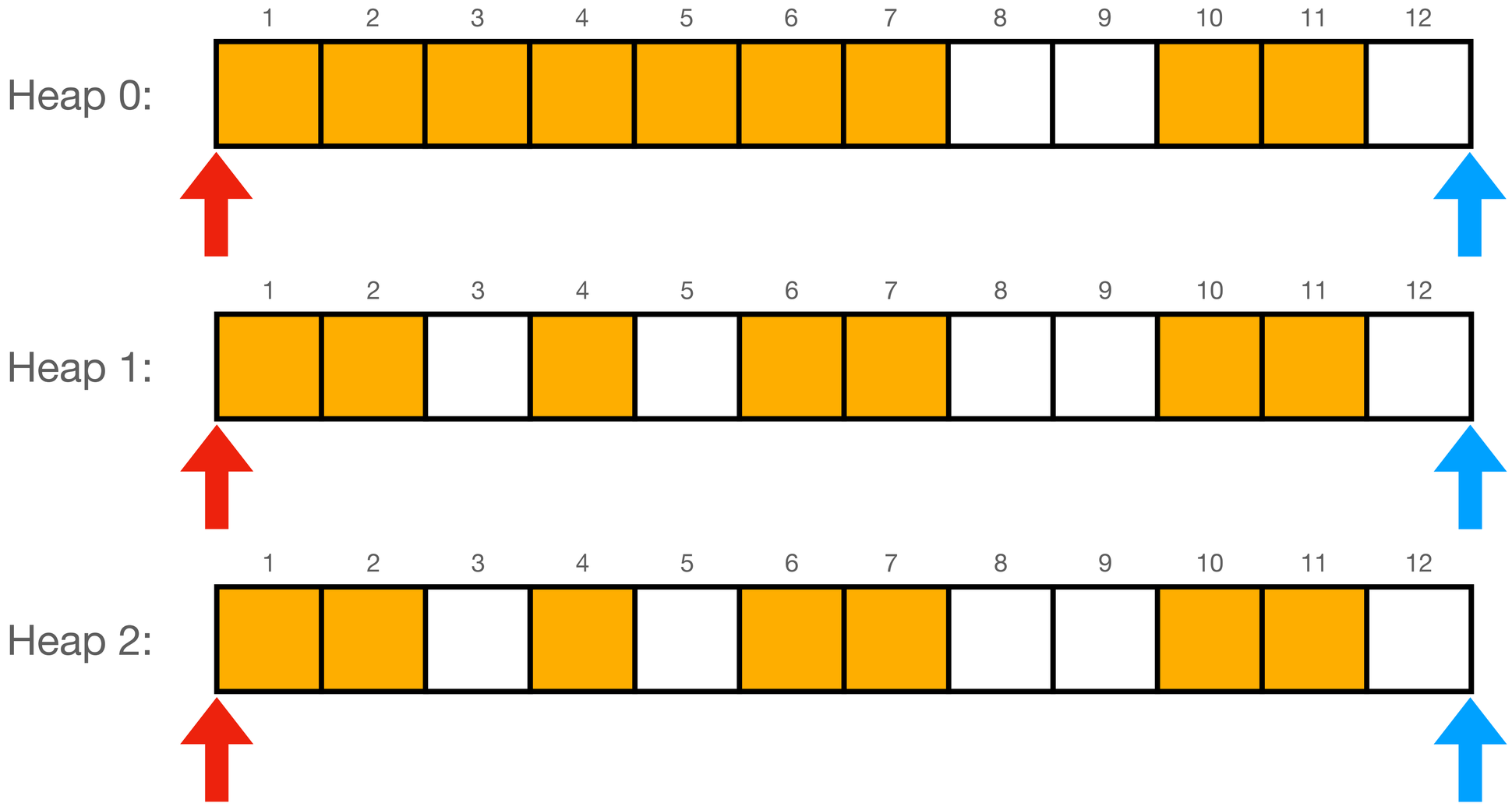
The Problem - Uncompacted Size Pools



The Problem - Compacted Size Pools



The Problem




```
objspace→flags.during_compacting = false;
```

The Problem - Attempt 1

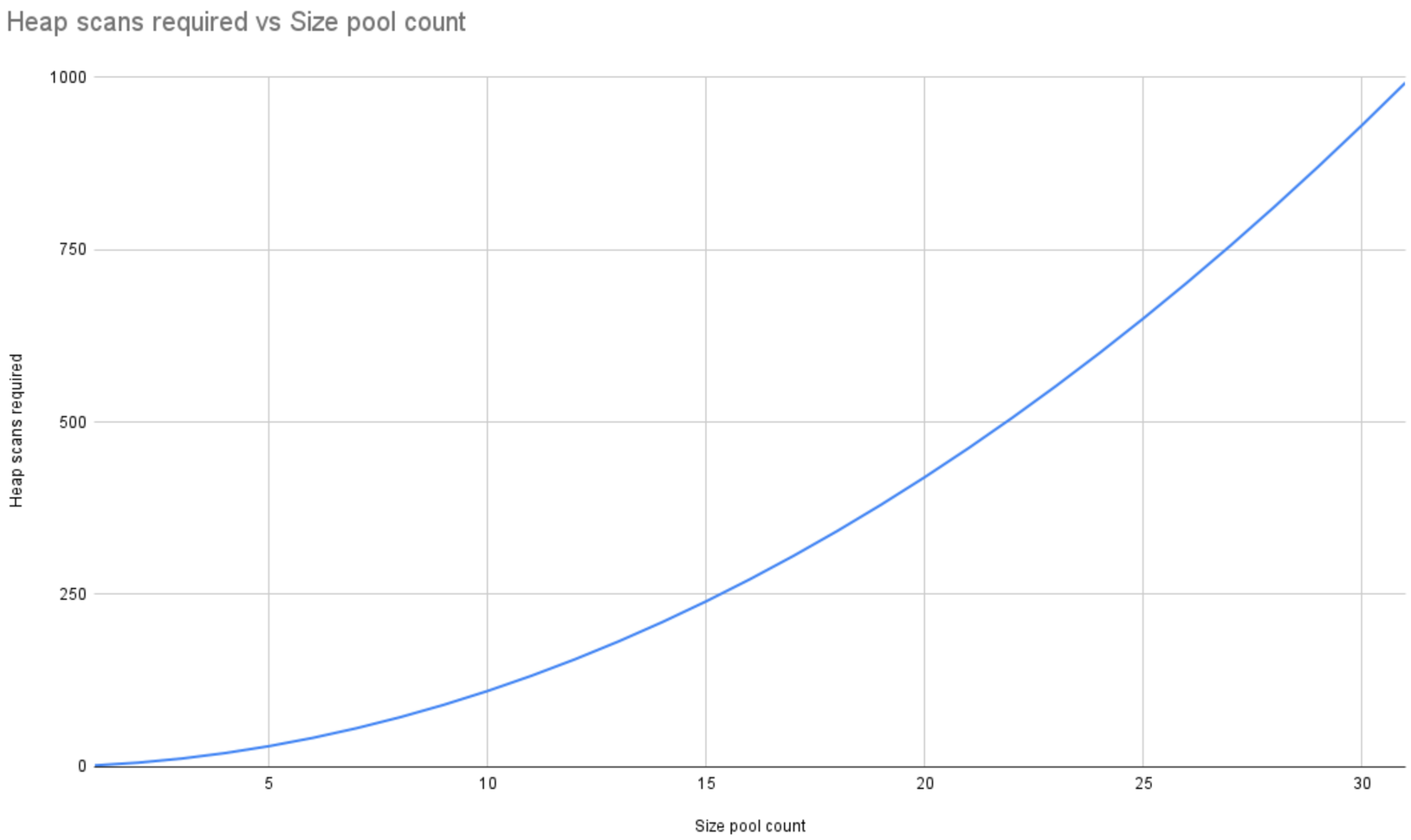
Compact each heap independently

✅ Simple

Scope all compaction context into the size pool. Each heap has it’s own compaction status flag.

The Problem - Attempt 1

Compact each heap independently



The Problem - Attempt 1

Compact each heap independently

✅ Simple

❌ Unacceptably slow due to excessive heap scans

```
Scope all compaction context into
