shopify

Shopify's breadth-first bet: rethinking GraphQL execution

GraphQL Conf • May 19, 2026

Speaker

Greg MacWilliam

Staff Software Engineer,

Shopify

Agenda

01 GRAPHQL @ SHOPIFY

02 THEORY

03 MECHANICS

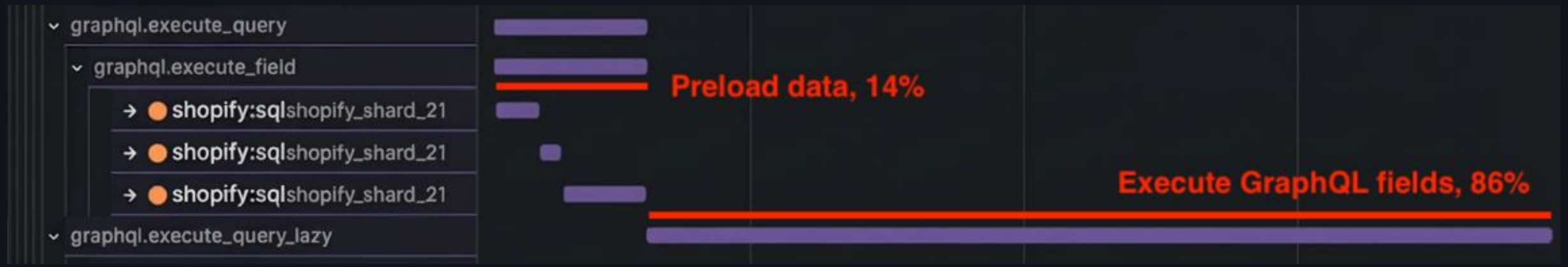
BLOG | DEVELOPMENT

Shopify's journey to faster breadth-first GraphQL execution

Conventional GraphQL execution uses depth-first traversal that incurs many hidden costs. We questioned why, rewrote it in a faster breadth-first manner, and saw dramatic results.

Published on Mar 12, 2026

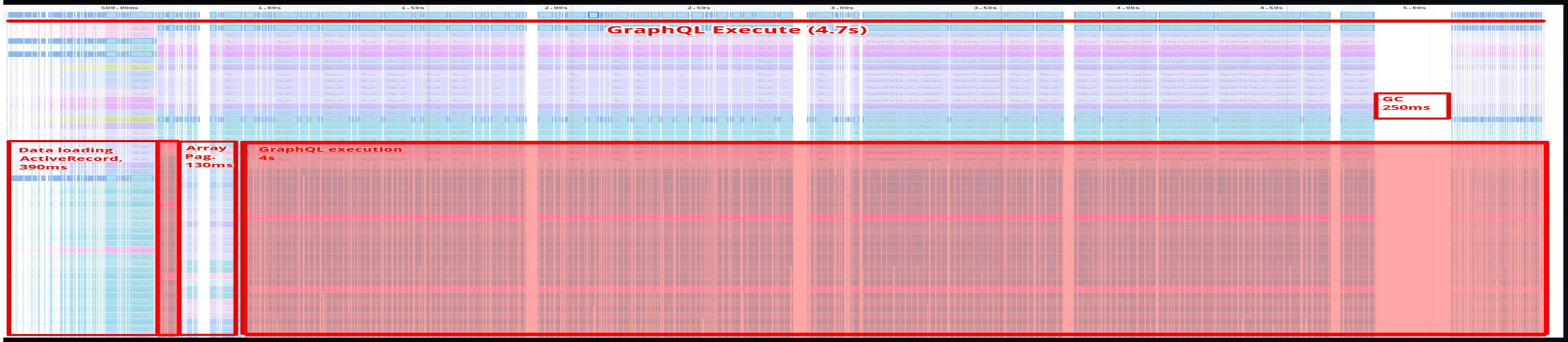
```
query {
  products(first: 250) {
    nodes {
      variants(first: 250) {
        nodes { ... MyBigVariantsSelection }
      }
    }
  }
}
```

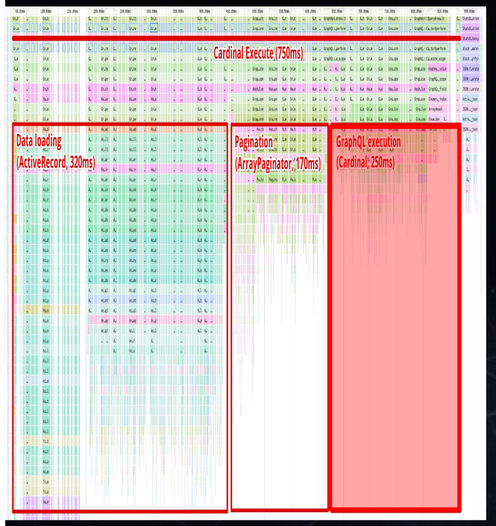


GRAPHQL CARDINAL

High-cardinality GraphQL execution using breadth-first traversal

GraphQL Ruby, 5.5s total, 4s on execution (~300ms GC)





