Architecting a Sub-200ms Product Discovery System

for Ecommerce with OpenSearch and Lucene

Kartik Sapra, Uber

Hi, I am Kartik Sapra

- Software Engineer, Uber
- Ex-Software Engineer, Amazon
- Ex-Senior Software Engineer, Akamai
- GSoC 2019, 2023 @JBoss, @Apache Software Foundation

"Every 100ms of latency
costs 1% in sales revenue"

Amazon

What must happen in 200 milliseconds?

Less than the blink of an eye

① Understand the query

② Search through 50 million products

③ Score and rank by relevance

④ Blend in popularity, freshness, ratings

⑤ Count results per category for filter sidebar

⑥ Return top 20 products with full details

The Scale of the Problem

50M

Products

10K

Queries/sec

< 5s

Data freshness

< 200ms

p99 latency

Ecommerce Search Patterns

What the search page actually does

4 Patterns Behind Every Search Page

Map each UI element to a search engine concept

🔍 **Search Box**	"wireless headphones"	must (text match)
☑️ **Filter Sidebar**	Category, brand, price, rating	filter (yes/no)
🔢 **Counts in Sidebar**	Sony (234), Bose (189)	aggregations
↕ **Sort Dropdown**	Featured (score) / Price / Rating	sort + function_score

"Sort by: Featured" = result ranking = BM25 × popularity × freshness × rating. Other sorts (Price, Rating) skip scoring entirely.

Our Running Example: 8 Products

We'll trace every concept through this catalog

ID	Product Name	Brand	Price	Category	Stock
1	Blue Cotton Casual Shirt	Nike	$29.99	Apparel	✓
2	Red Running Shoes	Adidas	$89.99	Footwear	✓
3	Blue Wireless Headphones	Sony	$149.99	Electronics	✓
4	Cotton Summer Dress	Zara	$49.99	Apparel	✗
5	Blue Running Shoes Pro	Nike	$129.99	Footwear	✓
6	Wireless Noise Cancelling	Sony	$299.99	Electronics	✓
7	Red Cotton T-Shirt	Nike	$19.99	Apparel	✓
8	Running Watch GPS	Garmin	$199.99	Electronics	✓

The Naive Approach

Why databases can't do this

Attempt #1: Use SQL

```
SELECT * FROM products
WHERE name LIKE '%blue%'
   OR name LIKE '%running%'
   OR name LIKE '%shoes%'
ORDER BY ??? -- how do you rank by relevance?
```

✗ Full Table Scan

LIKE '%blue%' checks EVERY row. 50M × 5µs = 250 seconds.

✗ No Relevance Ranking

LIKE returns true/false. Can't distinguish best match from weak match.

✗ No Sidebar Counts

Need a separate GROUP BY query. Double the work.

The Key Insight: Flip the Lookup

❌ Scanning (Database)

Find 'kafka' in a 500-page book:

Read page 1... nope
Read page 2... nope
Read page 3... found it!
Read page 4... nope
... scan all 500 pages

Checks every page. Hours.

✓ Index Lookup (Lucene)

Use the index at the back of the book:

kafka → pages 12, 45, 67
quorum → pages 78, 156
apache → pages 23, 89

Look up word → get page numbers.

3 seconds, not hours.