the size pool. Each heap has it’s
own compaction status flag.
```

An Aside

Resizing Objects

✅ `hash = { a: true, b: false }`

✅ `string = “Hello, RubyKaigi”`

✅ `array = [1, 2, 3, a, b, c]`

An Aside

Resizing Objects

✅ `hash = { a: true, b: false }`

✅ `string = “Hello, RubyKaigi”`

✅ `array = [1, 2, 3, a, b, c]`

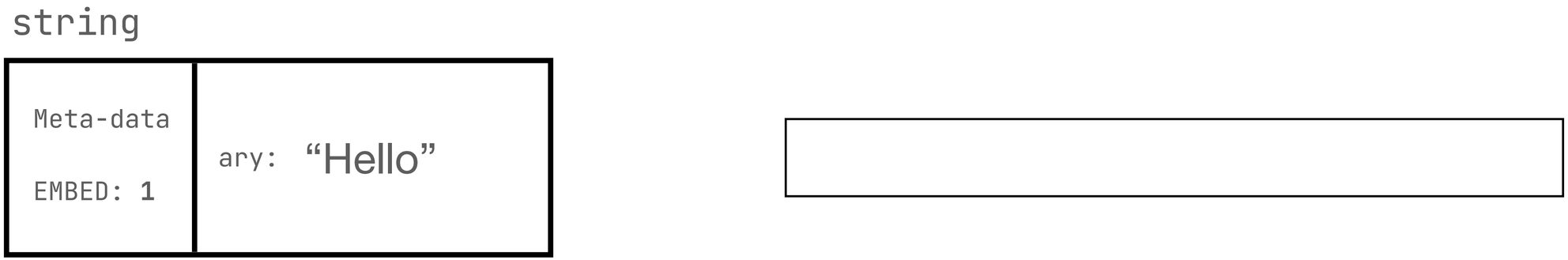
❌
```
array = []
array.push(1)
array.push(2)
```

❌ `string << “, pleased to meet you”`

An Aside

Resizing Objects

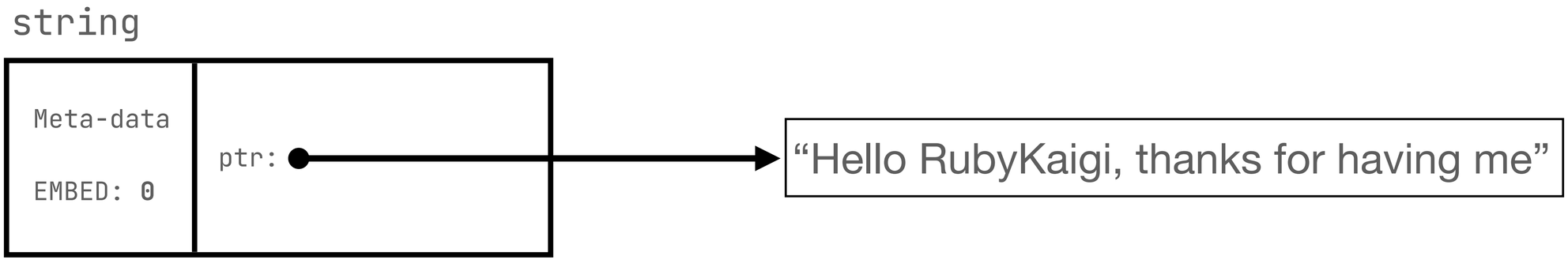
```
string = “Hello”
string << “ RubyKaigi, thanks for having me”
```



