

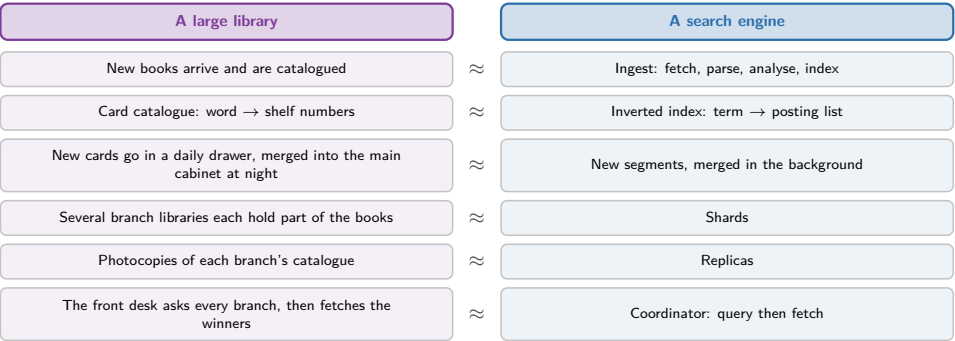


Search Engine Architecture

Search and recsys 01 | Crawl and ingest, inverted-index internals (segments, postings compression, refresh and merge, dynamic pruning), sharding and replication, the query-then-fetch serving path and tail latency, Lucene, Elasticsearch, OpenSearch, Solr and Vespa as of 2026

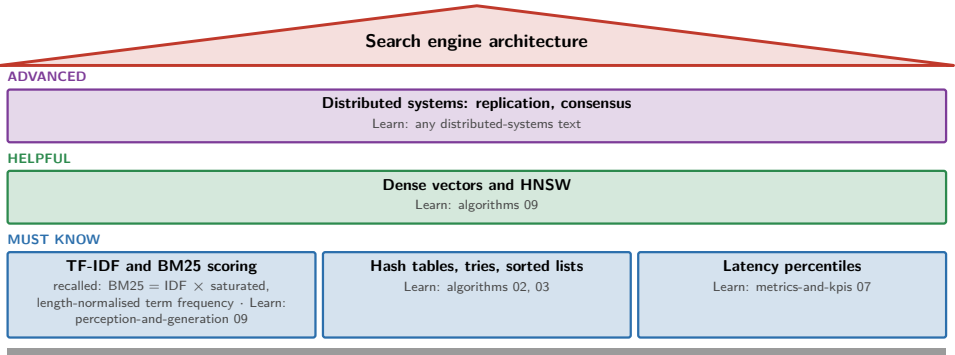
#Search #InvertedIndex #Lucene #Elasticsearch #OpenSearch #Vespa

🔍 The big picture



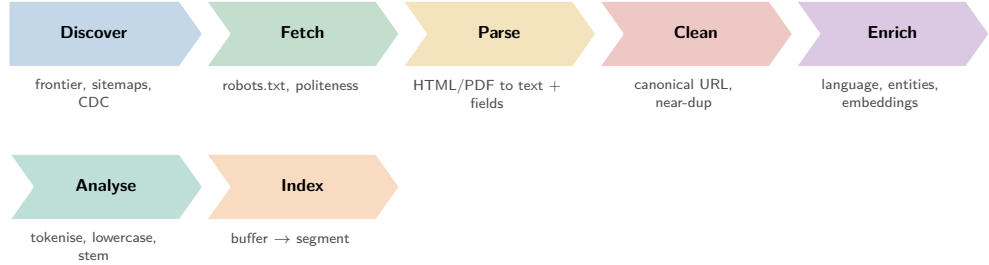
WHERE THE ANALOGY BREAKS A library answers one reader in minutes; a search engine answers thousands of queries per second in under 100 ms, so most of its design is about pruning work and hiding slow machines.

