Inside the OpenSearch Query Execution Pipeline

following one query from keystroke to ranked results

Samyuktha M S
Software Developer, IBM

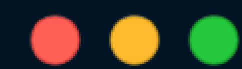
OpenSearchCon, Mumbai
15 June 2026

Why I'm here ?

Hi, I'm Samyuktha

- Software Developer at IBM, ISL
- AI Enthusiast
- *15x Hackathon winner*
- Notable wins - *Google Agentic AI Hackathon, GrabHack 2.0, Chhalaang 4.0, Google Build and Blog Marathon*
- Speaker at GHCI '25, GHC '26, OpenSearch Con India '26, Devfest
- linkedin.com/in/samyuktha-m-s
- github.com/samyuktha-12

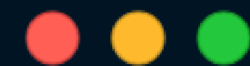
You Pressed Enter



```
GET /reviews/_search

{
  "query": { "match": { "text": "spicy paneer roll" } }
}

→ 200 OK      took: 38 ms      hits: 8,412      returned: 10
```



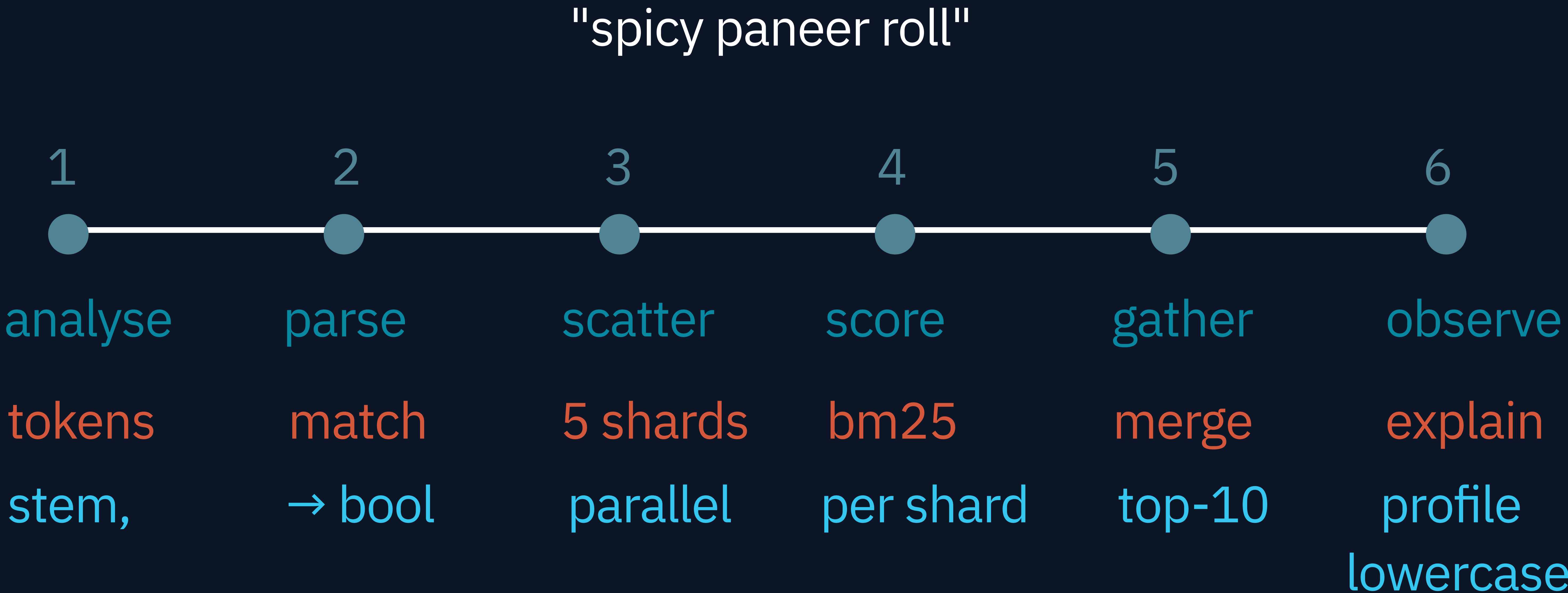
```
in 38 ms, across 1,000,000 reviews:

- analyzed    "spicy paneer roll"   →   [ spici · paneer · roll ]
- fanned      1 coordinator         →   5 shards, in parallel
- matched     8,412 documents
- survived    10
```