An Aside

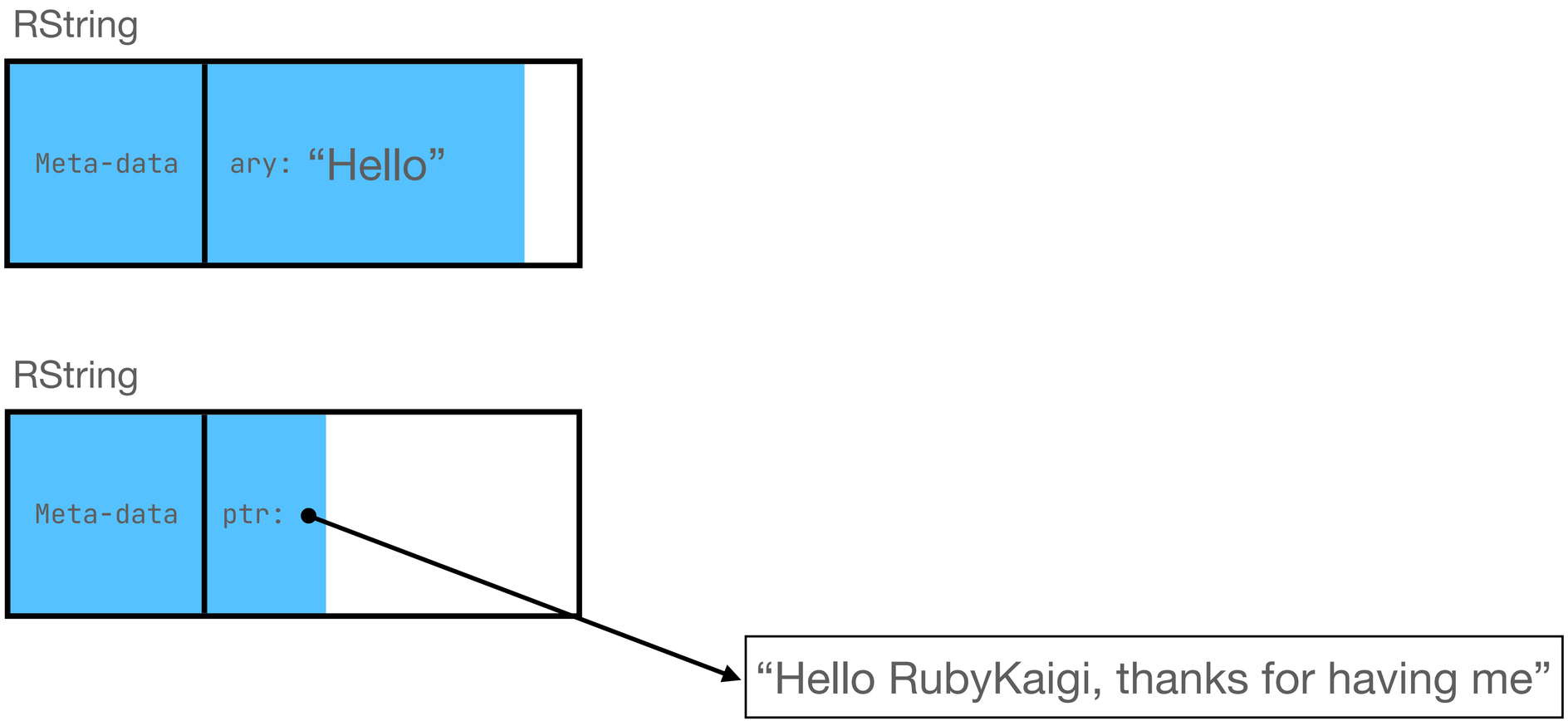
Resizing Objects

```
string = “Hello”
string << “ RubyKaigi, thanks for having me”
```



An Aside

Resizing Objects



The Problem - Attempt 1

Compact each heap independently

```
Scope all compaction context into
the size pool. Each heap has its
own compaction status flag.
```

✅ Simple

❌ Unacceptably slow due to excessive heap scans

❌ Potentially wasteful of memory

The Problem - Attempt 1

Compact each heap independently

```
Scope all compaction context into
the size pool. Each heap has its
own compaction status flag.
```

- ✅ Simple
- ❌ Unacceptably slow due to excessive heap scans
- ❌ Potentially wasteful of memory
- ❌ Doesn’t allow for object movement between size pools

The Problem - Attempt 2

Seperate Object Movement from Reference Updating

```
Perform object movement on each
heap in turn.

When object movement is complete
for all heaps, then update
references in a single step
```

The Problem - Attempt 2

Seperate Object Movement from Reference Updating

```
Perform object movement on each
heap in turn.

When object movement is complete
for all heaps, then update
references in a single step
```

✅ Faster due to fewer heap scans. References are only updated once

The Problem - Attempt 2

Seperate Object Movement from Reference Updating

```
Perform object movement on each
heap in turn.

When object movement is complete
for all heaps, then update
references in a single step
```

- ✅ Faster due to fewer heap scans. References are only updated once
- ✅ Allows some limited object movement between pools

The Problem - Attempt 2

Seperate Object Movement from Reference Updating

```
Perform object movement on each
heap in turn.

When object movement is complete
for all heaps, then update
references in a single step
```