- Both spent ~500ms on data load/pagination
- Cardinal had **16x** faster GraphQL execution / GC
- Improved end-to-end time by **5.7x**

Cardinal, 950ms total, 250ms on execution (~50ms GC)

README

graphql-breadth-js

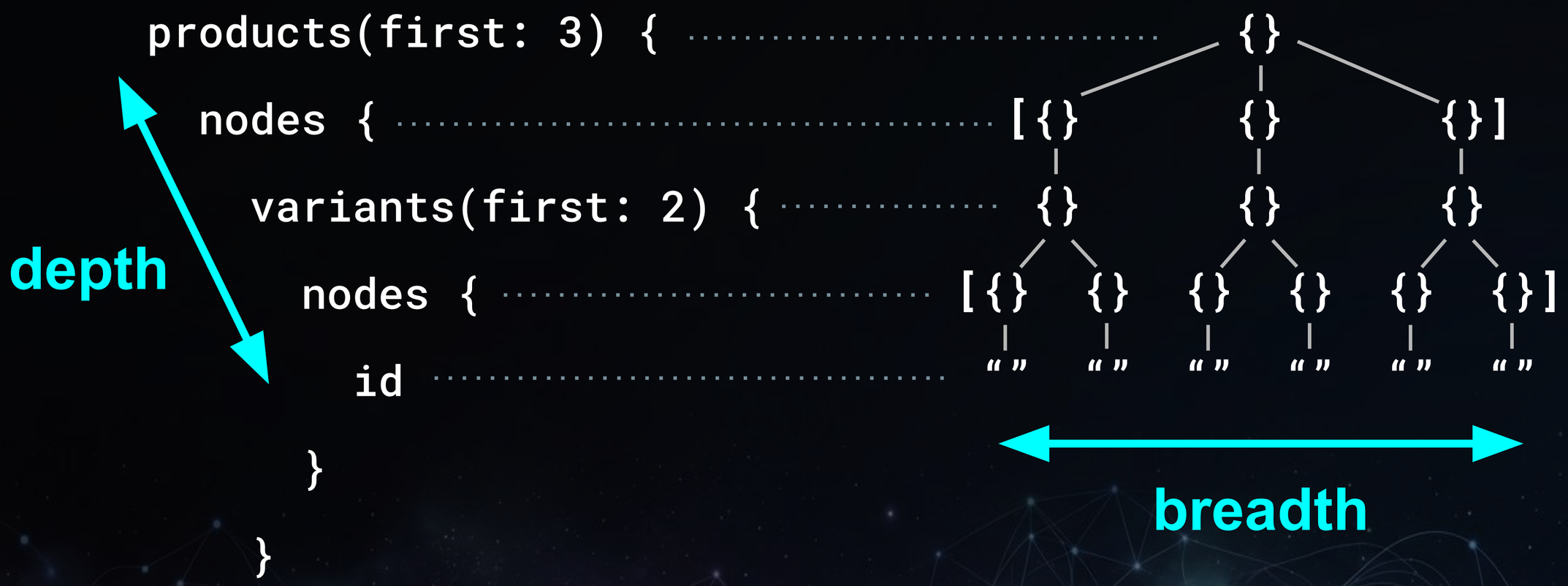
A basic breadth-first GraphQL executor based on Shopify's Cardinal engine. Written in TypeScript, built on top of graphql-js for parsing, type system, and input coercion.

Unlike graphql-js depth traversal, this executor operates breadth-first: every object at a given depth resolves the same field together, then the executor descends a level. This allows per-field overhead to amortize across the entire breadth of a level, and lets lazy loads batch using one Promise *per selection* rather than *per field instance*.

Benchmarks

Single-machine numbers from `pnpm run bench:*` on an M2 MacBook Air, Node 22.