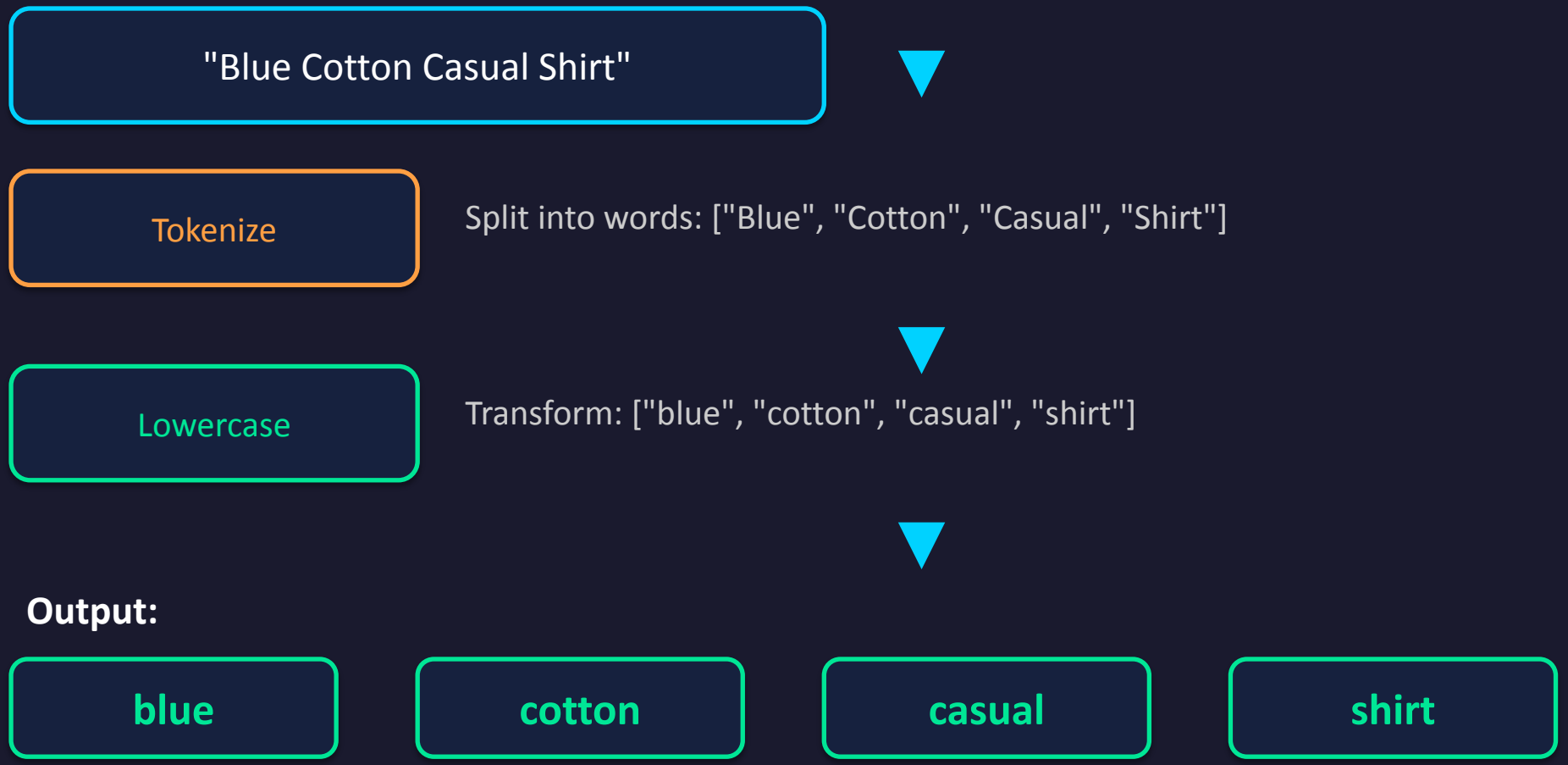
This 'reverse lookup' is called an INVERTED INDEX.
Lucene builds one for your product catalog.

Lucene Deep Dive

How the search engine actually works

Step 1: Analyze Text into Terms

Before indexing, Lucene breaks text into searchable tokens



All 8 Products Analyzed

Same process for every product

Doc 1: "Blue Cotton Casual Shirt"	→	[blue, cotton, casual, shirt]
Doc 2: "Red Running Shoes"	→	[red, running, shoes]
Doc 3: "Blue Wireless Headphones"	→	[blue, wireless, headphones]
Doc 4: "Cotton Summer Dress"	→	[cotton, summer, dress]
Doc 5: "Blue Running Shoes Pro"	→	[blue, running, shoes, pro]
Doc 6: "Wireless Noise Cancelling"	→	[wireless, noise, cancelling]
Doc 7: "Red Cotton T-Shirt"	→	[red, cotton, t, shirt]
Doc 8: "Running Watch GPS"	→	[running, watch, gps]

Step 2: Flip It - Build the Inverted Index

From 'doc contains terms' to 'term appears in docs'

BEFORE (forward index)

Doc 1 contains → [blue, cotton, casual, shirt]

Doc 2 contains → [red, running, shoes]

Doc 5 contains → [blue, running, shoes, pro]

...

To find 'shoes': scan ALL docs

⟹

AFTER (inverted index)

"blue" → [Doc 1, Doc 3, Doc 5]

"running" → [Doc 2, Doc 5, Doc 8]

"shoes" → [Doc 2, Doc 5]

...

To find 'shoes': ONE lookup

The complete inverted index:

blue	→ [1, 3, 5]	cotton	→ [1, 4, 7]	red	→ [2, 7]
running	→ [2, 5, 8]	shoes	→ [2, 5]	wireless	→ [3, 6]

Step 3: Answer a Query - "blue running shoes"

Look up each term:

blue → [1, 3, 5]

running → [2, 5, 8]

shoes → [2, 5]

3 lookups. Not 50 million scans.

Score & Rank:

#1 Blue Running Shoes Pro
3/3 terms ✓ ✓ ✓

#2 Red Running Shoes
2/3 terms ✓ ✓

#3 Running Watch GPS
1/3 terms ✓

#4 Blue Cotton Casual Shirt
1/3 terms ✓

#5 Blue Wireless Headphones
1/3 terms ✓

NOT returned (0 matching terms):