One query, six stages

1,000,000 reviews → 10

same index. now you can check every number by hand.



remember three numbers → df: paneer 4 · roll 7 · spici 8

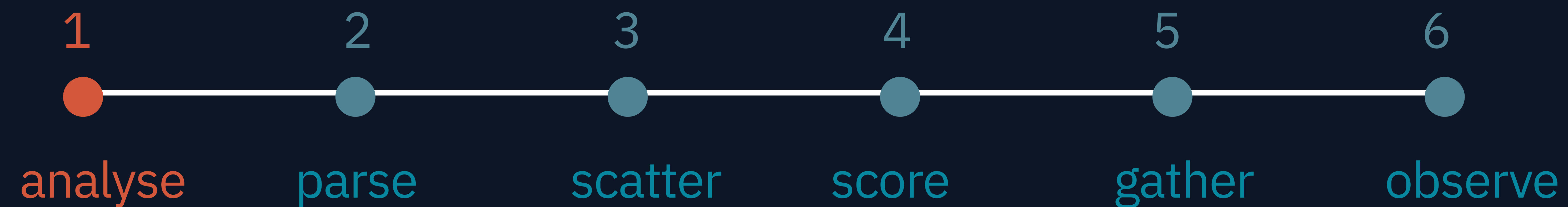
Act I

Nothing is found that was not first taken apart.

