Project M-Graph: Slaying the Silent Performance Killer

How a shift to lazy initialisation cut HTML processing time by 31%

v1.4.21 | Optimisation Period: 27–29 November 2025

A 31% Reduction in Total Execution Time

Total Processing Time (Size=30 Doc)

-31%

Tiempos Text Regular Execution time reduced from 441.79ms to 304.21ms.

Single Element Processing (Size=1 Doc)

-70%

Tiempos Text Regular Execution time reduced from 121.5ms to 36.3ms.

Index Object Construction

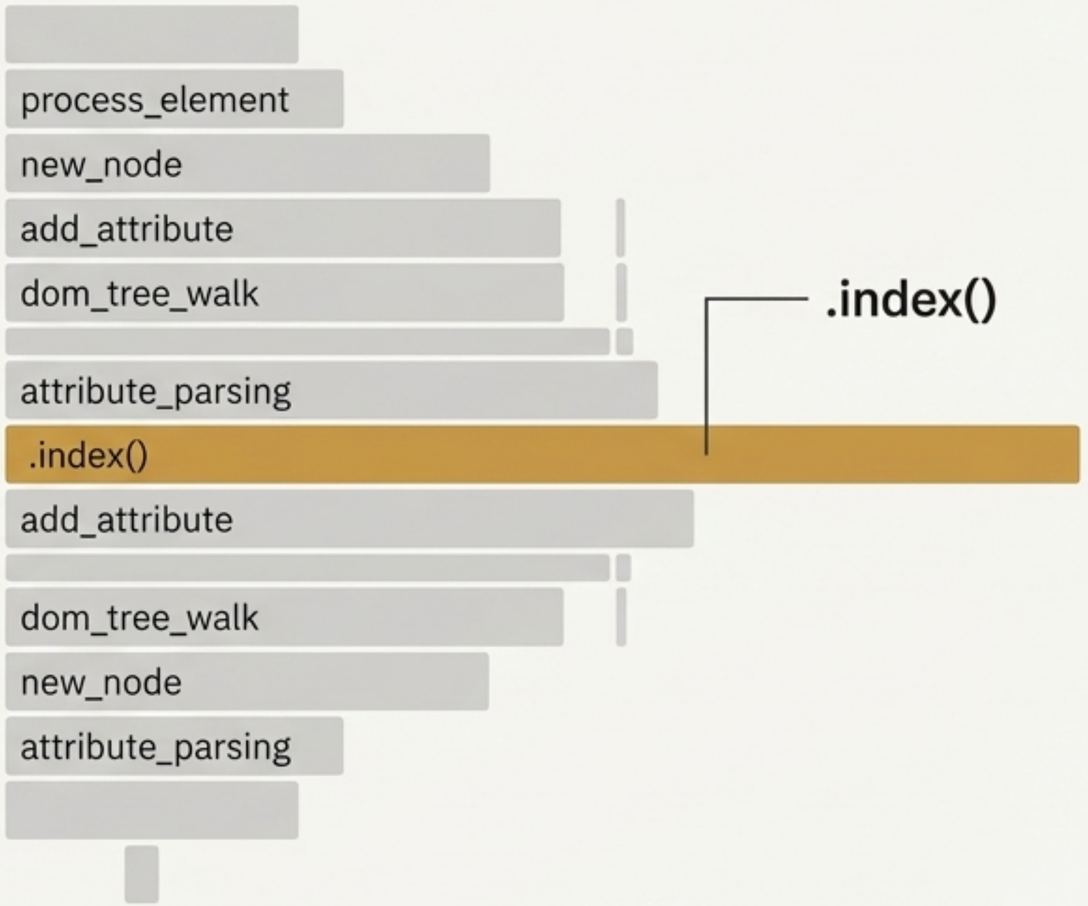
-97%

Tiempos Text Regular Construction time reduced from 1.9ms to just 50µs.