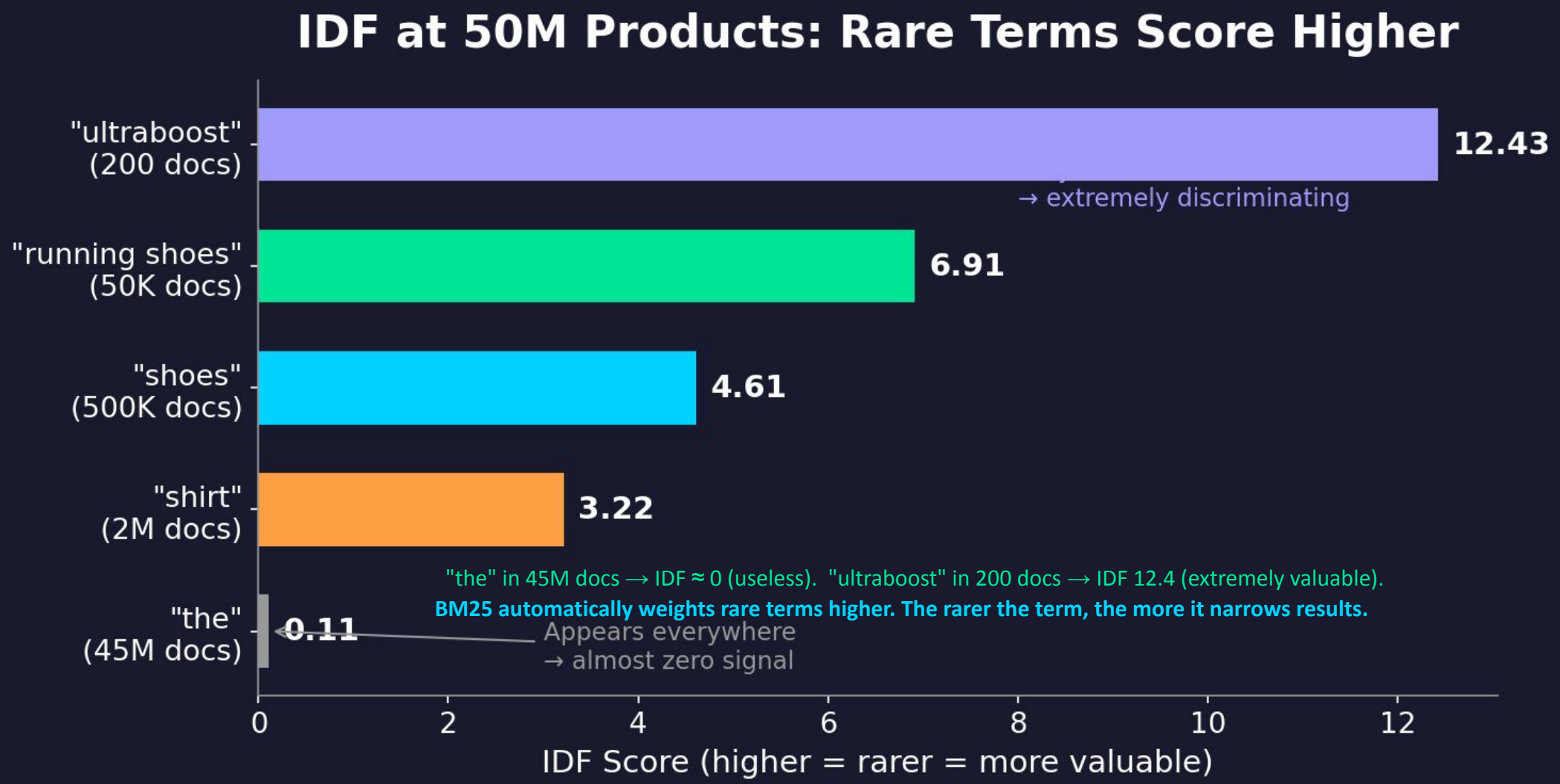
Doc 4: "Cotton Summer Dress"

Doc 6: "Wireless Noise Cancelling"

Doc 7: "Red Cotton T-Shirt"

BM25 Idea #1: Rare Terms Matter More (IDF)

Inverse Document Frequency at 50M product scale



BM25 Idea #2: Shorter Fields Score Higher

A match in a short field is more concentrated

Doc A: "Shoes"

1 token

"shoes" = 100% of the field

Score: HIGH ▲

Doc B: "Premium Italian Leather Shoes"

4 tokens

"shoes" = 25% of the field

Score: LOWER ▼

Same term, same frequency.
But Doc A is a more concentrated match → more relevant.

Beyond Text: function_score

BM25 alone isn't enough - blend text relevance with business signals

final_score = BM25 × popularity × freshness × rating

Product A (wins BM25)

"Running Shoes for Flat Feet"

BM25 = 8.5 (great text match)

Sales: 5 | Rating: 2.1 | 365 days old

final = 8.5 × 1.8 × 0.01 × 2.1 = 0.32

Product B (wins function_score)

"Running Shoes Pro"

BM25 = 7.9 (slightly worse text match)

Sales: 50K | Rating: 4.8 | 7 days old

final = 7.9 × 10.8 × 0.97 × 4.8 = 397.5

Product B wins by 1000×. This is what "Sort by: Featured" does.

How Data Lives on Disk: Segments

Lucene writes immutable chunks - never modifies written data

Segments are IMMUTABLE — once written, never modified:

✓ No locking for reads	Multiple queries read same segment simultaneously
✓ Filesystem cache is perfect	Data never changes → cached copy always valid
✓ Readers and writers don't block	New docs → new segment. Old segments untouched.