Theory



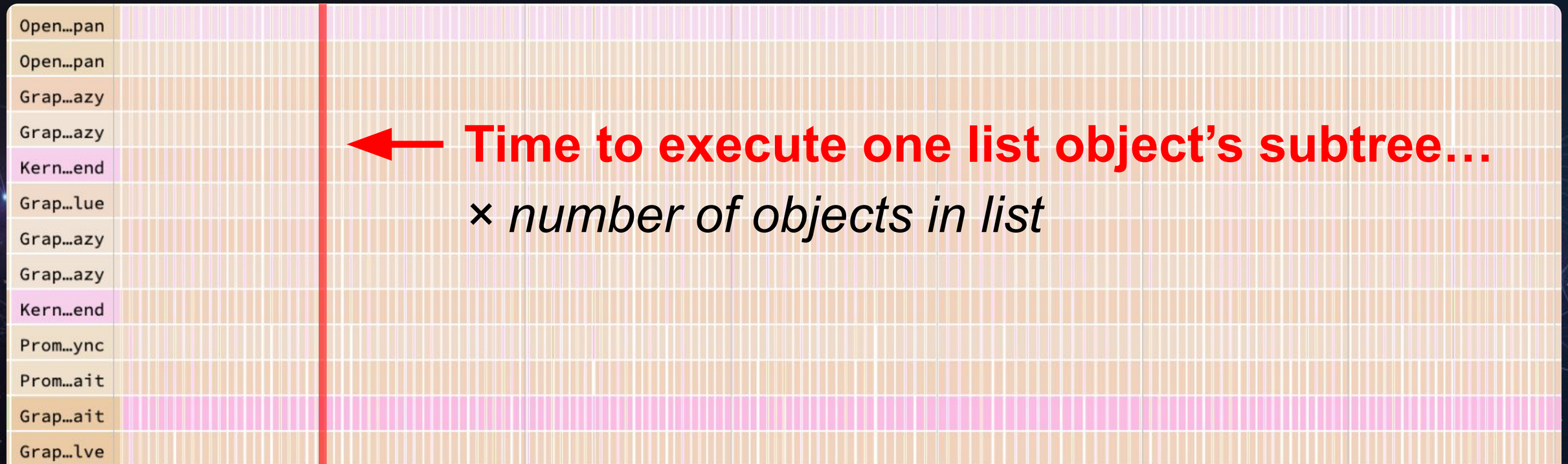
DEPTH: static document size
(large is 100+ fields)

BREADTH: dynamic results size
(large is 10K+ objects)

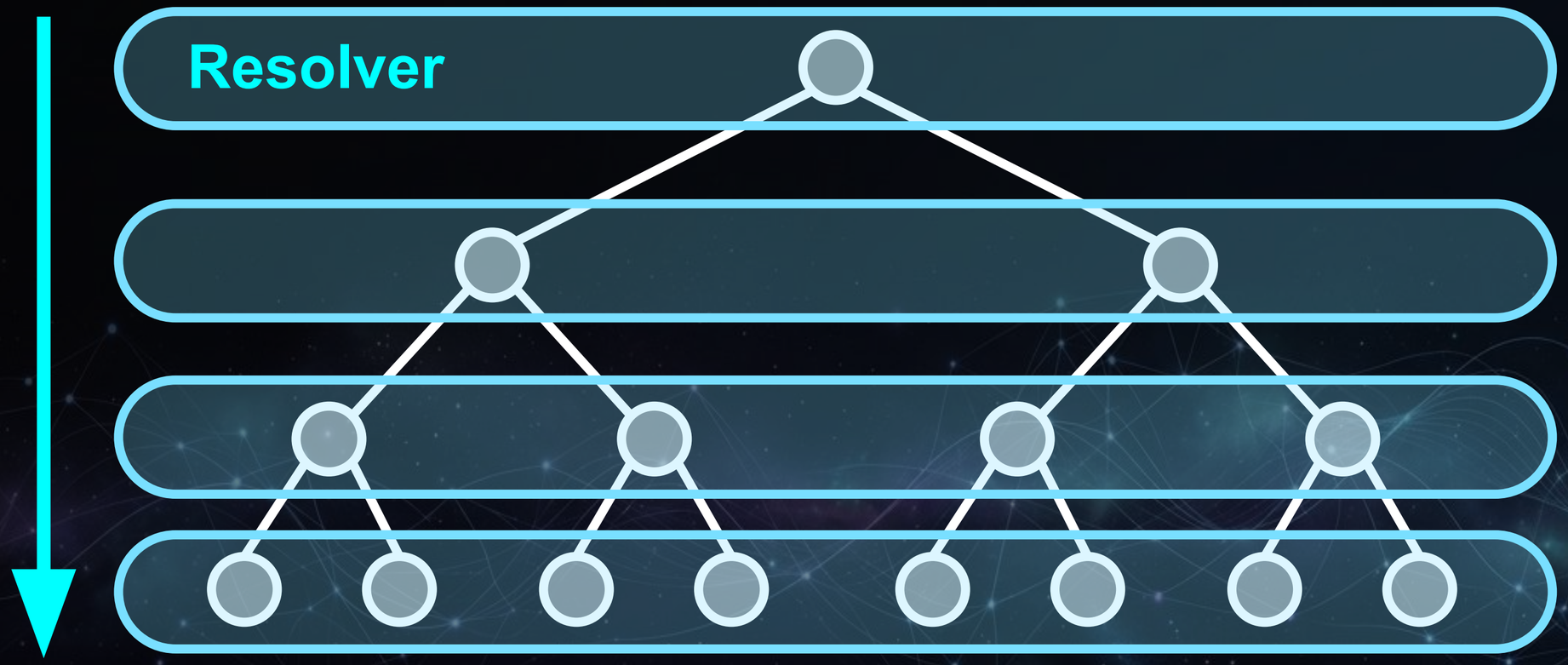
Almost every (public) GraphQL implementation uses depth-first execution*

* Almost…

Depth-first traversal has hidden linear costs.



