Big Data Computing

Master's of Science in Computer Science

Sapienza University of Rome

2026/2027

Gabriele Tolomei

tolomei@di.uniroma1.it

Module 3: Scaling Classical Tasks

Lecture 3.1 (Part II): Web-Scale Search — Distributed Search at Scale

Recap of Part I

In Part I we built a complete **single-machine** search engine:

- **Inverted index:** a dictionary plus sorted posting lists, built by sort-based construction (MapReduce-friendly).
- **Compression:** gap encoding + variable-byte shrink postings by 4x–8x.
- **Query processing:** sorted-list merges for Boolean queries, and **DAAT** scoring with a **min-heap of size** k for ranked top-k retrieval.

Open problem: the full-web index is terabytes and must answer in milliseconds. One machine is not enough.

Where We Are: Distributed Search

Part II scales the index of Part I across a cluster, in **4 sections**:

6. **Document Partitioning** — local indexing & query fan-out
7. **Term Partitioning** — vocabulary sharding & network trade-offs
8. **Routing & Scale-Out Architecture** — caching, replication & graph authority
9. **Synthesis** — end-to-end architecture & cross-module callbacks

> *Extended material (sharding, tail latency, bandwidth arithmetic, caching economics, rank fusion, etc.) is in the **Appendix** (A.6–A.9).*

§ 6 — Distributed Partitioning: Documents

One distributed strategy

📎 *Extended material → Appendix A.6*

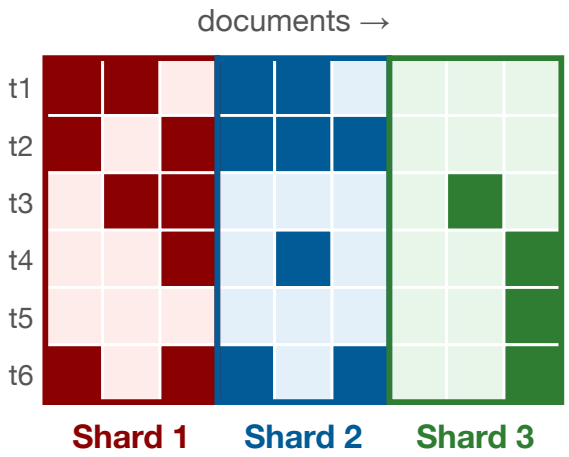
The Scale-Out Imperative

Even with 8× compression, an index of 10^{10} documents needs multiple terabytes of memory, and thousands of concurrent queries demand high throughput:

- Single-machine indexing is impossible $\implies$ the index must be **partitioned (sharded)** across a cluster.
- Parallels graph partitioning from Lecture 2.4, §4: how do we split a bipartite graph (terms $\leftrightarrow$ documents) across nodes?
- Two paradigms: **partition by document** or **partition by term**.