Inside a Segment: 4 Specialized Data Structures

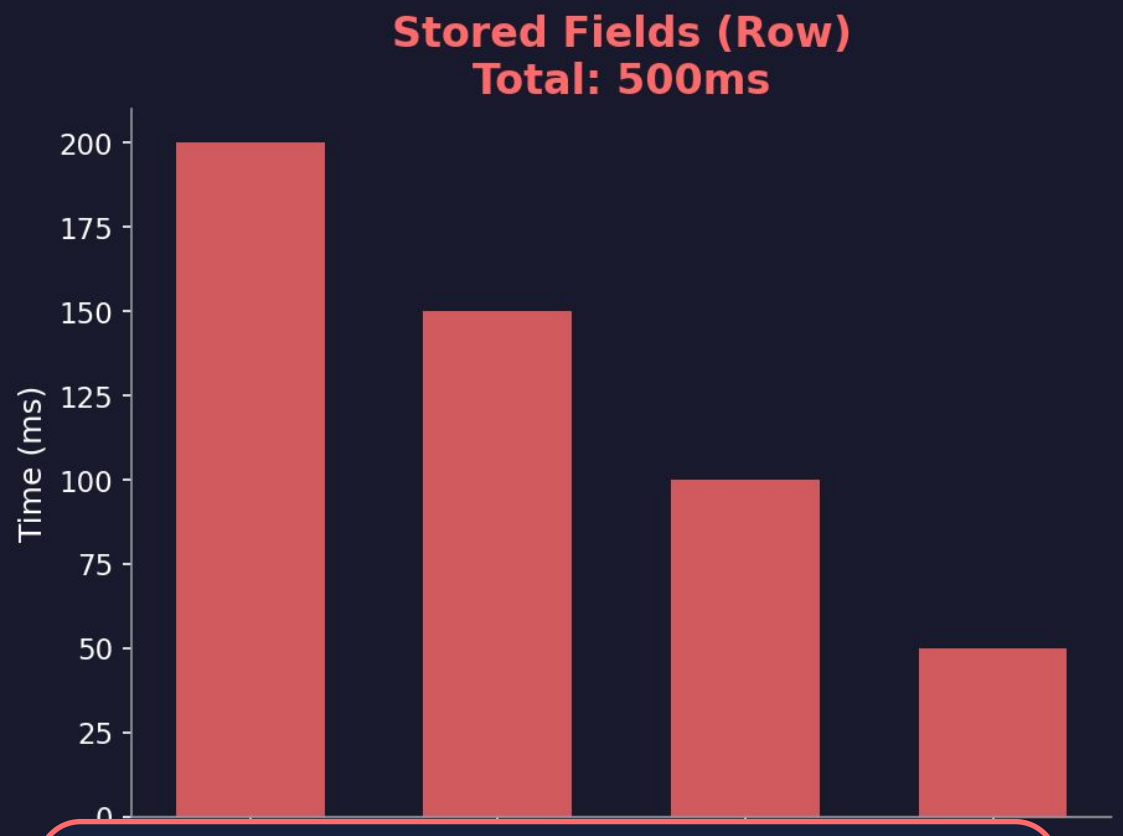
Each one optimized for ONE job

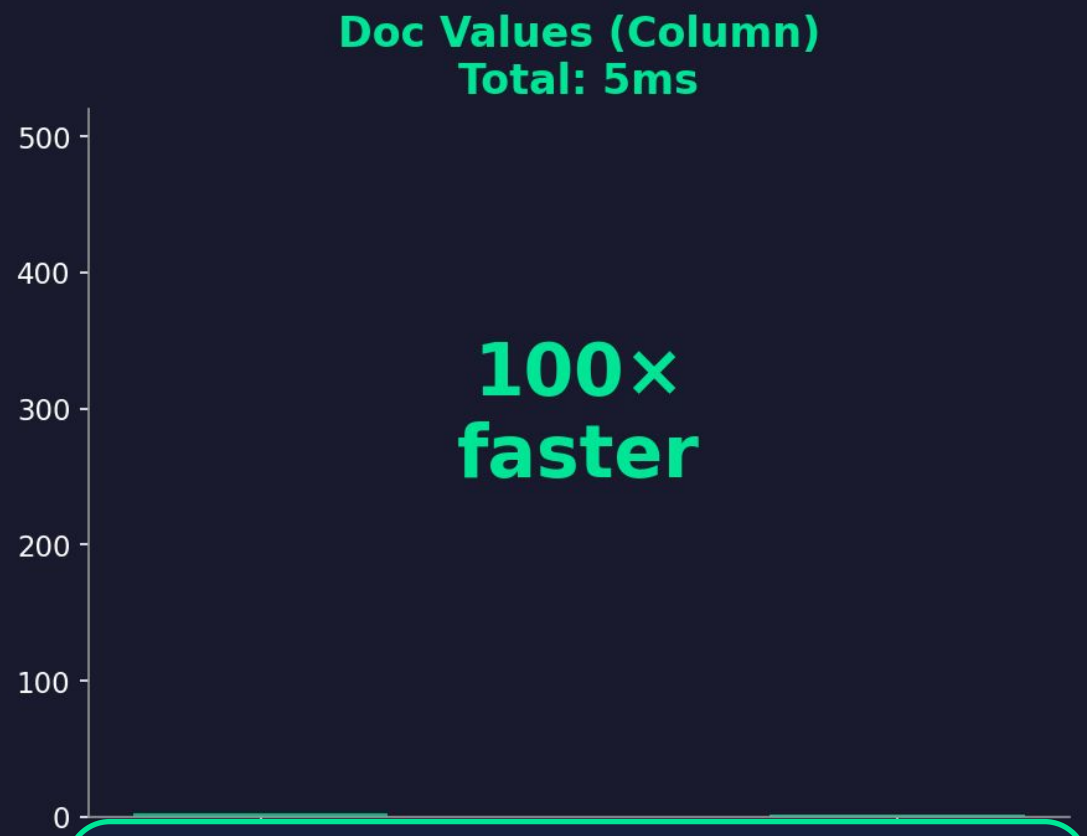
INVERTED INDEX term → doc IDs	For SEARCHING	"shoes" → [Doc2, Doc5]
DOC VALUES doc ID → field value (columnar)	For SORTING & FACETS	price: [29.99, 89.99, 149.99, ...]
STORED FIELDS doc ID → all fields (row-oriented)	For RETURNING results	doc1: {name:"Blue Cotton...", ...}
NORMS doc ID → field length	For SCORING (BM25)	name_length: [4, 3, 3, 3, 4, ...]