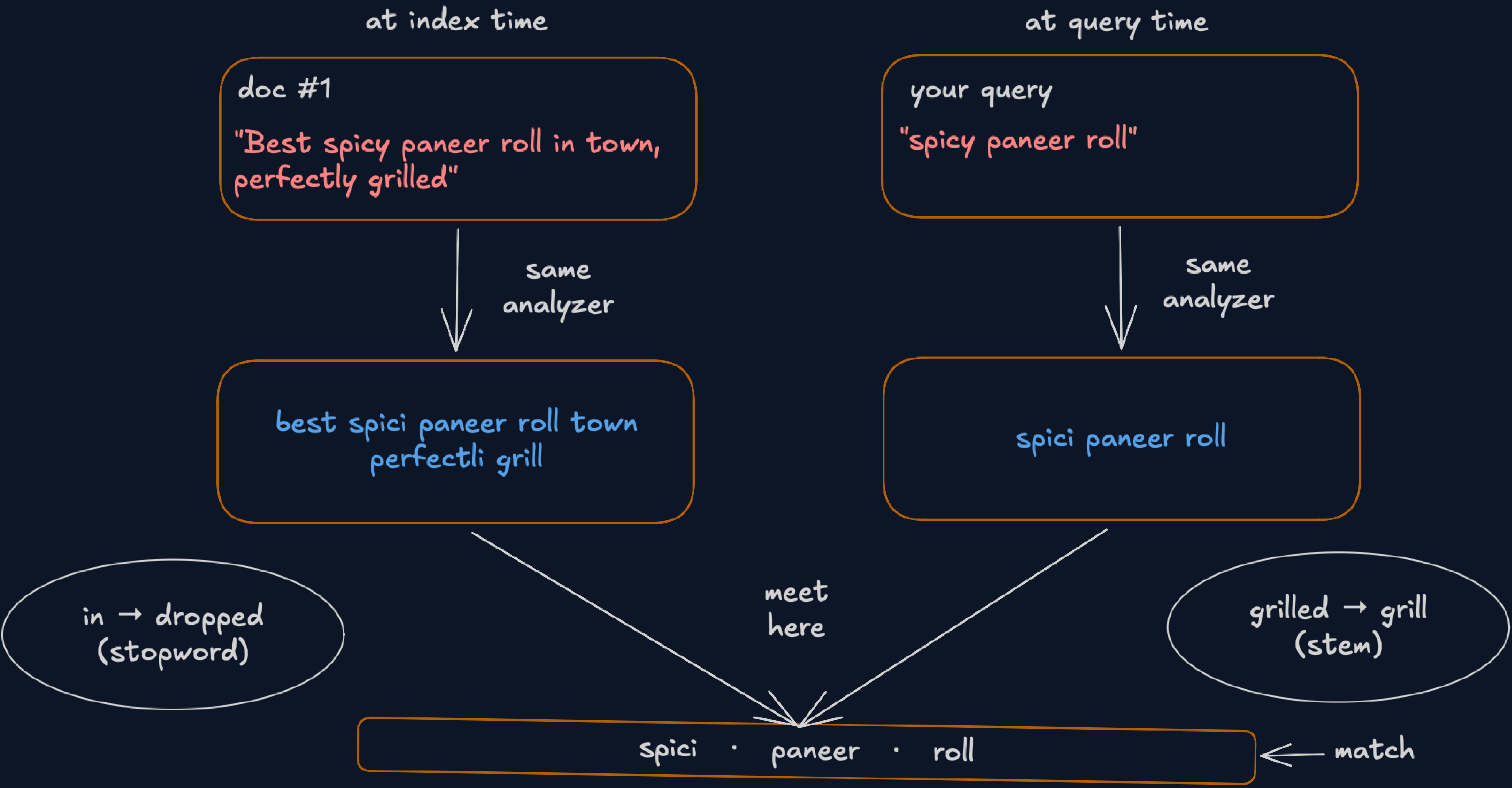
review #1

"Best spicy paneer roll in town, perfectly grilled"

[best · spici · paneer · roll · town · perfectli · grill]



Search never sees your text



The inverted index

not "scan every review for the words"
but "given a word, jump straight to the reviews that hold it"

```
── term ──────── posting list (doc : term-freq) ────────── df ──
paneer    →    1:1   3:1   6:1   9:1                         4
roll      →    1:1   2:1   5:1   6:1   8:1   9:1   10:1      7
spici     →    1:1   2:1   4:1   5:1   7:1   8:1   9:1 …     8
```

query [spici·paneer·roll] → union of 3 lists
candidates → docs 1,2,4,5,6,7,8,9,10
all 3 terms → 1, 9

the shorter the list, the rarer the term, the more it matters

The index is alive

segments are immutable. once written, never changed.
new reviews wait in a buffer until the next refresh.

```
── timeline ──────────────────────────────────────────────────────────

t=0.0s   POST review #11  "spicy paneer wrap"    → in buffer
t=0.3s   GET _search "paneer"                    → 4 hits  (11 not yet)
t=1.0s   refresh                                   buffer → new segment
t=1.1s   GET _search "paneer"                    → 5 hits  (11 visible)
```

seg_1 seg_2 seg_3 seg_4 → merge → seg_A
(old, immutable, many) (background) (fewer)

near-real-time, not real-time. the ~1s you never noticed.

Act II

The index was the past tense.
Now your query arrives, in the present.

the index is built. the documents are tokens. the engine waits.

```
14:32:07.000   GET /reviews/_search  { "spicy paneer roll" }
14:32:07.000   → lands on a node. that node is now in charge.
```

From JSON to a query tree

what you wrote

```
{
  "query": {
    "match": {
      "text": {
        "spicy paneer roll"
      }
    }
  }
}
```

analyze → 3 tokens → 3 leaf queries

match more terms → rank higher

what the query engine saw

```
BooleanQuery (should)
├── TermQuery  text:spici
├── TermQuery  text:paneer
└── TermQuery  text:roll
```

the node that received this is the coordinator.
it does not own the data. it owns the plan.