✅ Faster due to fewer heap scans. References are only updated once

✅ Allows some limited object movement between pools

❌ More complex algorithm

The Problem - Attempt 2

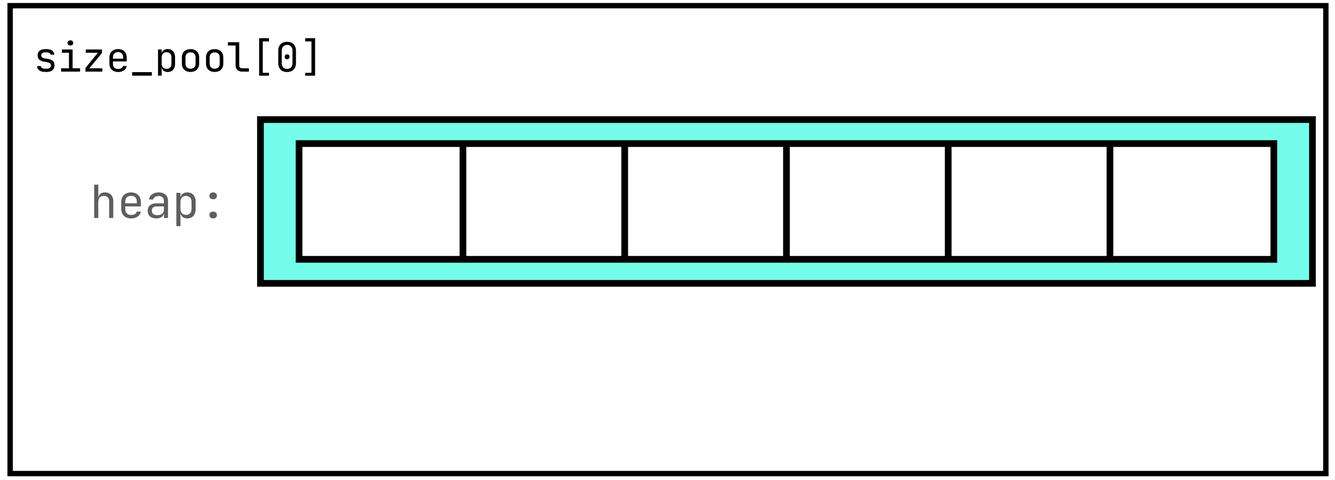
Seperate Object Movement from Reference Updating

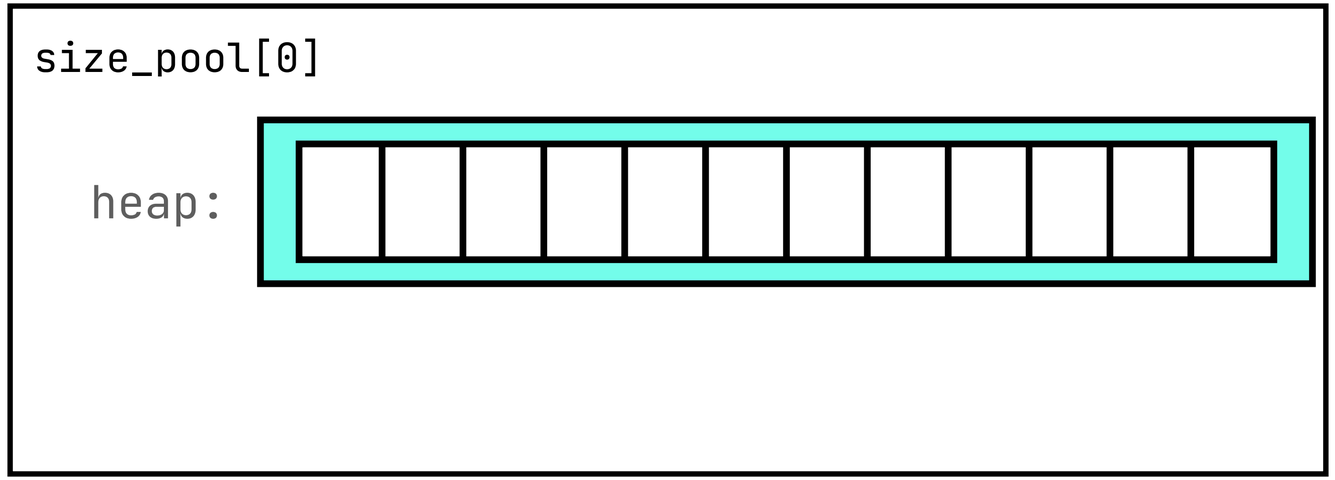
```
Perform object movement on each
heap in turn.

When object movement is complete
for all heaps, then update
references in a single step
```

- ✅ Faster due to fewer heap scans. References are only updated once
- ✅ Allows some limited object movement between pools
- ❌ More complex algorithm
- ❌ Object Movement only one way