Using the wrong structure is 100x slower. Each does ONE thing fast.

Sorting by Price: Doc Values vs Stored Fields

Why column-oriented storage is critical for ecommerce



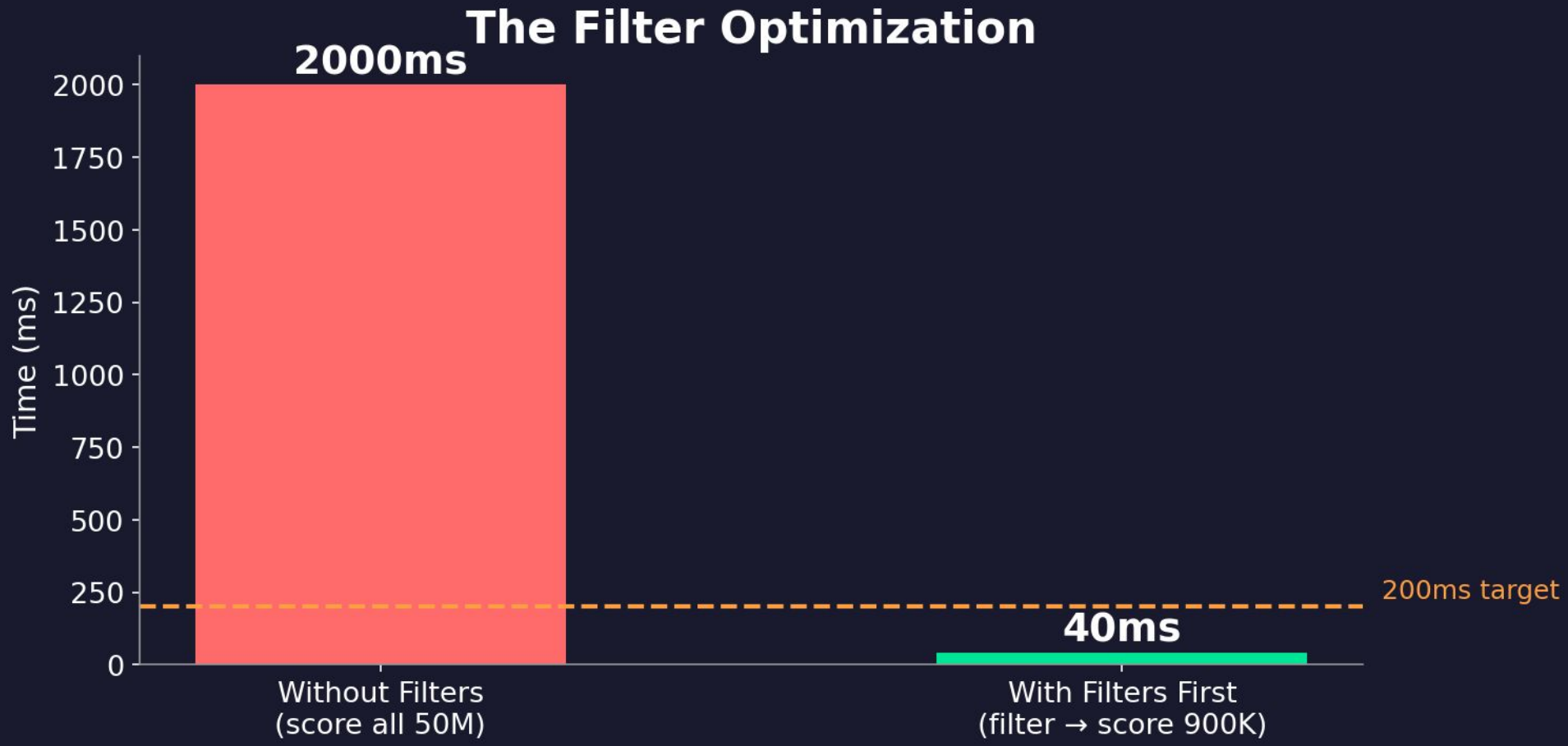


Stored Fields: read each doc, decompress,
parse JSON, extract price field.
50M docs → 500ms