Breadth-first execution inverts the cost model.



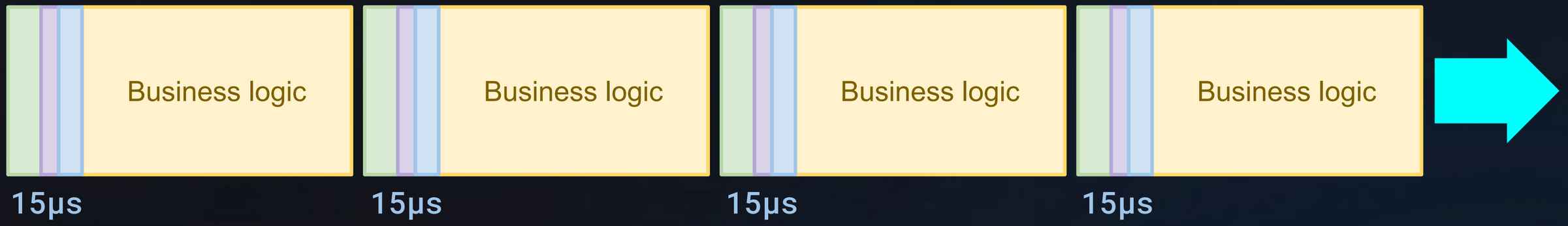
Depth-first resolver (receive and return **one**)

```
def resolve(object, args, context)
  object["item"]
end
```

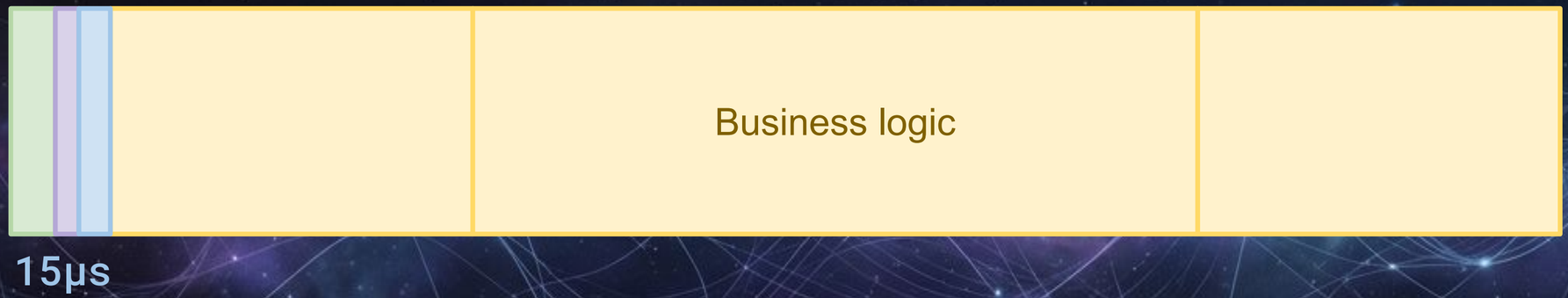
Breadth-first resolver (receive and return **many**)

```
def resolve(objects, args, context)
  objects.map { |object| object["item"] }
end
```

Depth-first resolvers run N-times, each with setup costs*



A breadth-first resolver runs once with flat cost



***Authorization, instrumentation, validation, etc.**

```
query {
  products(first: 250) {
    nodes {
      id
      title
      variants(first: 100) {
        nodes {
          id
          price
          lazy
        }
      }
    }
  }
}
```

```
cost    count    savings
25µs x 1      = -0
15µs x 1      = -0
15µs x 250    = -3.75ms
15µs x 250    = -3.75ms
25µs x 250    = -6.25ms
15µs x 250    = -3.75ms
15µs x 25000  = -375ms
15µs x 25000  = -375ms
25µs x 25000  = -500ms
-------------------------
                -1.27s
```

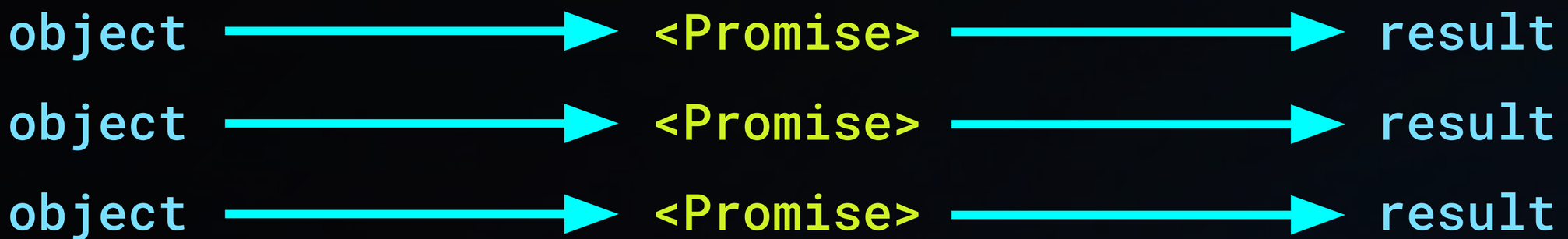
Tiny costs add up.

```
query { widgets(first: N) {
  widget { widget { widget { widget { id } id } id } id } id }
}
```