A search engine is a library whose card catalogue is rebuilt every second



What to know first

🔄 Crawl and ingest



From a source to a searchable document. Web search starts with a crawler; site and enterprise search start with connectors or change-data-capture

Concern	Mechanism	Why
Politeness	per-host queues in the URL frontier, crawl delay, honour robots.txt (RFC 9309, 2022)	avoid overloading or being blocked by sites
Freshness	recrawl interval from each page's observed change rate; news and prices first	stale snippets and prices lose trust
Duplicates	canonical URL, exact hash, near-duplicate SimHash or MinHash	duplicates waste index space and crowd results
Analysis chain	char filters → tokenizer → token filters (lowercase, stop words, stemming, synonyms)	the same chain must run on documents and queries
Enrichment	language id, entity tags, quality and spam scores, embeddings at index time	precomputed features are free at query time
Schema	one field per role: title, body, anchors, price, date, vector	per-field boosts and filters need separate fields

Crawler and pipeline decisions that decide index quality

⚠️ Index-time vs query-time analysis

A synonym or stemming change applied only at index time needs a full reindex; applied only at query time it is instant but costs query latency. Most teams stem at index time and expand synonyms at query time. Mismatched analysers between index and query are the classic 'it is in the index but search cannot find it' bug.

Inverted index internals

Structure	Holds	Used for
Term dictionary + FST index	every term, sorted, with a pointer to its postings	find a term in a few memory reads
Postings	doc ids as gaps in blocks of 128, packed bit-width per block; freqs	match and score terms
Positions and offsets	token positions per doc	phrase and proximity queries, highlighting
Norms	one byte per doc and field: encoded field length	BM25 length normalisation
Doc values	per-field columns (numbers, keywords)	sort, aggregate, ranking features
Stored fields	compressed original fields, row-wise	return the document in the fetch phase
Points (BKD tree)	numeric, date and geo values	range and geo filters
Vectors (HNSW graph)	dense embeddings, optionally quantised	k-NN and hybrid search
Live docs bitset	which docs are deleted	deletes and updates without rewriting

What one Lucene segment holds (Lucene 10, as of 2026). Each structure is immutable once written

PROBLEM Encode the posting list of term 'lamp' = doc ids [3, 7, 21, 22, 150] with delta (gap) coding plus variable-byte (VByte) encoding. Compare with plain 32-bit integers.

GIVEN

- VByte stores 7 payload bits per byte; values 0-127 take 1 byte, 128-16 383 take 2 bytes
- Gap $g_i = id_i - id_{(i-1)}$, the first gap is the first id

1 Compute the gaps

3, 7-3=4, 21-7=14, 22-21=1, 150-22=128

Why: sorted ids give small positive gaps

2 Bytes per gap

3→1, 4→1, 14→1, 1→1, 128→2

Why: 128 needs 8 bits, more than 7

3 Total

1+1+1+1+2 = 6 bytes vs $5 \times 4 = 20$ bytes

Why: plain ints waste the high bytes

ANSWER 6 bytes instead of 20: a 3.3× saving on this list, more on long lists of frequent terms whose gaps are tiny

CHECK  Lucene goes further: blocks of 128 gaps bit-packed at the block's max bit width (frame of reference), decoded with SIMD-friendly loops

Why postings are stored as gaps

A toy inverted index with AND and top-k (Python)

```
from collections import defaultdict
import heapq, math
docs = {1: "red lamp", 2: "desk lamp led", 3: "red desk"}
index = defaultdict(list) # term -> [(doc, tf)]
for d, text in docs.items(): # ids added in order: sorted
    for t in set(text.split()):
        index[t].append((d, text.split().count(t)))
def search(q, k=2):
    lists = [dict(index[t]) for t in q.split()]
    hits = set.intersection(*(set(l) for l in lists)) # AND
    N = len(docs)
    score = lambda d: sum(1[d] * math.log(N / len(l))
                        for l in lists) # tf-idf
    return heapq.nlargest(k, hits, key=score)
print(search("red lamp")) # [1]
```

Near-real-time (NRT) — Documents become searchable at the next refresh (1 s default in Elasticsearch), not on write; set `refresh_interval` to 30 s or -1 during bulk loads to index faster.

Flush / commit — Lucene commit that fsyncs segments to disk and lets the translog be trimmed.