The Investigation Began with a Clear Bottleneck: Indexing

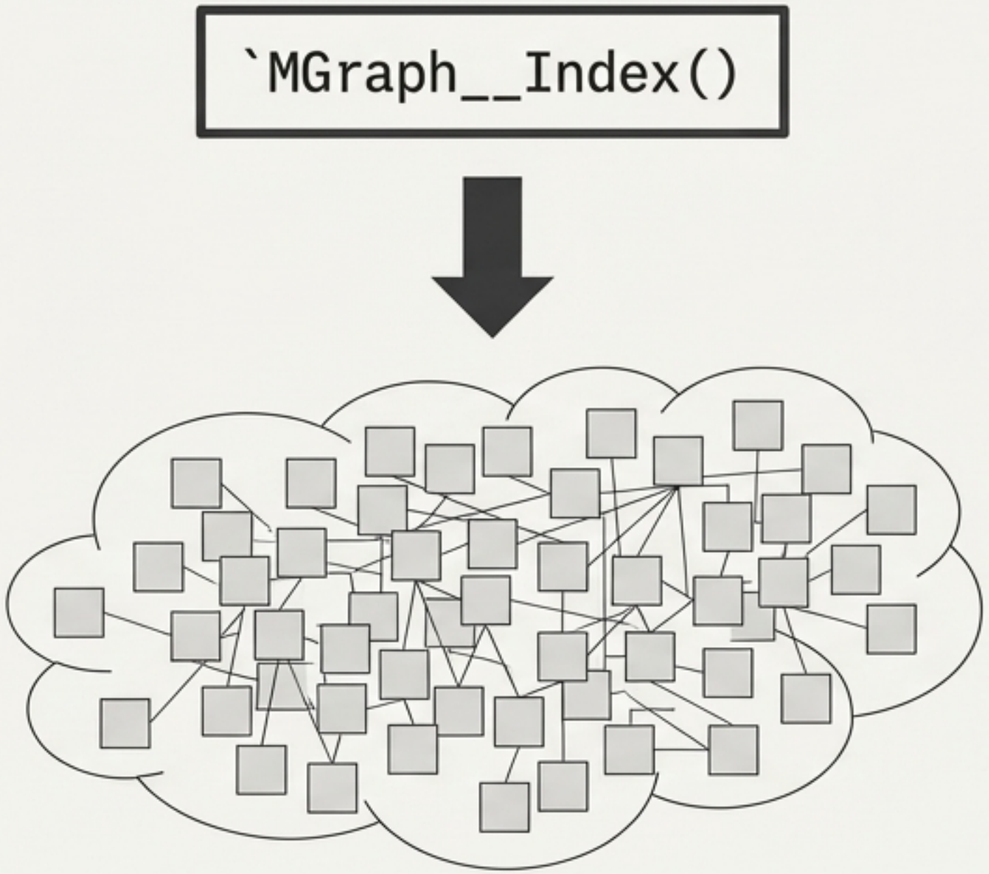
Profiling revealed that calls to `.index()` were disproportionately expensive, consuming a significant portion of the total processing time.

For a simple HTML document, **.index()** calls alone were responsible for **31%** of total processing time (14.2ms of 45.3ms).



The Root Cause: Wasteful, Eager Object Initialisation

The `Type_Safe` base class initialised all nested attributes recursively on construction. For a single `MGraph__Index` object, this meant creating a cascade of **over 100 nested objects**, regardless of whether they were needed.



> “Most of these 100+ objects were immediately discarded as references were rewired.”

The True Cost of Construction Was Staggering

73:1

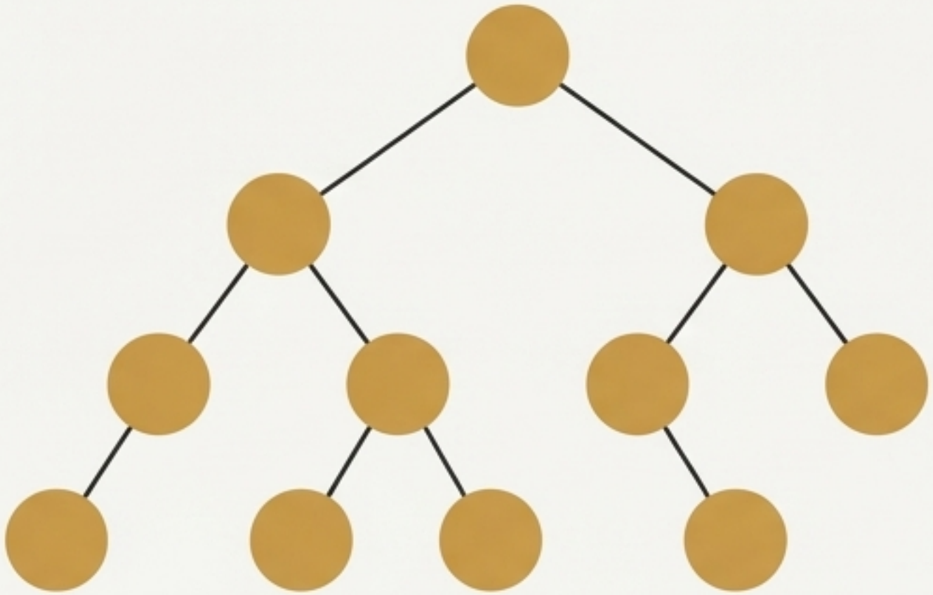
The ratio of object construction time to actual, useful work for every index operation.

Additional Detail: After optimisation, this ratio dropped to approximately **2:1**.