5 depth x N breadth	Speed	GC Pressure
5 × 1	graphql-js, 1.58× faster	breadth, 2.78× less GC
5 × 10	breadth, 1.87× faster	breadth, 3.45× less GC
5 × 100	breadth, 2.66× faster	breadth, 5.64× less GC
5 × 1,000	breadth, 2.82× faster	breadth, 5.19× less GC
5 × 10,000	breadth, 2.89× faster	breadth, 5.06× less GC

Comparing graphql-js to graphql-breadth-js. Breadth wins with repetition.
Gains plateau as building results starts to constrain both.

Depth-first lazy fields build **N** promises



Breadth-first binds entire field sets to **one** promise

```
query { widgets(first: N) { lazyPromise } }
```

Size	Speed	GC Pressure
1	breadth, 1.02x faster	breadth, 1.53× less GC
10	breadth, 2.10× faster	breadth, 2.96× less GC
100	breadth, 2.82× faster	breadth, 18.7× less GC
1,000	breadth, 3.20× faster	breadth, 47.1× less GC
10,000	breadth, 5.18× faster	breadth, 120× less GC

Gains accelerate as graphql-js promise allocations explode.
At 10K objects, graphql-js spends roughly a quarter of its wall-clock in GC.

Mechanics

```
query {
  products(first: 3) {
    variants(first: 2) {
      id
    }
  }
}
```

```
Scope.new(type: Query, fields: {
  "products" => Field.new(Scope.new(type: Product, fields: {
    "variants" => Field.new(Scope.new(type: Variant, fields: {
      "id" => Field.new,
    })
  })
})
```


SCOPE Query
objects: [shop]
results: [{}]
FIELD products(3)
resolved: nil
SCOPE Product
objects: []
results: []
FIELD variants(2)
resolved: nil
SCOPE Variant
objects: []
results: []
FIELD id
resolved: nil
root object
and result