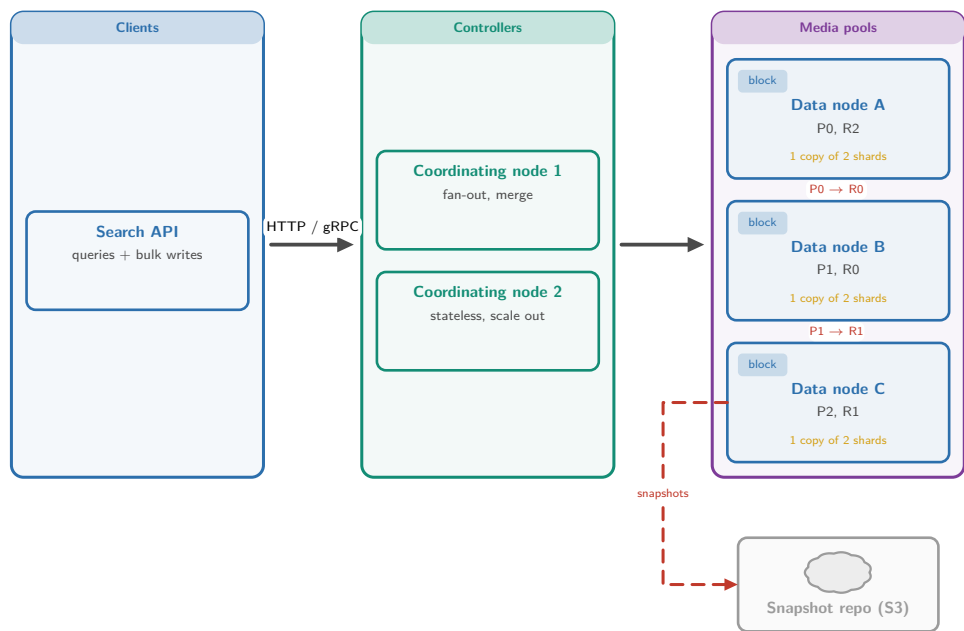
Update — Delete (mark in the live-docs bitset) plus add of a new version; frequent updates to large documents are expensive, partial updates in Vespa avoid this for attribute fields.

Tiered merge policy — Lucene's default: merges segments of similar size; a force-merge to one segment speeds read-only indices but never run it on an index still taking writes.

Top-k without scoring every match

A query like 'the red lamp' matches millions of documents. WAND and Block-Max WAND keep a heap of the k best scores and use each term's maximum score (per list and per block of postings) to skip documents that cannot enter the top k. Consequence: exact total hit counts are expensive, so Elasticsearch counts accurately only up to 10 000 hits by default (`track_total_hits`).

Sharding, replication and the serving path



An index of 3 primaries with 1 replica each on 3 data nodes: a primary and its replica never share a node, so any one node can fail without data loss

Quantity	Formula	Reading
Shard of a document	$s = \text{hash}(r) \bmod P$	Elasticsearch hashes the routing key with Murmur3; default r = document id
Primaries needed	$P = \lceil S/s_{\max} \rceil$	keep each shard roughly 10-50 GB (Elastic guidance)
Copies for throughput	$C = \lceil Q_{\text{peak}}(1+h)/q_{\text{copy}} \rceil$	replicas = $C - 1$, and at least 1 for availability
P(query waits on a slow shard)	$1 - (1 - p)^N$	fan-out turns rare slow shards into common slow queries

where r = routing key (document id or a custom key such as tenant id) (string); P = number of primary shards (count); S = primary data size of the index (GB); $s_{\max}$ = target maximum shard size (GB); Q_{peak} = peak queries per second (QPS); h = headroom fraction for node loss and spikes (0-1); q_{copy} = QPS one full copy of the index sustains at the latency SLO (load-tested) (QPS); C = total copies of every shard (primary + replicas) (count); p = probability that one shard answers slowly (0-1); N = shards a query fans out to (count)

Routing and sizing rules. P is fixed at index creation (changing it means split, shrink or reindex)

Worked example: sizing a product-search cluster
A catalogue index holds $S = 1\,200$ GB of primary data; target $s_{\max} = 40$ GB per shard. Peak $Q_{\text{peak}} = 2\,000$ QPS; a load test shows one full copy sustains $q_{\text{copy}} = 400$ QPS at $p_{99} < 100$ ms; headroom $h = 0.3$. Primaries $P = \lceil 1200 / 40 \rceil = 30$. Copies $C = \lceil (2000 \times 1.3 / 400) \rceil = \lceil 6.5 \rceil = 7$, so 6 replicas and $30 \times 7 = 210$ shards in total. Every query touches all 30 shards, so the slow-shard exposure below applies.