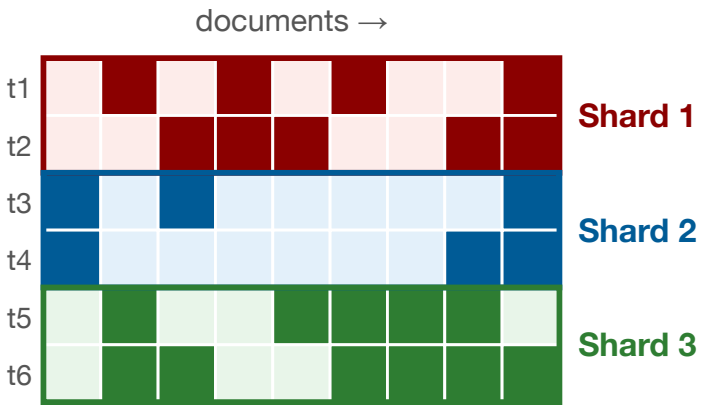
Two Ways to Slice the Index

Document partitioning



each shard: full index of its docs

Term partitioning

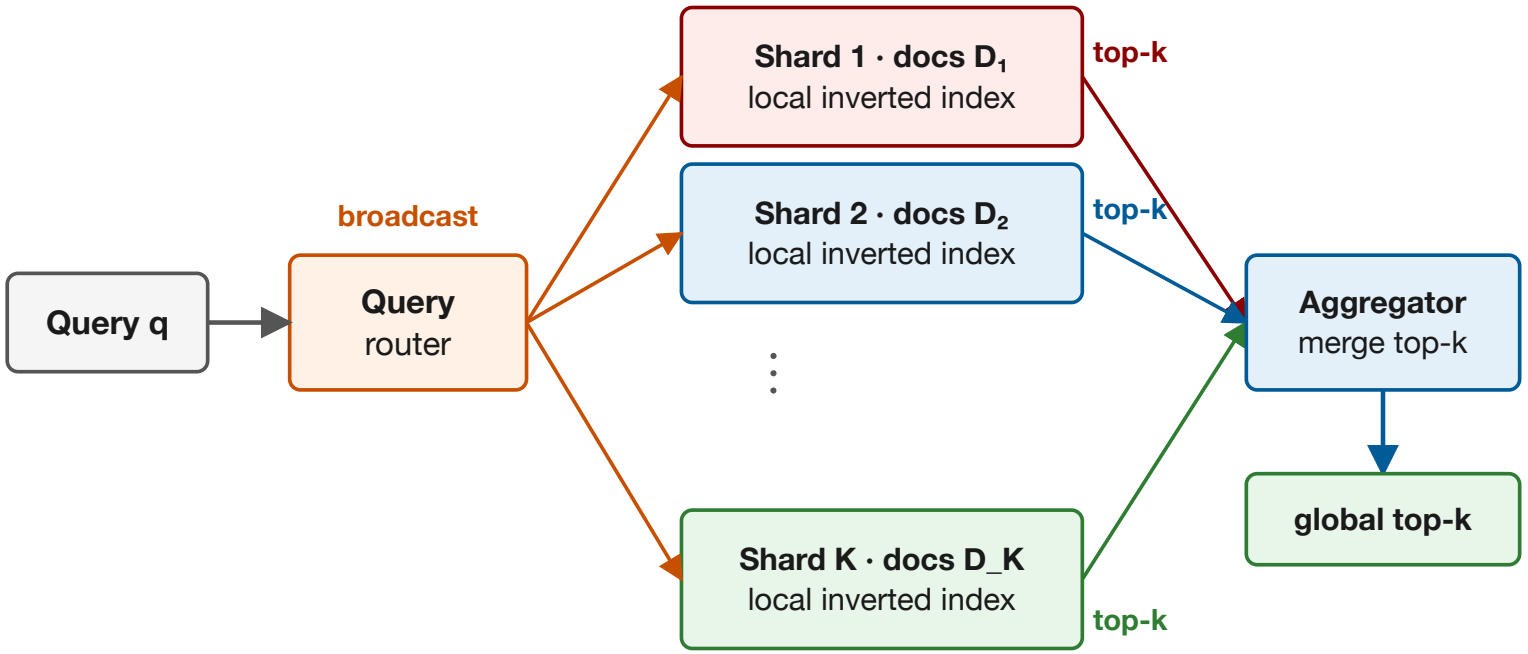


each shard: postings of its terms

Same term–document matrix, two cuts: by columns (documents) or by rows (terms).