Doc Values: read price column directly.
No decompression. No parsing.
50M docs → 5ms

Filters: The Most Important Optimization

Filter FIRST to eliminate 98% of docs before expensive scoring



① No scoring — 1000× cheaper per doc than BM25

② Cached as bitsets — second query is free

③ Hardware-accelerated — CPU ANDs 512 docs in one instruction

How Filters Work: Each Field Has Its Own Inverted Index

Every filterable field gets a separate inverted index:

category (keyword)		brand (keyword)		in_stock (boolean)	
Apparel	→ [1, 4, 7]	Adidas	→ [2]	true	→ [1,2,3,5,6,7,8]
Electronics	→ [3, 6, 8]	Nike	→ [1, 5, 7]	false	→ [4]
Footwear	→ [2, 5]	Sony	→ [3, 6]		

Multiple filters? AND the bitsets together:

```
category=Electronics:   [0, 0, 1, 0, 0, 1, 0, 1]
AND in_stock=true:      [1, 1, 1, 0, 1, 1, 1, 1]
─────────────────────────────────────────────────
Result:                 [0, 0, 1, 0, 0, 1, 0, 1]  ← Only docs 3, 6, 8

50M docs × 4 filters = 0.4ms (CPU does 512 bits per instruction)
```

What Goes Where: must vs filter

must (scored - expensive)

Only the text search query:

"running shoes"

→ Computes BM25 score per doc
→ Uses inverted index + norms
→ The most expensive step (30-50ms)

ONE clause. That's it.

filter (not scored - free)

Everything that is yes/no:

category = "Footwear"
brand IN ["Nike", "Sony"]
price $50 - $200
rating ≥ 4.0 stars
in_stock = true

→ Cached bitsets, ~0.1ms total

Rule: ONE thing in must, EVERYTHING else in filter.

Testing Your Analyzer

Use the _analyze API to see exactly how text gets tokenized

Request: How does OpenSearch analyze text?

```
POST /products/_analyze
{
  "analyzer": "standard",
  "text": "Blue Running Shoes Pro"
}
```

Response:

```
{
  "tokens": [
    { "token": "blue",
      "position": 0 },
    { "token": "running",
      "position": 1 },
    { "token": "shoes",
      "position": 2 },
    { "token": "pro",
      "position": 3 }
  ]
}
```

Custom analyzer with stemming + synonyms:

```
POST /products/_analyze
{
  "tokenizer": "standard",
  "filter": ["lowercase", "stemmer"],
  "text": "Running Shoes"
}
```

With stemmer:

```
"tokens": [
    { "token": "run",  "position": 0 },
    { "token": "shoe", "position": 1 }
  ]
  → "running" matches "run", "runs", "runner"
```

The General Query Structure

Every ecommerce search query follows this template

```
{
  "query": {
    "function_score": {              ← OPTIONAL: business signals
      "query": {
        "bool": {
          "must": {                  ← TEXT SEARCH (scored)
            "match": {
              "name": "running shoes"
            }
          },
          "filter": [                ← YES/NO FILTERS (free)
            {"term": {"in_stock": true}},
            {"term": {"category": "Footwear"}},
            {"range": {"price": {"gte":50,"lte":200}}},
            {"range": {"avg_rating": {"gte": 4.0}}}
          ]
        }
      },
      "functions": [                 ← RANKING BOOSTS
        {"field_value_factor":
          {"field":"sales_count","modifier":"log1p"}},
        {"gauss":
          {"created_at":{"origin":"now","scale":"30d"}}}
      ],
      "boost_mode": "multiply"
    }
  },
  "aggs": {                          ← SIDEBAR COUNTS
    "brands": {"terms": {"field": "brand"}},
    "prices": {"range": {"field": "price", ...}}
  },
  "sort": [{"price": "asc"}],        ← OPTIONAL: override score sort
  "size": 20                                  ← HOW MANY RESULTS
}
```