Two ways to ask

query context → "how well does it match?" scores · no cache
filter context → "does it match? yes / no" no score · cached

one search, both at once

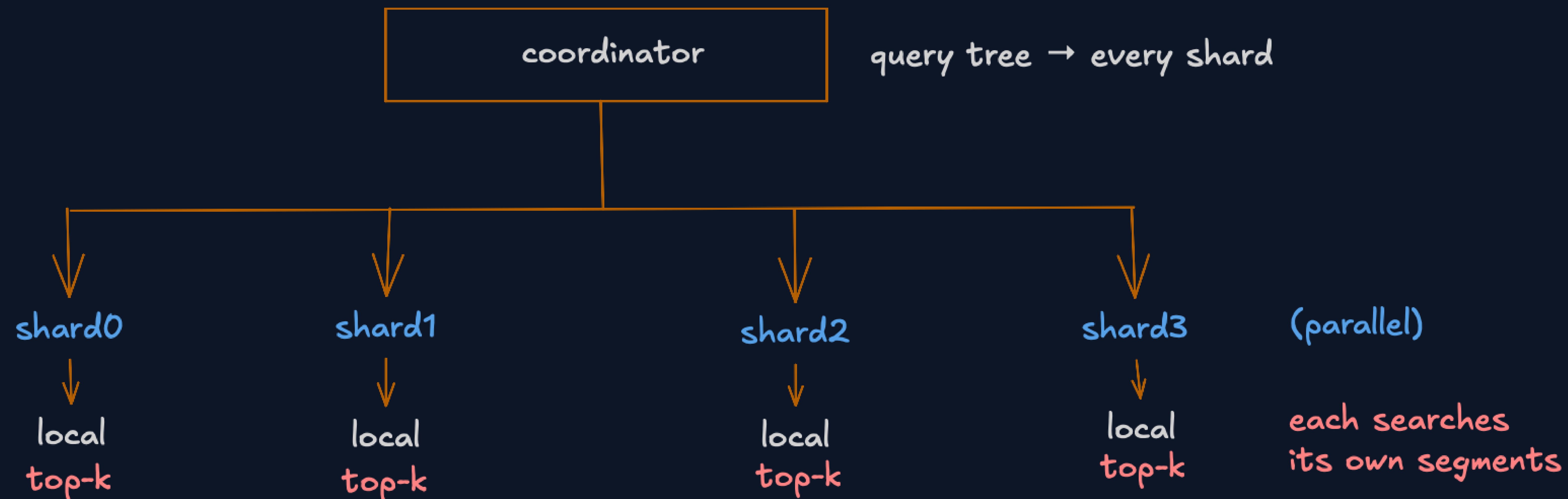
```
{
  "bool": {
    "must":   { "match":  { "text": "spicy paneer roll" } }, ◄ score
    "filter": { "term":   { "city": "mumbai" } }             ◄ yes/no
  }
}
```

city:mumbai → bitset [1,0,1,1,0,...] computed once, reused, cached

the cheapest work is the work you turn into a yes/no.

Scatter

the coordinator does not search. it delegates, then waits.



each shard returns: [doc_id : score] · NOT the documents
shard0 → 42:8.41 17:5.20
shard3 → 42:6.17 88:4.93

4 shards search at the same time. wall-clock = the slowest shard.

How a shard decides what matters

BM25, in three intuitions:

1. rare words matter more → idf
   a. paneer (df 4) > roll (df 7) > spici (df 8)
   b. matching paneer says more than matching spici
2. repetition helps, but fades → tf, saturating
   a. 1 "paneer" → big jump, but 5th "paneer" → barely moves
   b. k1 = 1.2 controls how fast it flattens
3. shorter docs win → length norm
   a. "spicy paneer roll" (3 words) > the same term in 30 words
   b. b = 0.75 controls how much length is penalised

```
score(term) = idf · ( tf saturated, length normalised )
```

The scoring you skip

the shard does NOT score all 8,412 matches.
it refuses to score any document that cannot reach the top 10.

block-max WAND

every block of postings knows its own max possible score

```
if (block max)  <  (10th-best score so far)  →  skip the block.
```

want top 10.   10th-best so far = 6.40

```
block A   max 9.1   →  could win    →  SCORE IT
block B   max 5.2   →  9.1? no, 5.2 < 6.40  →  SKIP, never opened
block C   max 7.8   →  could win    →  SCORE IT

matched: 8,412       actually scored: ~120
```

the fastest scoring is the scoring you never do.