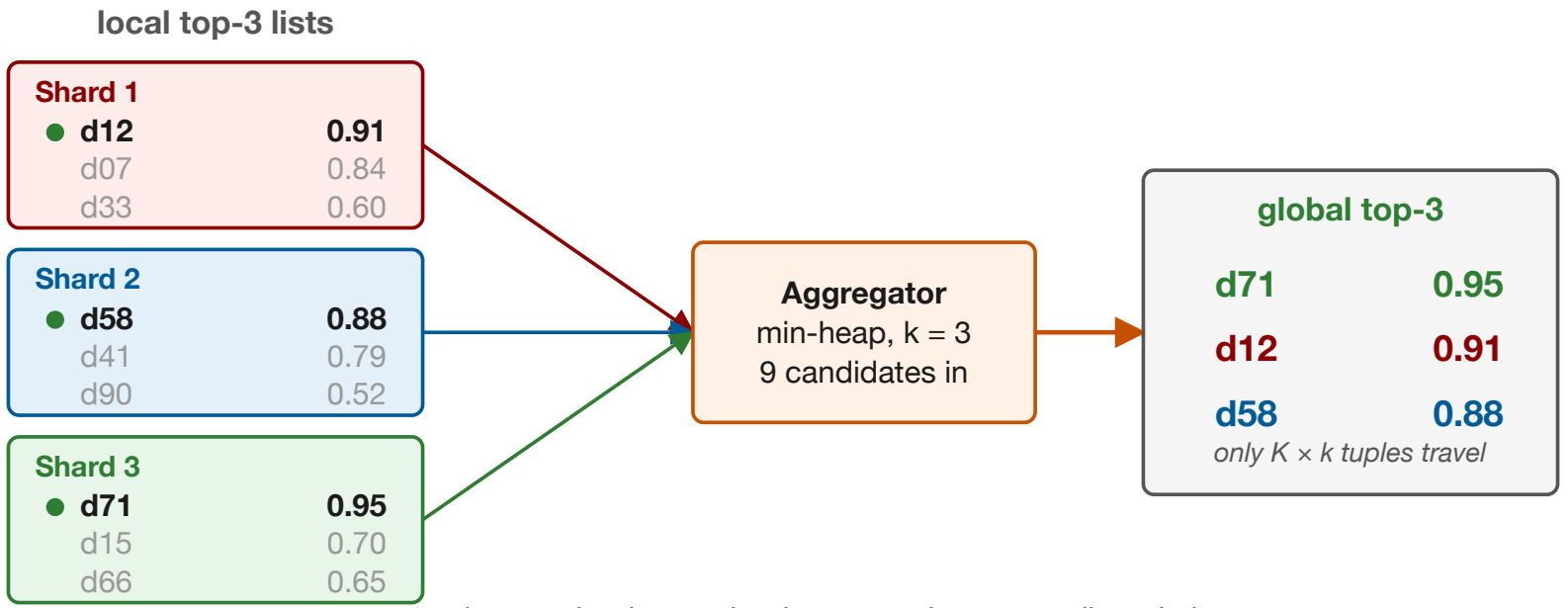
Document Partitioning Architecture

Split corpus D into K disjoint subsets; each node holds a **local inverted index** of its slice.



- Each shard runs local TAAT/DAAT and returns its local top-k to an aggregator.

Query Routing & Result Aggregation



green dots: entries that survive the merge; the rest are discarded

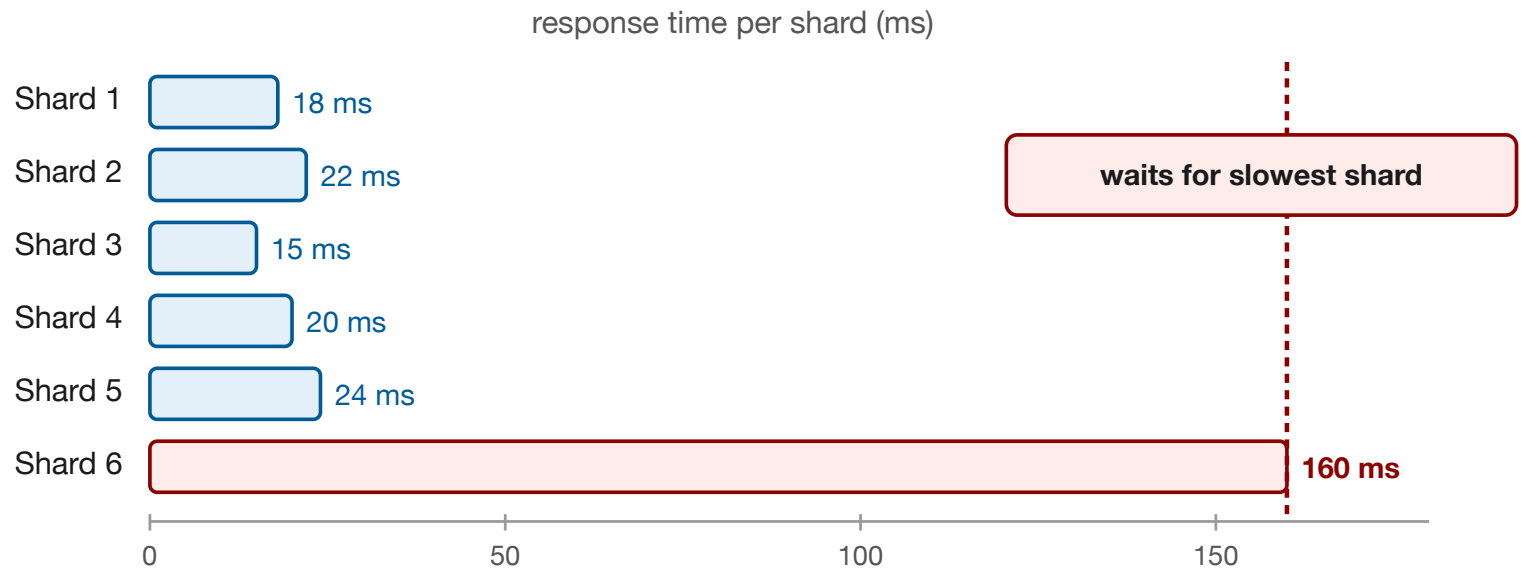
- The engine cannot know which shard holds the best documents $\Longrightarrow$ **broadcast** q to all K shards.
- Aggregator merges $K \times k$ (docID, score) tuples with a min-heap: tiny network payload!