Component Breakdown:

bool.must
Text search query
Always present

bool.filter
Category, brand, price, stock
Always present

function_score
Popularity × freshness × rating
Only for 'Featured' sort

aggs
Counts for filter sidebar
When sidebar is shown

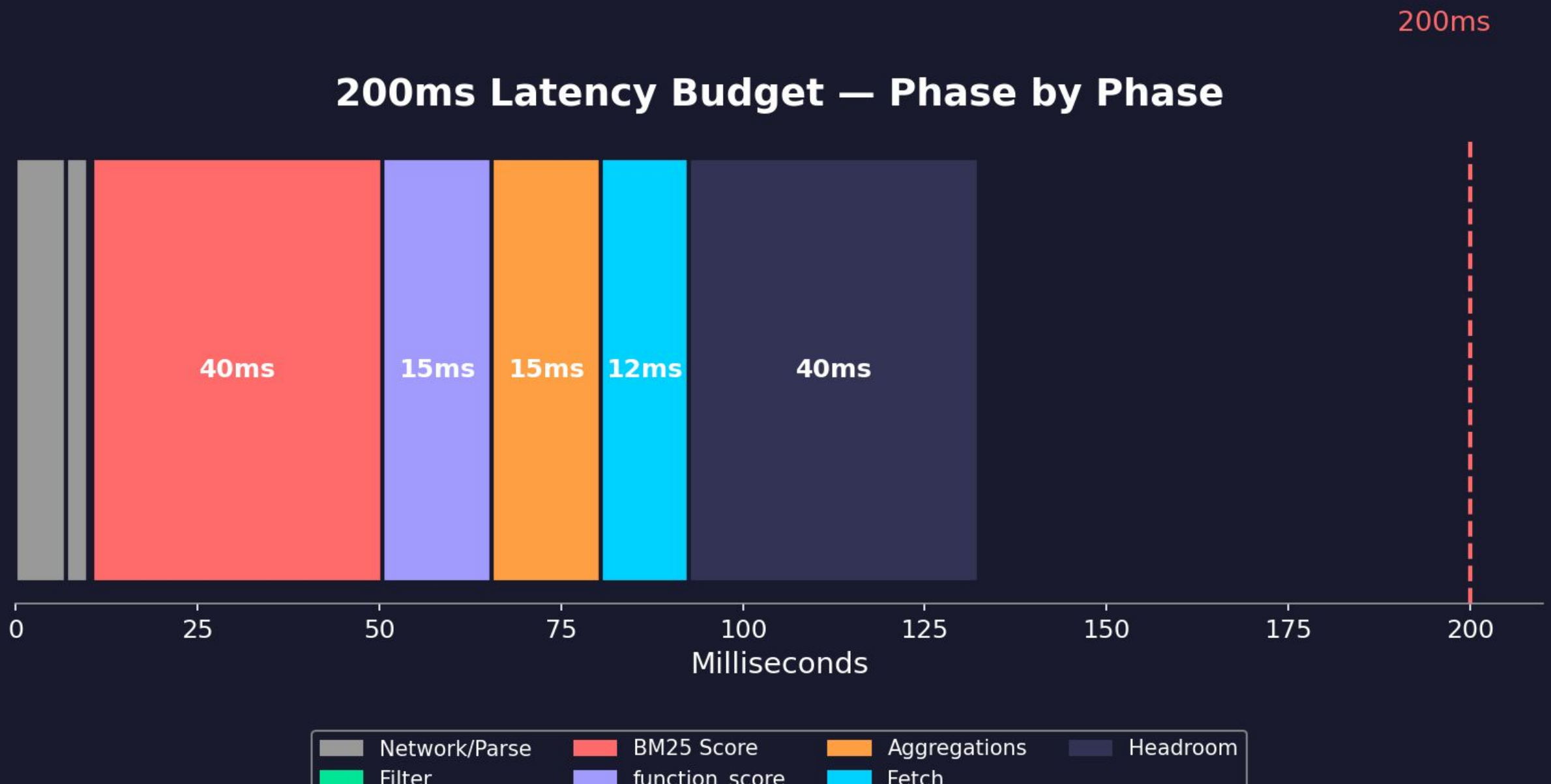
sort
Price / rating / newest
Only for field sort
(replaces score sort)

size
Number of results
Always present

The 200ms Latency Budget

Where every millisecond goes

Breaking Down 200 Milliseconds



OpenSearch

Lucene on steroids - distributed, fault-tolerant, production-ready

From Library to Distributed System

Lucene = Engine

✓ Inverted index + BM25
✓ Segments + doc values
✓ Fast single-machine search

✗ Single machine only
✗ No HTTP API
✗ No fault tolerance

OpenSearch = Car

✓ Everything Lucene does
✓ Distributed across 100+ machines
✓ Automatic fault tolerance
✓ REST API (JSON in/out)
✓ Plugin system (LTR, neural)
✓ Monitoring, security, alerting

Same engine, now across 100 machines.

Cluster Architecture

Three node types, three jobs

COORDINATING NODE
Route queries, merge results

Replicas on DIFFERENT nodes - if Node A dies, Node B has copies of everything.

DATA NODE A

Shard 0 PRIMARY
Docs 1, 2, 3, 4 [Lucene engine inside]

Shard 1 REPLICA (backup of Node B)

DATA NODE B

Shard 1 PRIMARY
Docs 5, 6, 7, 8 [Lucene engine inside]

Shard 0 REPLICA (backup of Node A)

3× CLUSTER MANAGERS - track shard locations, handle failures, elect leader

What Is a Shard?