Same document, different score

doc 42 = "Amazing spicy paneer roll, came back twice"
the exact same review. scored on two shards. two answers.

shard0	shard3
local docs: 200,000	local docs: 200,000
paneer df (local): 1,900	paneer df (local): 7,300
idf(paneer) = HIGH	idf(paneer) = LOW
doc 42 → 8.41	doc 42 → 6.17

scores are computed from LOCAL term stats. shards never compare.
→ search_type = dfs_query_then_fetch gathers global stats first
→ one extra round trip. now both shards agree.

Gather

five shards reply. each sends its local top 10. ids + scores only.

50 ranked ids in. 10 ranked ids out. still NO review text.
total data moved so far: a few hundred bytes.

Two phases, on purpose

query_then_fetch. the name is the whole design.

phase 1 : QUERY

ask all 5 shards
"who are your best ids?"

→ 50 ids + scores
→ merge → global top 10

cheap. tiny. every shard.

phase 2 : FETCH

ask only the shards that
own the final 10 ids

→ "give me the actual reviews"
→ doc 42, 09, 17 ... full text

heavy. but only 10 documents.

why not fetch everything in phase 1?

8,412 full reviews × 5 shards across the network ... to show 10.

move ids first. move bodies last. move only what wins.

Why page 1000 is expensive

the two-phase design has one weakness: deep pages.

"from": 0, "size": 10 → each shard returns 10 ids
"from": 10000, "size": 10 → each shard returns 10,010 ids

why? to find the global 10001st, every shard must surrender its top 10,010. the coordinator can't know which shard holds it.

5 shards × 10,010 = 50,050 ids merged → to return 10
sorted, discarded, every single page request.

the fix: don't count pages. remember where you stopped.
search_after (last sort value)
point_in_time (frozen view)
cursor, not offset. cost stays flat at any depth.

The query you never run

the whole pipeline you just watched … skipped, three ways.
a query arrives. how deep does it get before something remembers?

caught here	stores	cleared on
shard request cache same query, same shard	whole response (hits, aggs)	refresh (new segment)
node query cache city:mumbai, reused	filter → bitset [1,0,1,1,0,…]	LRU eviction segment merge
OS page cache Lucene mmaps the files	raw segment bytes (the disk itself)	memory pressure file change

closer to the user = bigger win. the fastest query is no query.

DEMO

Recorded Demo: https://youtu.be/JchOZ5cUBNs

Tuning is knowing the pipeline

every slow search, every bad ranking, lives at one stage.
knowing the stage IS the fix.

```
analyse   →  wrong analyzer = invisible docs. match index & query
parse     →  filter vs query context. yes/no work should cache.
scatter   →  shard count & routing. one slow shard caps you.
score     →  k1, b, boosts. dfs_query_then_fetch for consistency.
gather    →  size, deep paging. search_after, not from:10000.
observe   →  _explain a bad rank. profile a slow query.
```

you no longer have a black box. you have six places to look.

Back to the gap

```
"spicy paneer roll"

analyse   →  torn into tokens, the same way as every review
parse     →  one match became a tree of three
scatter   →  fanned to 5 shards, scored in parallel
score     →  rare words won, most were never scored at all
gather    →  50 ids home, 10 bodies fetched, nothing wasted
observe   →  and half the time, never run at all
```

38 ms no longer a black box.

THANK YOU

https://linktr.ee/opensearchcon_samyuktha

All resources linked here