The Problem - Attempt 3

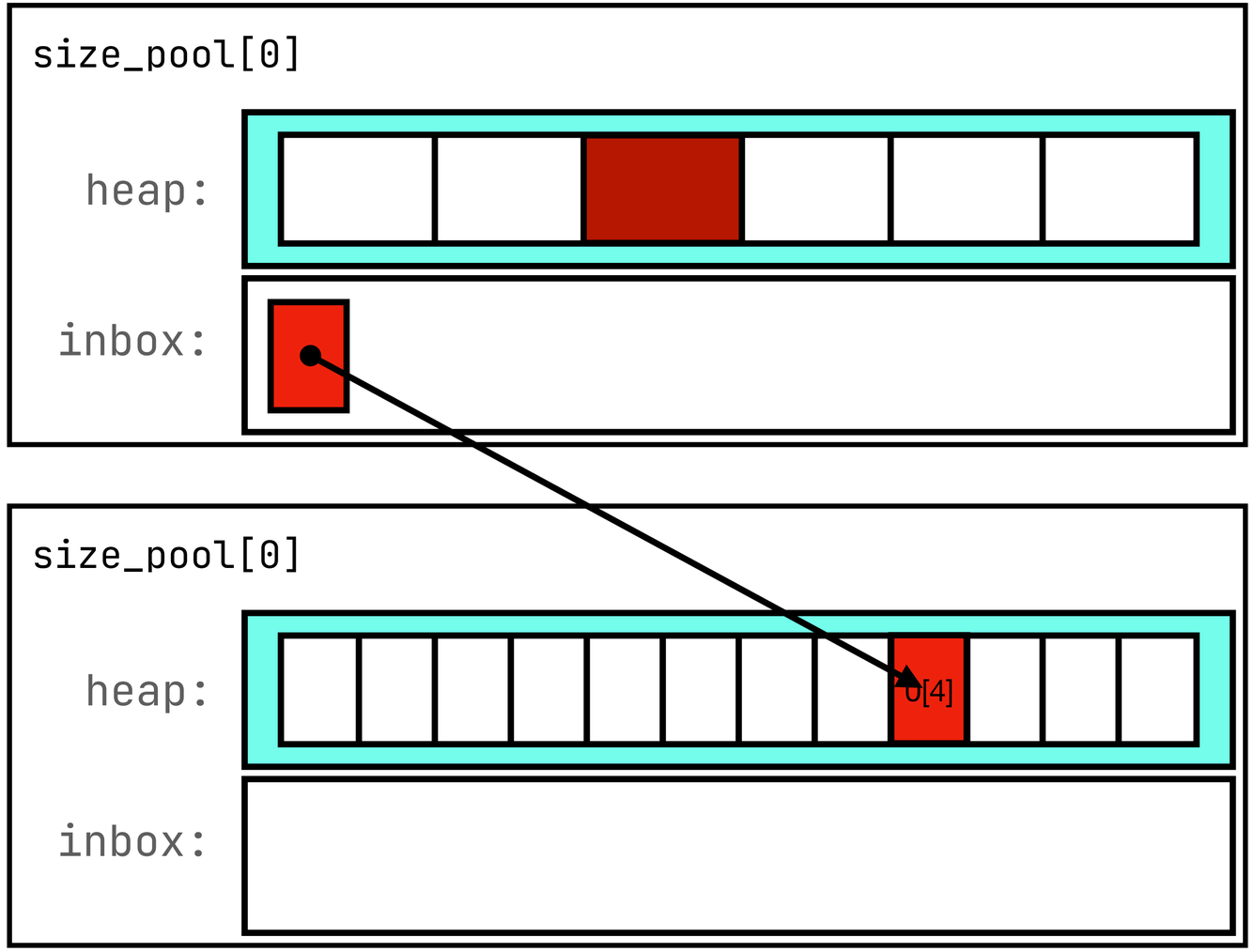
Size Pool Inboxes





The Problem - Attempt 3

Size Pool Inboxes



The Problem - Attempt 3

Size Pool Inboxes

Assign each size pool it’s own seperate data structure; an “inbox”.

When an object is mutated, calculate the best size pool for it and push a pointer to it into the inbox for that pool.

When compacting the pool, first empty the inbox before moving from the compact cursor

✅ As fast as attempt 2

The Problem - Attempt 3

Size Pool Inboxes

```
Assign each size pool it’s own
seperate data structure; an
“inbox”.

When an object is mutated,
calculate the best size pool for
it and push a pointer to it into
the inbox for that pool.

When compacting the pool, first
empty the inbox before moving
from the compact cursor
```

✅ As fast as attempt 2

✅ Allows complete bi-directional object movement

The Problem - Attempt 3

Size Pool Inboxes

```
Assign each size pool it’s own
seperate data structure; an
“inbox”.

When an object is mutated,
calculate the best size pool for
it and push a pointer to it into
the inbox for that pool.

When compacting the pool, first
empty the inbox before moving
from the compact cursor
```

✅ As fast as attempt 2

✅ Allows complete bi-directional object movement

❌ Requires extra managed data structures

The Problem - Attempt 3

Size Pool Inboxes

```
Assign each size pool it’s own
seperate data structure; an
“inbox”.

When an object is mutated,
calculate the best size pool for
it and push a pointer to it into
the inbox for that pool.

When compacting the pool, first
empty the inbox before moving
from the compact cursor
```

- ✅ As fast as attempt 2
- ✅ Allows complete bi-directional object movement
- ❌ Requires extra managed data structures
- ❌ Leaks GC abstractions outside the GC

What now?

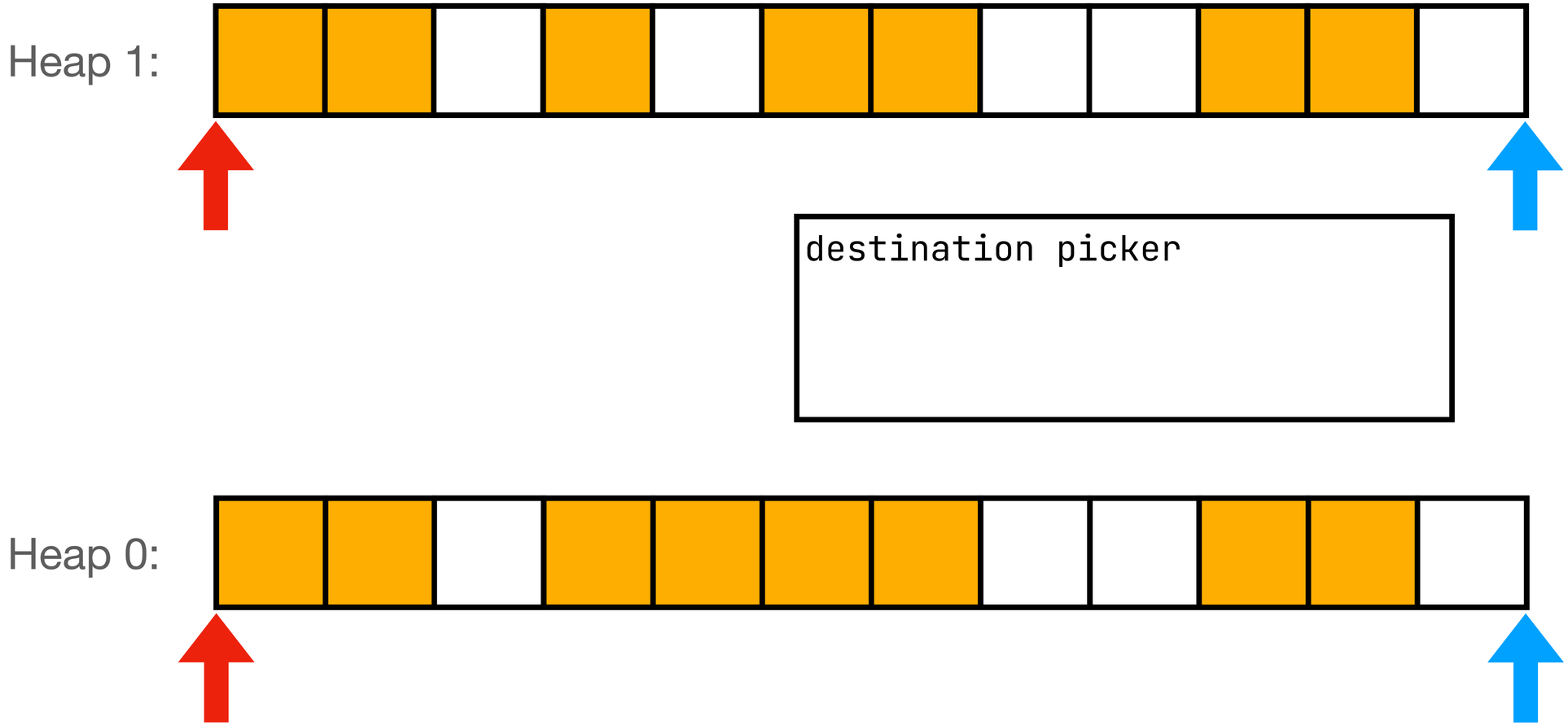
What now?