Share of queries that wait on at least one slow shard

FORMULA $E = 1 - (1 - p)^N$

WHERE

- E = share of queries delayed by at least one slow shard (0-1)
- p = probability that one shard answers slowly (GC pause, merge, noisy neighbour) (0-1)
- N = shards a query fans out to (count)

WORKED

- $p = 0.01$
- $N = 30$
- $E = 1 - (1 - 0.01)^{30}$
- $E = 0.2603$

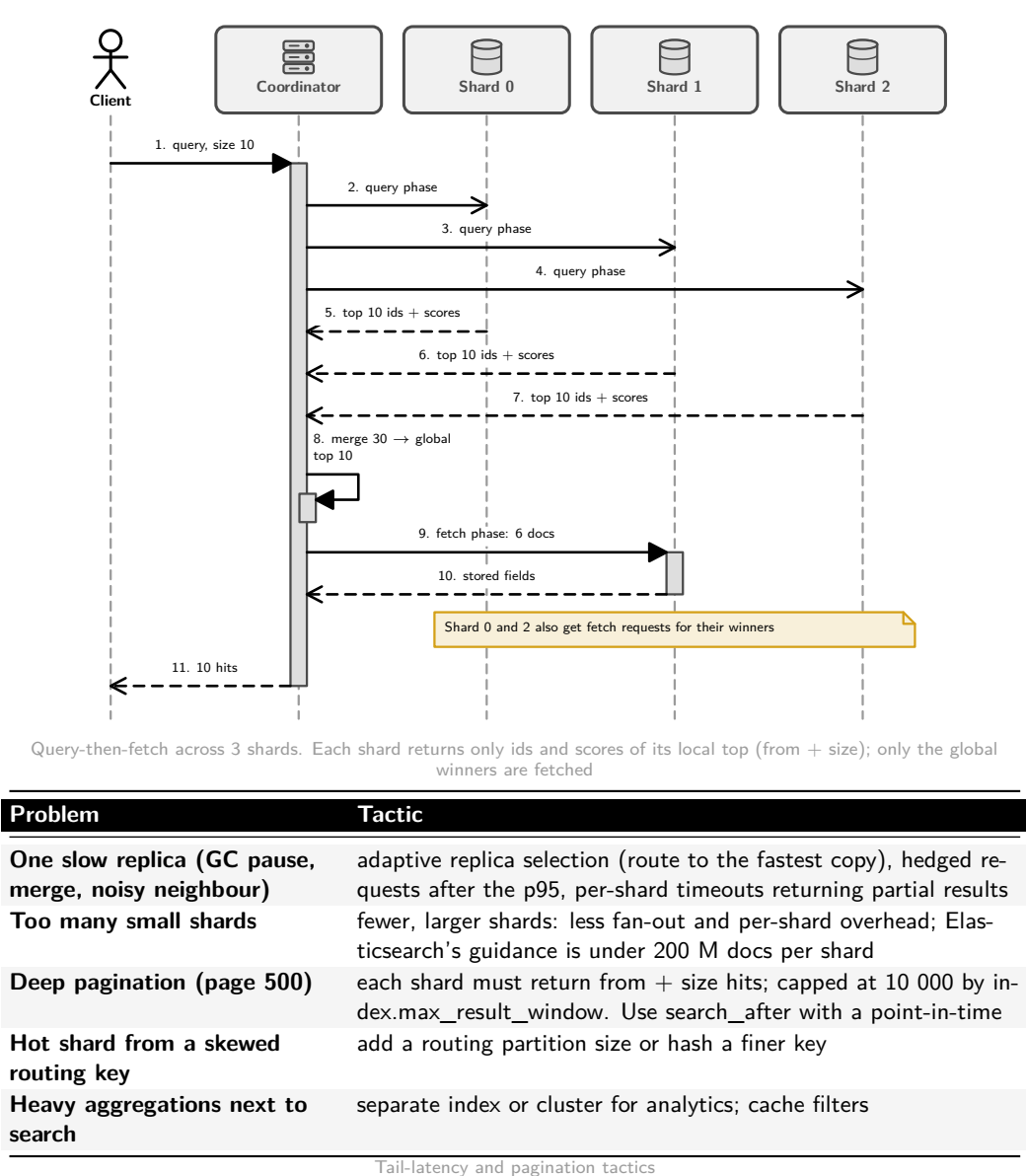
RESULT $E = 0.2603$

READ IT About 26 % of queries are slowed by a 1 % per-shard event: mitigate with hedged requests, adaptive replica selection or fewer shards

MISTAKES

- p is per request, measured at the shard's p99 latency, not the average
- Replicas do not lower N : each query still touches one copy of every shard