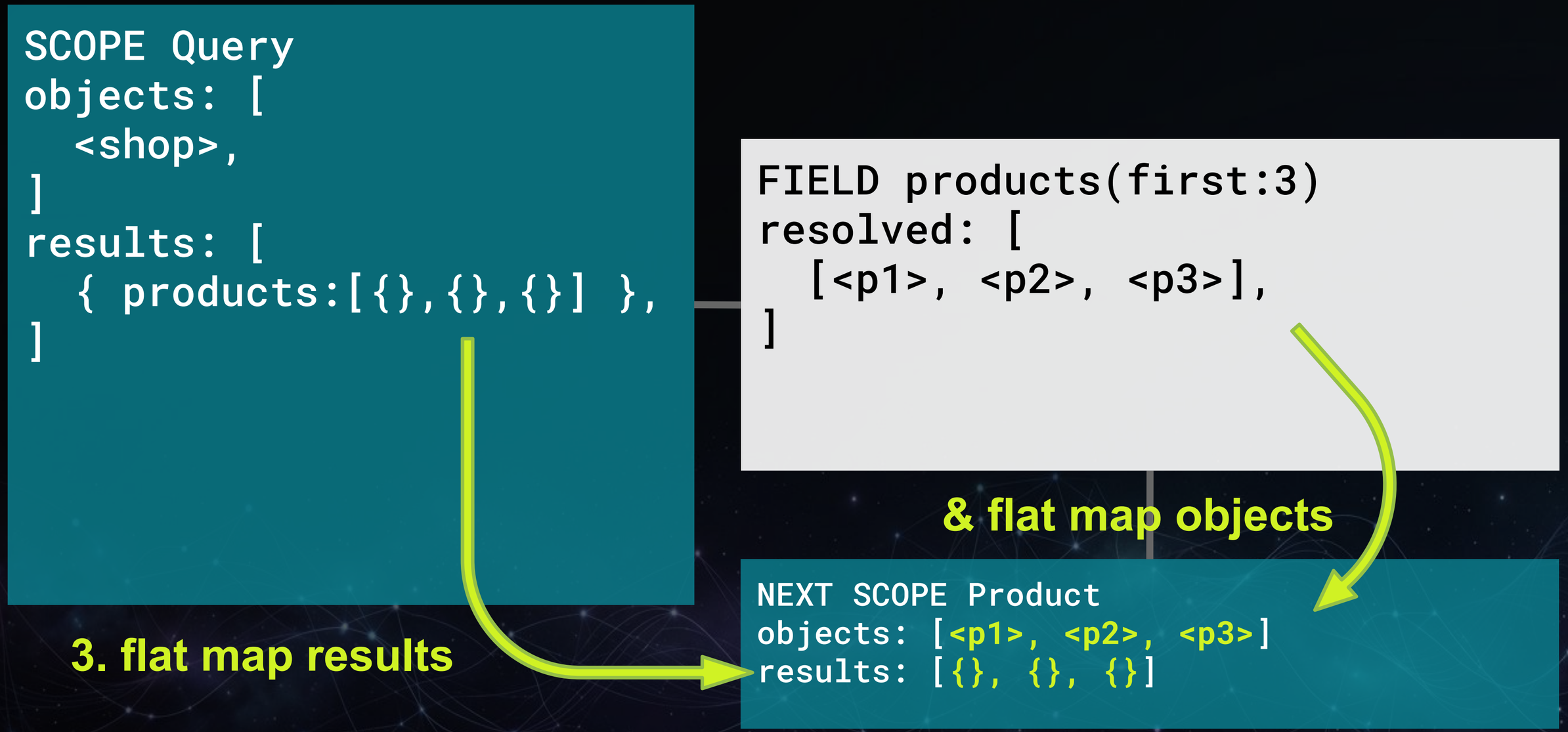

SCOPE Query
objects: [
<shop>,
]
results: [
{},
]
1. resolver
FIELD products(first:3)
resolved: [
[<p1>, <p2>, <p3>],
]
NEXT SCOPE Product
objects: []
results: []

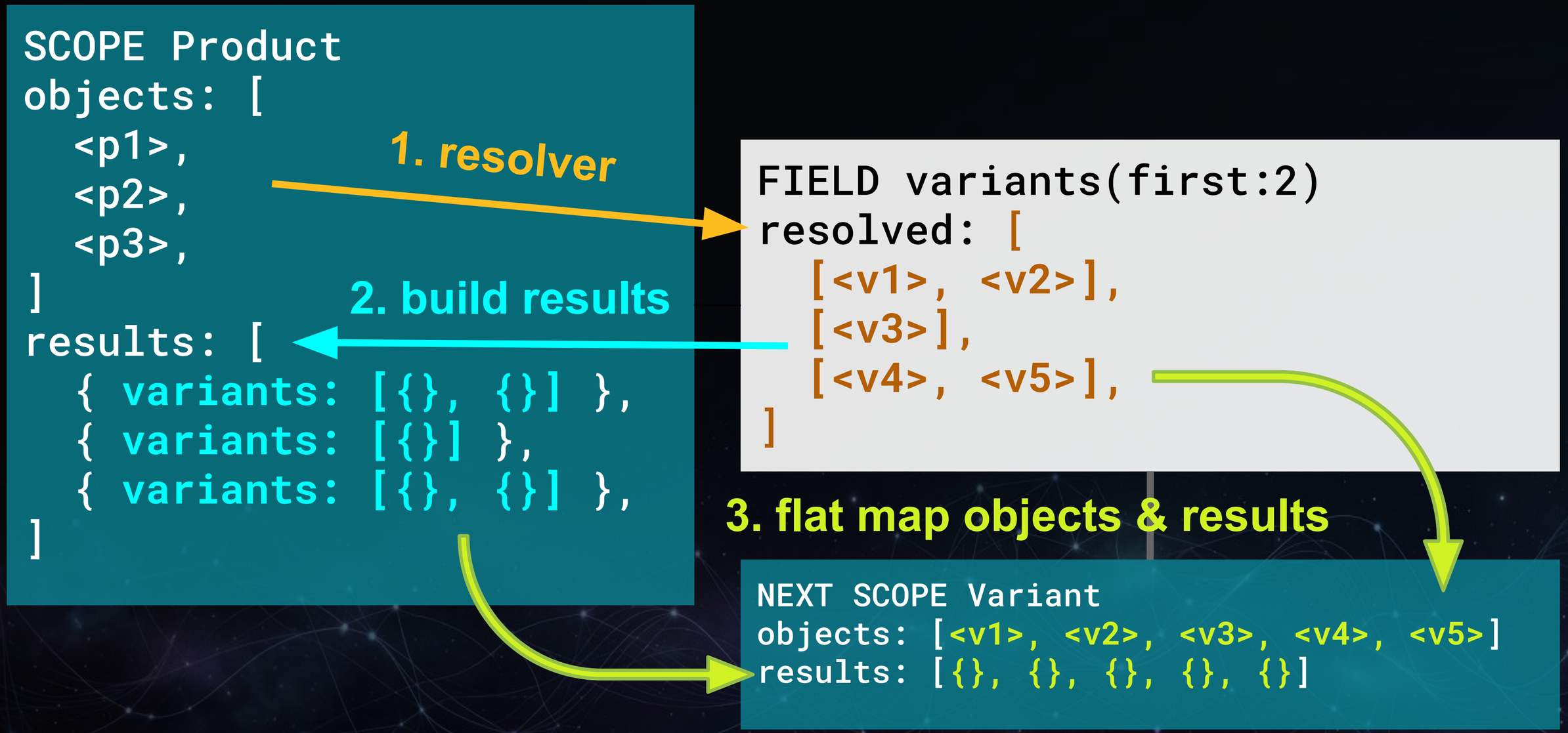
```
SCOPE Query
objects: [
  <shop>,
]
results: [
  { products:[{},{},{}] },
]
```