"What is the best place to put this object?"

"Where is the best place to put this object?"

The Problem - Attempt 4

Reversing the cursors



The Problem - Attempt 4

Seperate Object Movement from Reference Updating

```
Move the compact cursor down the
heap until a live object is
reached.

Determine the best size pool
based on the object size.

Start sweeping the destination
pool’s heap until a free slot is
found

Move the live object into the
free slot
```

✅ Almost as fast as the existing compaction algorithm

The Problem - Attempt 4

Seperate Object Movement from Reference Updating

```
Move the compact cursor down the
heap until a live object is
reached.

Determine the best size pool
based on the object size.

Start sweeping the destination
pool’s heap until a free slot is
found

Move the live object into the
free slot
```

- ✅ Almost as fast as the existing compaction algorithm
- ✅ negligible amount of extra complexity

The Problem - Attempt 4

Seperate Object Movement from Reference Updating

```
Move the compact cursor down the
heap until a live object is
reached.

Determine the best size pool
based on the object size.

Start sweeping the destination
pool’s heap until a free slot is
found

Move the live object into the
free slot
```

- ✅ Almost as fast as the existing compaction algorithm
- ✅ negligible amount of extra complexity
- ✅ No extra data structures to manage

The Problem - Attempt 4

Seperate Object Movement from Reference Updating

```
Move the compact cursor down the
heap until a live object is
reached.

Determine the best size pool
based on the object size.

Start sweeping the destination
pool’s heap until a free slot is
found

Move the live object into the
free slot
```

- ✅ Almost as fast as the existing compaction algorithm
- ✅ negligible amount of extra complexity
- ✅ No extra data structures to manage
- ✅ Natural bi-directional movement between pools

The Problem - Attempt 4

Home My page Projects Help

Logged in as eightbitraptor My account Sign out

Ruby »
Ruby master

Search: Ruby master

Overview Activity Roadmap Issues New issue Wiki Repository

Feature #18619 CLOSED

Edit Unwatch •••

Reverse the order of GC Compaction cursor movement

« Previous | 4 of 7 | Next »

Added by eightbitraptor (Matthew Valentine-House) 5 months ago. Updated 5 months ago.

Status: Closed
Priority: Normal
Assignee: -
Target version: -
[ruby-core:107818]

Description

Quote

Reverse the order of GC Compaction cursor movement

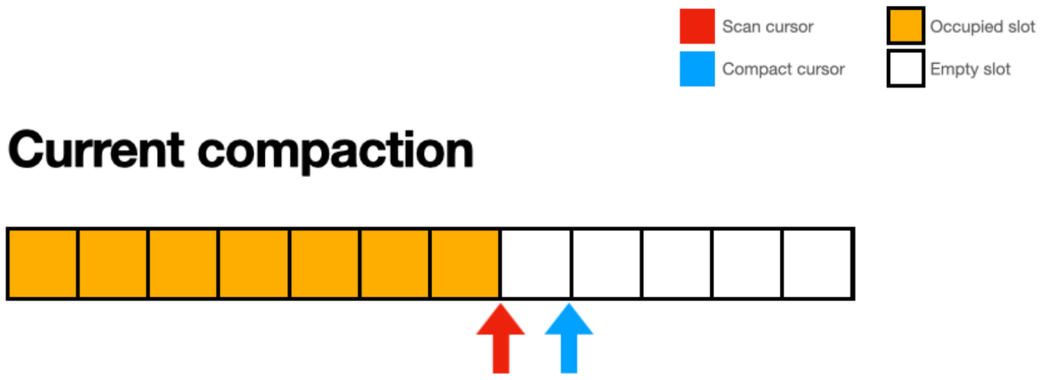
Github PR: https://github.com/ruby/ruby/pull/5637

Summary

The current compaction algorithm works by walking up the heap until it finds a slot to fill and then asking "what object can I use to fill this slot"

This PR reverses the cursor movement to walk down the heap until it gets to a moveable live object and then asking "where is the best place to put this object"

This animation shows a very simplified view of what we mean by "reversing the cursor movement"