A shard IS a complete Lucene index

SHARD 0 (Node A)

Doc 1: Blue Cotton Casual Shirt

Doc 2: Red Running Shoes

Doc 3: Blue Wireless Headphones

Doc 4: Cotton Summer Dress

Own inverted index:

blue → [1, 3]

running → [2]

shoes → [2]

SHARD 1 (Node B)

Doc 5: Blue Running Shoes Pro

Doc 6: Wireless Noise Cancelling

Doc 7: Red Cotton T-Shirt

Doc 8: Running Watch GPS

Own inverted index:

blue → [5]

running → [5, 8]

shoes → [5]

They know NOTHING about each other.
Completely independent Lucene indexes on different machines.

Query Flow: End to End

Tracing 'blue running shoes' through every layer

①	**App → Coordinator** Parse query, look up shard locations	2ms
②	**Fan out to shards** Send query to Node A and Node B in PARALLEL	1ms
③	**Shard execution** Each shard: Filter → Match → Score → Aggregate	60ms
④	**Return partial results** Each shard sends top doc IDs + scores	1ms
⑤	**Coordinator merges** Global top-20 via priority queue	3ms
⑥	**Fetch documents** Get full product JSON for top 20	10ms
⑦	**Response** JSON with hits + aggregation counts	1ms
		= 78ms ✓

Key: latency = SLOWEST shard, not the average. One slow shard ruins everything.

Zoom In: What Happens Inside One Shard

Shard 0 processing 'blue running shoes' + filter in_stock=true

FILTER	in_stock inverted index: true → [1,2,3] (doc4 out!) Time: 0.1ms	Inverted Index → bitset [1,1,1,0]
MATCH	blue→[1,3] running→[2] shoes→[2] Time: 0.5ms	Inverted Index → matching doc IDs
SCORE	doc1: BM25=0.86 doc2: BM25=2.30 doc3: BM25=0.86 Time: 5ms	Inverted Index + Norms
BOOST	doc2: 2.30 × log(10001) = 21.1 Time: 2ms	Doc Values (sales_count column)
AGGREGATE	brand counts: Nike=1, Adidas=1, Sony=1 Time: 3ms	Doc Values (brand column)

Each step uses a DIFFERENT data structure — the one optimized for that exact job.

How to Size Shards

Three factors determine shard count

① Data Size

250GB total / 30GB per shard = ~8 shards

Sweet spot: 10-50GB per shard

② Query Throughput

5,000 QPS ⟶ need replicas to spread load across copies

More replicas = more read capacity (but costs more)

③ Tail Latency

10 shards ⟶ p99 = max(10 responses) ≈ 180ms

More shards = worse p99 (one slow shard ruins it)

Production Sizing Example

50M products, 5,000 QPS, p99 < 200ms

```
Data:      50M products × 5KB = 250GB
Indexed:   ~400GB (with index overhead)
Shards:    400 / 30GB = 12 primary shards
Replicas:  2 copies each
Total:     36 shard copies

QPS per copy: 5,000 / 36 = 139
Node capacity: ~192 QPS (12-core)
Status: 139 < 192 ✓ comfortable

CLUSTER:
  12 data nodes     (64GB RAM, NVMe SSD)
   3 managers       (8GB RAM)
   2 coordinators   (16GB RAM)
  ──────────────────────────────────────
  17 nodes total    (~$15K/month)
```

Primary Shards

12

Replicas Each

2

Total Shard Copies

36

Data Nodes

12

Total Nodes

17

Production at Scale

Real companies, real numbers

Case Studies: This Architecture Works

eBay Platform

- 60+ clusters, 2,000+ nodes
- 18 billion documents indexed/day
- 3.5 billion search queries/day

Key lesson:
"Filter context aggressively"

Shopify - Product Search

- Billions of queries during BFCM
- 284 million requests/minute at peak
- 57.3 petabytes served

Key lesson:
"Sub-second latency across millions of merchants"

Swiggy - In-Engine LTR

- LTR model inside OpenSearch (LambdaMART)
- Eliminated external ML service call
- Saved 10-30ms per query

Key lesson:
"Every network hop counts against 200ms"

The Key Insight

Every Optimization = Not Doing Work

TECHNIQUE	WHAT IT AVOIDS
Inverted Index	Scanning every document
Filter Context	Computing scores for yes/no clauses
Bitset Caching	Re-evaluating repeated filters
Doc Values (columnar)	Decompressing stored fields for sort/agg
Index Sorting	Scanning all matches for top-N
Segment Immutability	Lock contention between readers/writers
In-Engine LTR	Network hop to external ML service
Sharding	One machine doing all the work
Multi-Stage Ranking	Running expensive ML on all candidates

Sub-200ms search is not about making things fast.

It's about not doing things at all.

Resources & References

Primary Sources:

- Lucene Javadoc - lucene.apache.org/core
- OpenSearch Docs - opensearch.org/docs
- Elastic function_score - elastic.co/docs

Case Studies:

- eBay Engineering Blog - innovation.ebayinc.com
- Shopify Engineering - shopify.engineering
- Swiggy LTR (via InfoQ) - infoq.com

Further Reading:

- OpenSearch Concurrent Segment Search Blog
- Elastic BM25 Deep Dive Blog
- Vespa - Improving Product Search with LTR

Thank You!

Connect with me

LinkedIn

YouTube