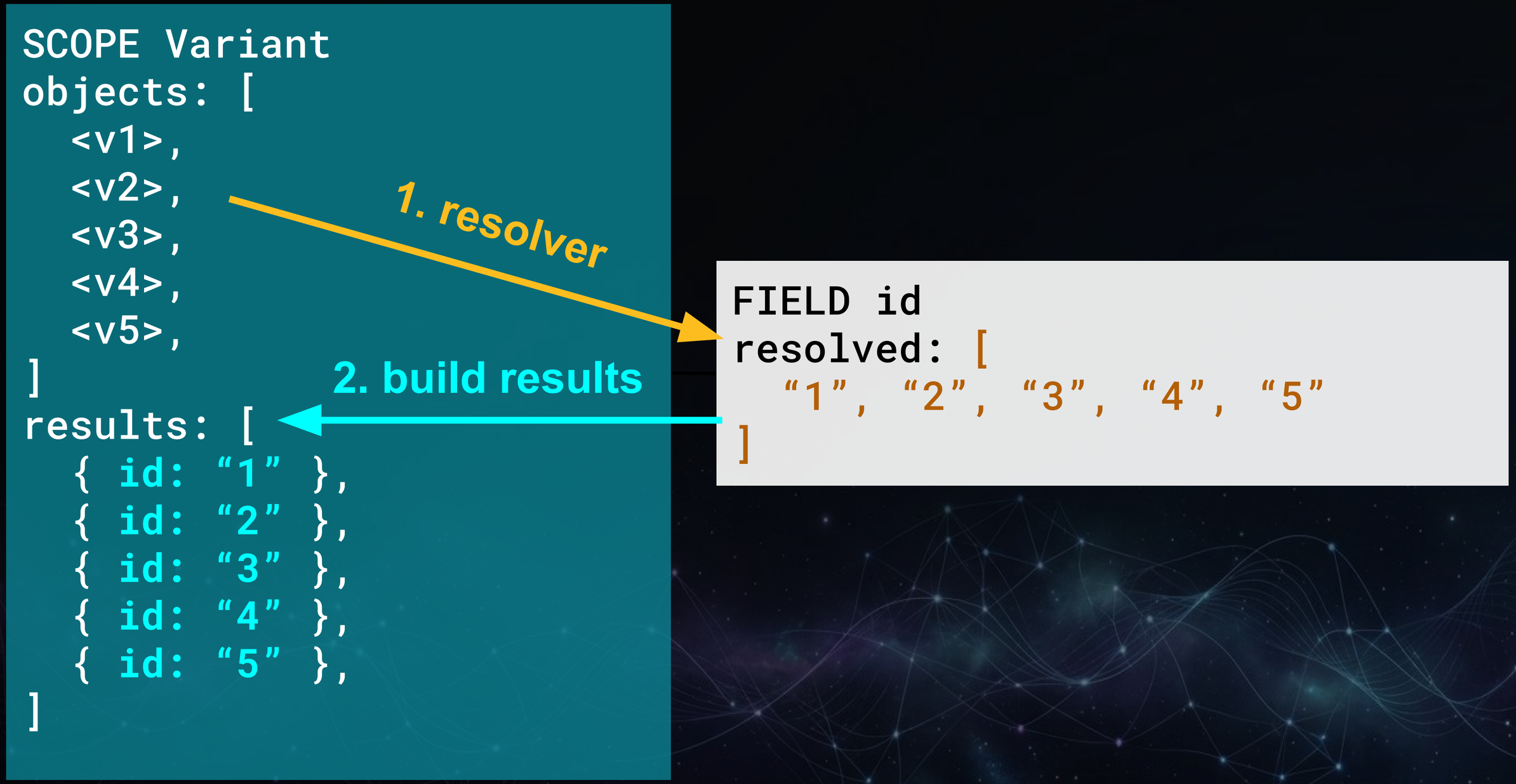
2. build results

```
FIELD products(first:3)
resolved: [
  [<p1>, <p2>, <p3>],
]
```

```
NEXT SCOPE Product
objects: []
results: []
```


SCOPE Query
objects: [
<shop>,
]
results: [
{ products:[{},{},{}] },
]
FIELD products(first:3)
resolved: [
[<p1>, <p2>, <p3>],
]
& flat map objects
NEXT SCOPE Product
objects: [<p1>, <p2>, <p3>]
results: [{}, {}, {}]
3. flat map results