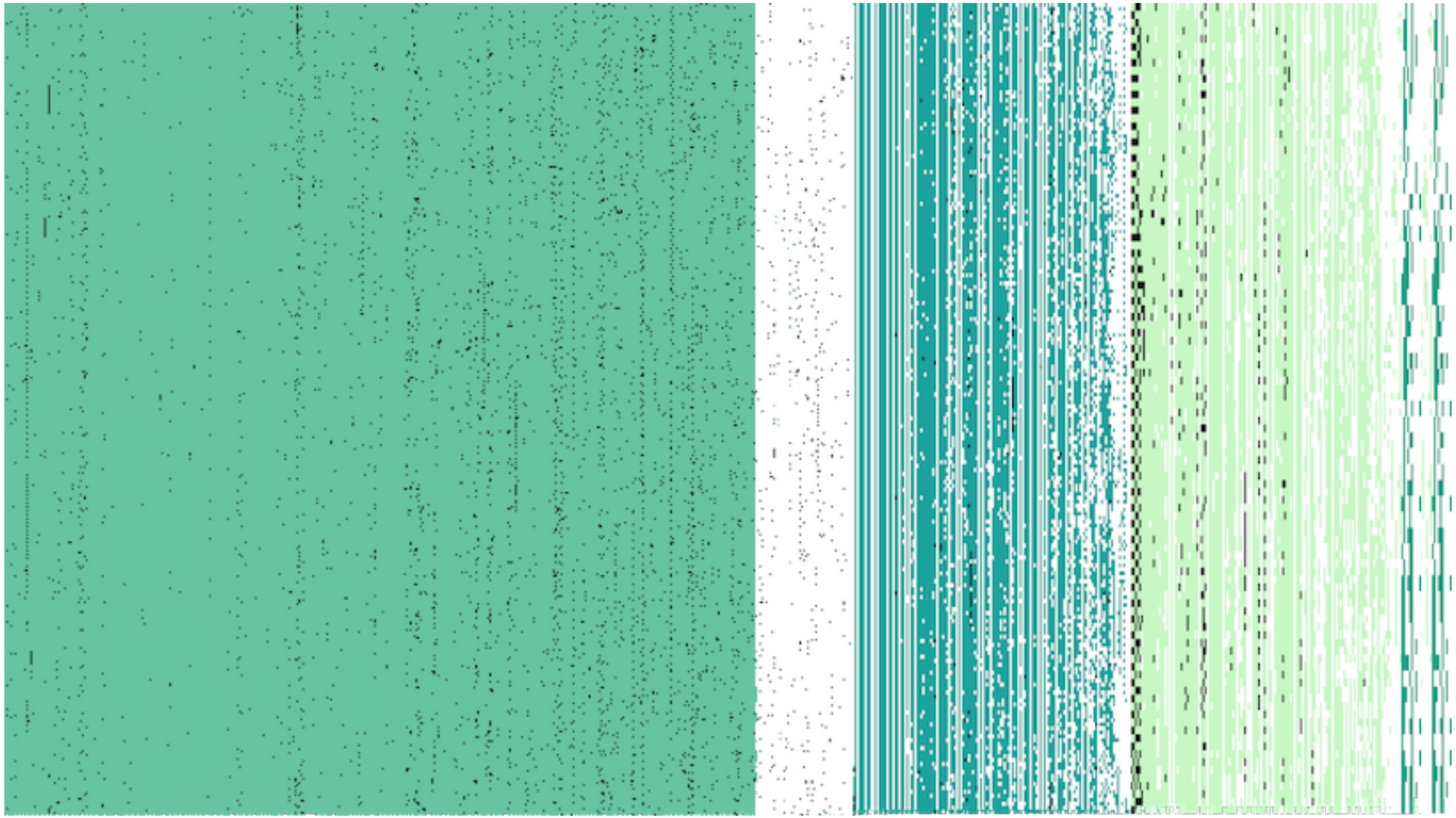
My custom queries

open bugs

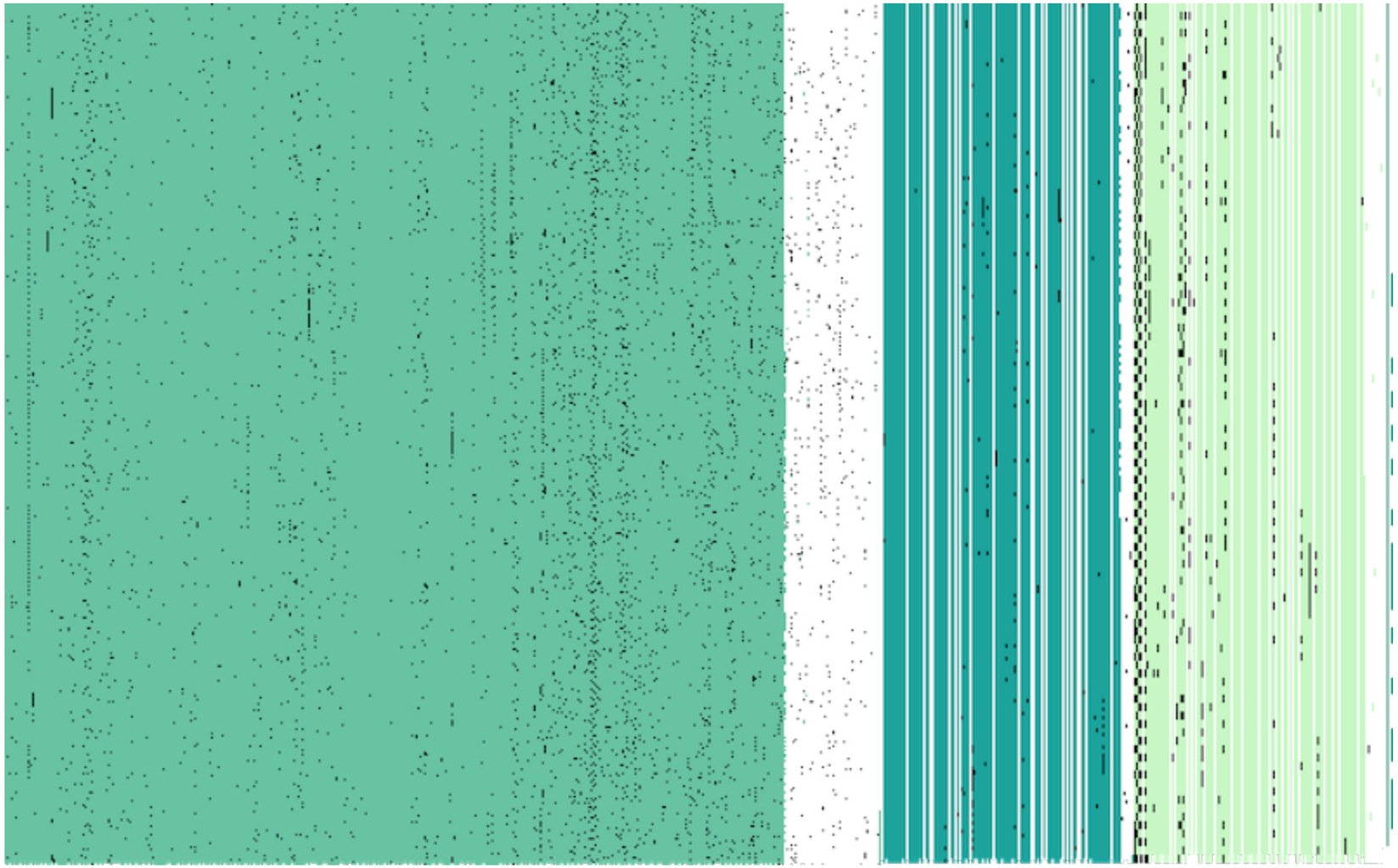
Custom queries

Backport 2.2
Backport 2.3
Backport 2.4
Backport 2.5
Backport 2.6
Backport 2.7
Backport 3.0
Backport 3.1
bugs: unassigned
DevelopersMeeting
matz