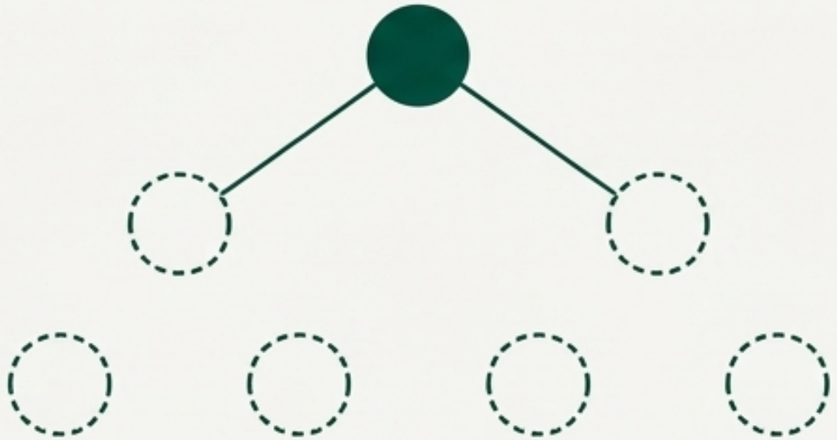
The Solution: Build Only What You Need, When You Need It

We introduced `Type_Safe__On_Demand`, a new base class that defers attribute initialisation until the moment of first access. This eliminates the upfront construction overhead entirely.

BEFORE: `Type_Safe`: Eager Initialisation



AFTER: `Type_Safe__On_Demand`: Lazy Initialisation



A Targeted Upgrade Across the Indexing Layer

The shift to lazy initialisation was applied to all classes involved in the M-Graph indexing mechanism.

Index Layer

`MGraph__Index`

`Schema__MGraph__Index__Data`

`Schema__MGraph__Index__Data__Edges`

`Schema__MGraph__Index__Data__Labels`

`Schema__MGraph__Index__Data__Paths`

`Schema__MGraph__Index__Data__Types`

Supporting Indexes

`MGraph__Index__Edges`

`MGraph__Index__Labels`

`MGraph__Index__Paths`

`MGraph__Index__Types`

`MGraph__Index__Values`

Head-to-Head: A 31% Speedup on the Same Workload

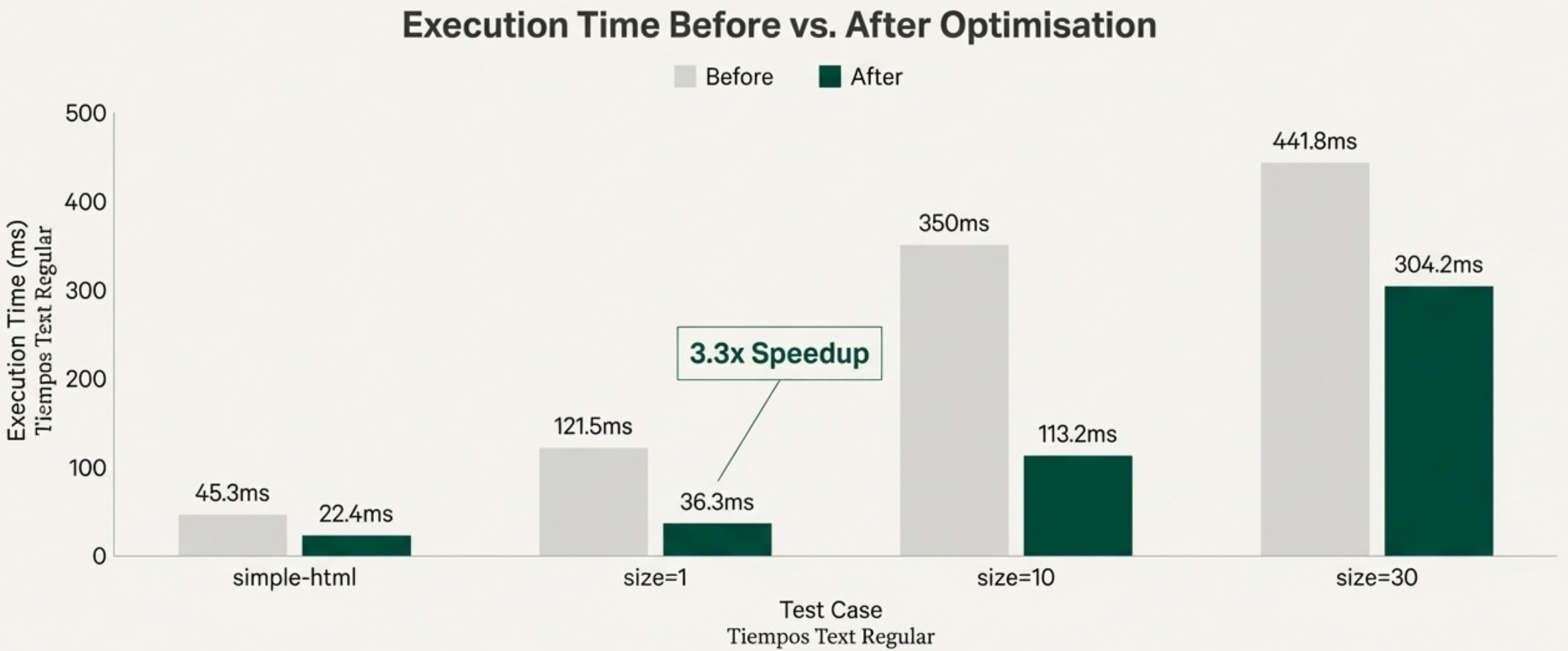
Comparison for a "Size=30" document, ensuring an apples-to-apples test.
Both runs executed exactly **4,322 calls**.