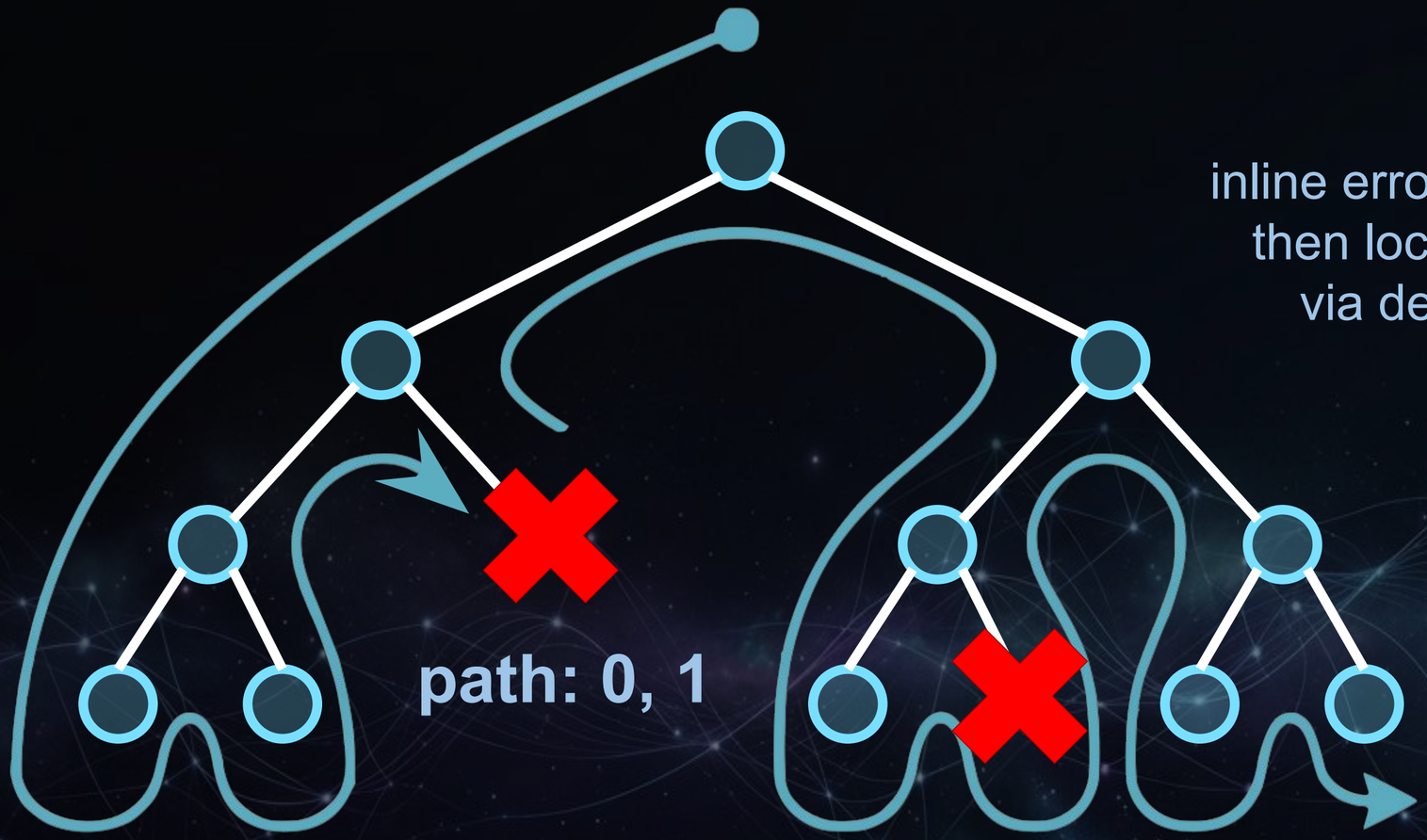

SCOPE Product
objects: [
<p1>,
<p2>,
<p3>,
]
results: [
{ variants: [{}, {}] },
{ variants: [{}] },
{ variants: [{}, {}] },
]
1. resolver
2. build results
FIELD variants(first:2)
resolved: [
[<v1>, <v2>],
[<v3>],
[<v4>, <v5>],
]
3. flat map objects & results
NEXT SCOPE Variant
objects: [<v1>, <v2>, <v3>, <v4>, <v5>]
results: [{}, {}, {}, {}, {}]


SCOPE Variant
objects: [
<v1>,
<v2>,
<v3>,
<v4>,
<v5>,
]
results: [
{ id: “1” },
{ id: “2” },
{ id: “3” },
{ id: “4” },
{ id: “5” },
]
1. resolver
2. build results
FIELD id
resolved: [
“1”, “2”, “3”, “4”, “5”
]

Final root object shape

```
{
  products: [
    { variants: [{ id: “1” }, { id: “2” }],
    { variants: [{ id: “3” }],
    { variants: [{ id: “4” }, { id: “5” }],
  ]
}
```

Breadth execution can still achieve early abort and spec-compliant errors.



inline error results,
then locate them
via depth pass


Join us!
GRAFAST!


POWERED BY SHOPIFY
POWERED BY SHOPIFY
Thank you