Tail-latency exposure of the cluster above. With 10 primaries of 120 GB it falls to about 10 %, but such shards recover slowly after a node loss: a trade-off, not a free win



	Lucene	Elasticsearch	OpenSearch	Vespa
Licence	Apache 2.0	AGPL / SSPL / ELv2	Apache 2.0 (Linux Foundation)	Apache 2.0
Form	Java library	server + cloud	server + AWS managed	server + cloud
Dense vectors	HNSW, quantised	HNSW, int8/int4/BBQ	Lucene or Faiss engine	HNSW with filters
Learned ranking inside	you code it	LTR rescorer, RRF retrievers	LTR plugin, hybrid normaliser	multi-phase ranking, ONNX on content nodes
Real-time partial updates	no (delete + add)	no (reindex doc)	no (reindex doc)	yes, attribute fields
Best at	embedding search in your app	logs + search ecosystem	open licence, AWS shops	big recs and search with ML ranking

Open-source engines (as of 2026; check current versions and licences before choosing). Elasticsearch, OpenSearch and Solr all build on Apache Lucene

Apache Solr — The older Lucene server (Apache 2.0) with an LTR module and dense vectors; strong in enterprise and library search.

OpenSearch history — Forked from Elasticsearch 7.10.2 in 2021 after Elastic's licence change; governed by the OpenSearch Software Foundation (Linux Foundation) since 2024.

Vespa phases — first-phase and second-phase ranking run on each content node next to the data, a global phase re-ranks the merged top on the container: the funnel inside the engine.

Rust and lightweight engines — Tantivy (Lucene-like library), Quickwit (logs), Meilisearch and Typesense (typo-tolerant site search) for smaller footprints.

Interview trap: 'just add more shards'

More primaries spread indexing load and storage but every query fans out to all of them, raising coordinator work and tail latency. Read throughput scales with replicas, not primaries. Answer: size primaries by data volume and write rate, replicas by QPS.

🚩 Myths, recap and self-test

MYTH OR FACT?
“A written document is immediately searchable”

MYTH: Only after the next refresh
Near-real-time: about 1 s by default in Elasticsearch; get-by-id reads are real-time.

MYTH OR FACT?
“Deleting documents frees disk at once”

MYTH: Deletes are tombstones until a merge
The live-docs bitset hides them; merges reclaim the space.

MYTH OR FACT?
“Replicas make indexing faster”

MYTH: Replicas cost indexing work
Each replica re-indexes every document; they speed up reads and add availability.

MYTH OR FACT?
“Total hit counts are always exact”

MYTH: Usually capped for speed
Exact counts disable dynamic pruning; the default stops counting at 10 000.

Search-infrastructure beliefs, checked

Tick a box only when you could explain the point to someone else

- | | |
|--|---|
| ☐ Segments are immutable
refresh adds, merge compacts, delete marks | ☐ Postings are gap-encoded blocks
small gaps, few bits |
| ☐ Primaries by data size, replicas by QPS
10-50 GB per shard | ☐ Query then fetch
ids and scores first, documents last |
| ☐ Fan-out amplifies tail latency
$1 - (1 - p)^N$ | ☐ Same analyser at index and query time
or nothing matches |

Tick when you can explain each point

Q1 Gaps of the posting list [3, 7, 21, 22, 150]?
A 3, 4, 14, 1, 128: VByte stores them in 6 bytes instead of 20.

Q2 Why are Lucene segments immutable?
A Lock-free concurrent reads, cache-friendly files, simple crash recovery; the price is merges and tombstone deletes.

Q3 Shard for a document with 5 primaries?
A $\text{hash}(\text{routing key}) \bmod 5$; changing the primary count breaks routing, hence split, shrink or reindex.

Q4 Primaries for 900 GB at 45 GB per shard?
A $\text{ceil}(900 / 45) = 20$.

Q5 Share of queries hitting a slow shard: $p = 2\%$, 20 shards?
A $1 - 0.98^{20} \approx 0.33$.

Q6 What does the query phase return?
A Per shard, the ids and scores of its local top from + size; the coordinator merges and fetches only the winners.

Q7 How to page past result 10 000?
A search_after with a point-in-time (sort on score + a tiebreaker), not from + size.

Q8 What does Block-Max WAND skip?
A Posting blocks whose maximum possible score cannot beat the current k-th best score.

Q9 Which engine runs multi-phase ML ranking on the content nodes?
A Vespa (first, second and global phases; ONNX models).