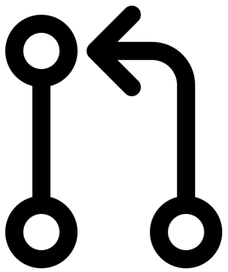
The Problem - Attempt 4



The Problem - Attempt 4



The Problem - Attempt 4



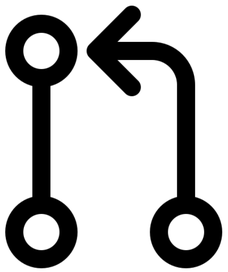
What’s next?

Object Movement & Variable Width Allocation Status

String Movement

```
static rb_size_pool_t *
gc_compact_destination_pool(rb_objspace_t *objspace, rb_size_pool_t *src_pool, VALUE src)
{
    size_t obj_size;

    switch (BUILTIN_TYPE(src)) {
        case T_STRING:
            obj_size = rb_str_size_as_embedded(src);
            break;

        default:
            return src_pool;
    }

    if (rb_gc_size_allocatable_p(obj_size)){
        return &size_pools[size_pool_idx_for_size(obj_size)];
    }
    else {
        return &size_pools[0];
    }
}
```

Moving Strings



VWA Update

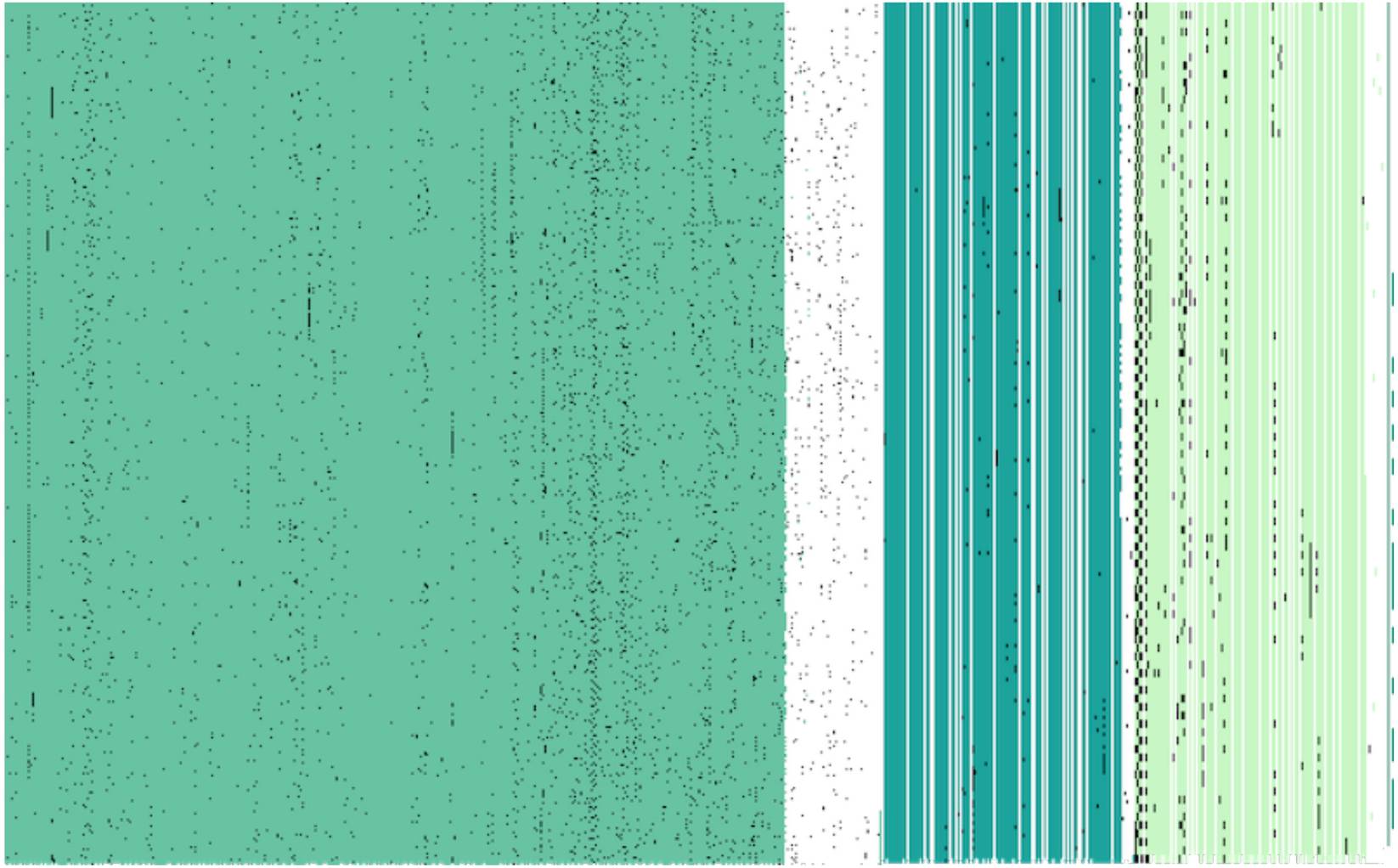
VWA is currently implemented for:

- **Classes**
- **Strings**
- **Arrays**
- **Objects**

Heapviz

A Ruby Gem for Heap Visualisation

Heapviz



Heapviz

Visualizing your Ruby Heap

- Aaron Patterson, 2017, tenderlovemaking.com

```
> gem install heapviz
```

https://github.com/eightbitraptor/heapviz

Thanks RubyKaigi