Frame	Before (27 Nov)	After (29 Nov)	Improvement
process_body_children	904.2ms	636.6ms	-30%
_process_body__element	873.3ms	614.3ms	-30%
execute_pipeline	**441.8ms**	**304.2ms**	**-31%**
add_attribute	252.8ms	177.5ms	-30%
new_node (mgraph_edit)	148.2ms	100.8ms	-32%
mgraph_node	46.5ms	29.9ms	-36%

-31%

Total Time

The Performance Gains Hold Up at Every Scale



Microseconds Matter: Transforming Core Operations

`MGraph__Index()` Construction

-97%

From 1.9ms down to 50µs

`add_attribute` call

-67%

From 4.2ms down to 1.4ms

First `.index()` call

-76%

From 2.5ms down to 0.6ms

`new_node (model_mgraph)`

-65%

From 0.31ms down to 0.11ms

Key Architectural Insights From This Optimisation

1. Construction is Not Free

The hidden cost of object creation, even when implicit, can easily dominate total processing time. Our initial 73:1 construction-to-work ratio is a stark reminder.

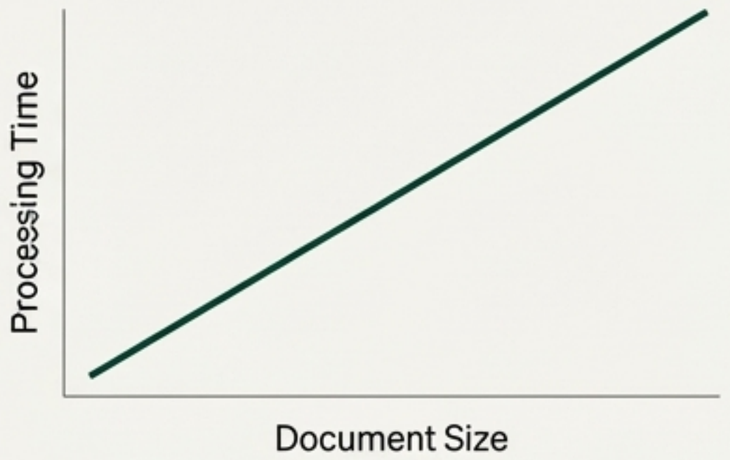
2. Fixed Costs Now Amortise Correctly

The setup cost is now truly fixed, ensuring predictable performance. The `document.setup` phase is constant regardless of document size.

Size	`document.setup`
1	54.4ms
10	53.5ms
30	52.7ms

3. Linear Scaling is Now a Reality

With the construction bottleneck removed, processing time for `_process_body__element` now scales linearly with document size, matching its O(n) design.



Our Next Performance Frontiers

This optimisation has cleared the way for us to tackle the next layer of performance bottlenecks.

- **Flatten Delegation Chains:** The `add_attribute` call chain is 6 levels deep. Flattening it could save significant function call overhead.
- **Batch Edge Creation:** Each attribute currently creates 3 separate edges. A batch API could reduce this overhead.
- **Reduce Index Wiring Overhead:** Further lazy wiring of `_sync_index_data()` can eliminate more unnecessary object creation.
- **Optimise Node Schema:** Simplify or cache complex conditional logic within the `new_node(model_mgraph)` method.

A 31% Improvement with Zero API Changes

A complex HTML document that took **442ms** to process now completes in **304ms**.

> *"This success demonstrates the immense value of systematic profiling and micro-benchmarking."*

Appendix: Profiling Methodology

Tools Used

- Custom @timestamp decorator for method-level timing
- Speedscope for flame graph visualization
- Profile Analyzer UI for comparison views
- Perf utility for micro-benchmarks

Test Cases

- **simple-html**:

```
<html><body><div class="main"
id="content">Hello World</div>
</body></html>
```

- **with_size__N**: Generated HTML containing N repeated div elements.