Strengths of Document Partitioning

- **Zero inter-node communication during scoring:** each shard ranks its documents independently.
- **Trivial horizontal scalability:** new documents $\Longrightarrow$ new shards; existing shards stay untouched.
- **Excellent fault tolerance:** a shard failure hides only $1/K$ of the documents.

Price to pay: every query is broadcast to all K shards (cluster work $= K\times$ cost per shard).

Weakness: The Straggler Problem



- Query latency is set by the **slowest** shard: $T_{\text{query}} = \max_i T_i$.
- The more shards, the likelier one is slow. *Why? Derivation in Appendix A.6.3.*

✅ Checkpoint 6 — Document Partitioning

Test your understanding:

1. Define document partitioning, and describe how the corpus is split across machines.
2. Why must every machine be queried under document partitioning, regardless of query selectivity?
3. List one strength and one weakness of document partitioning.
4. How does the per-query fan-out cost scale with the number of machines?

Consolidate your answers before proceeding.

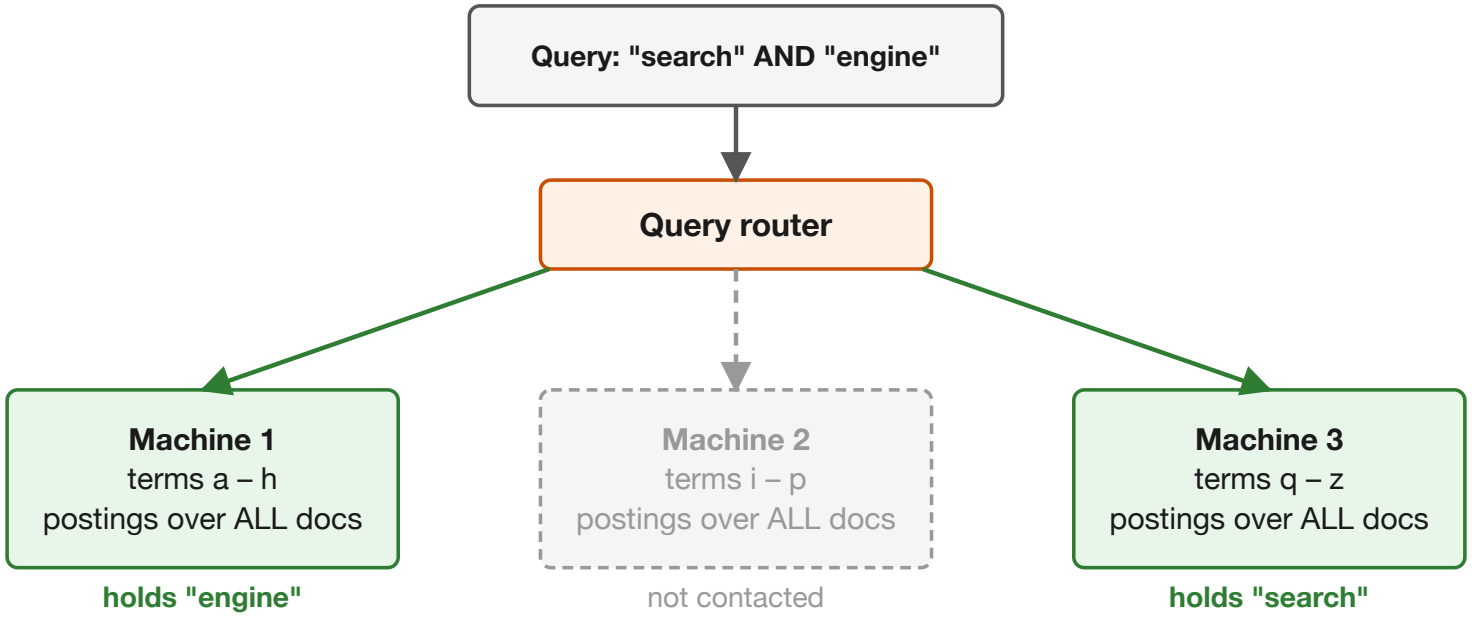
§ 7 — Distributed Partitioning: Terms

The alternative distributed strategy

📎 *Extended material → Appendix A.7*

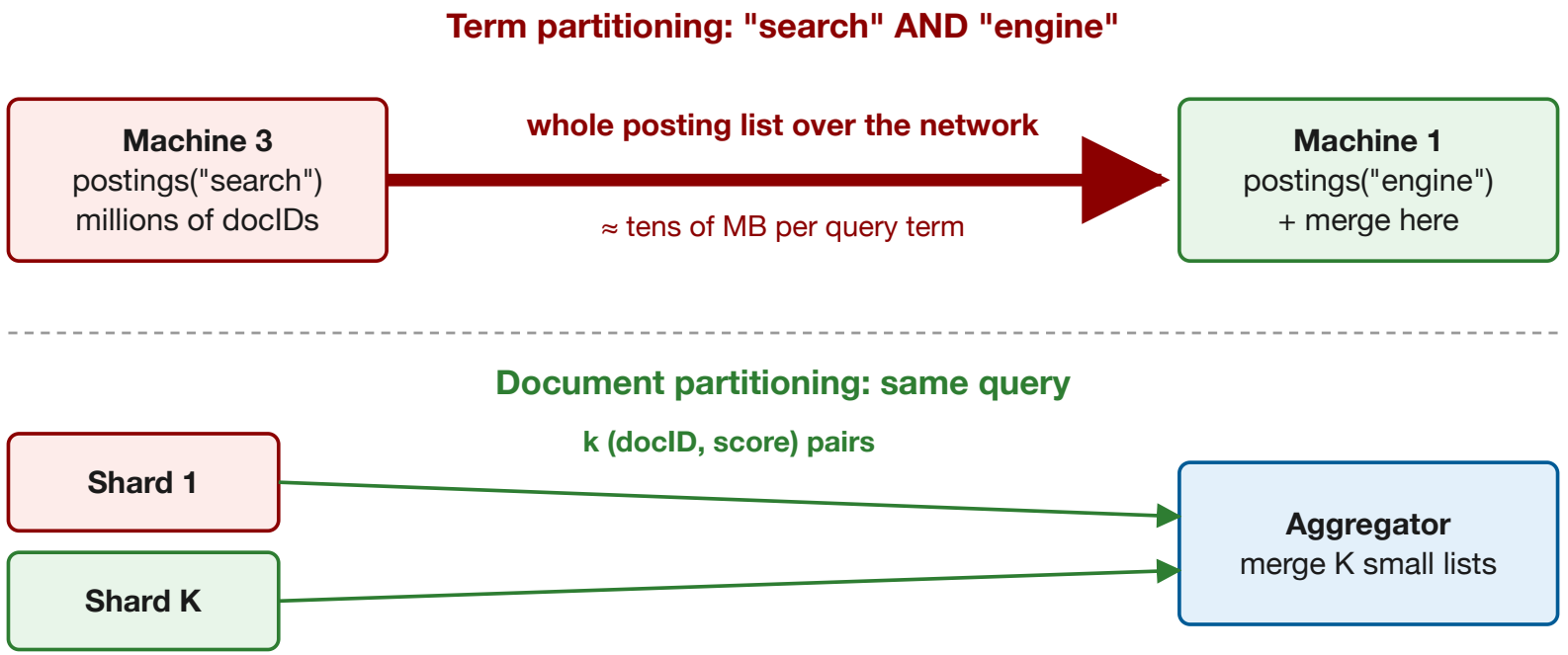
Term Partitioning Architecture

Instead of slicing documents, split the vocabulary V across machines: each holds the **global** posting lists of its terms.



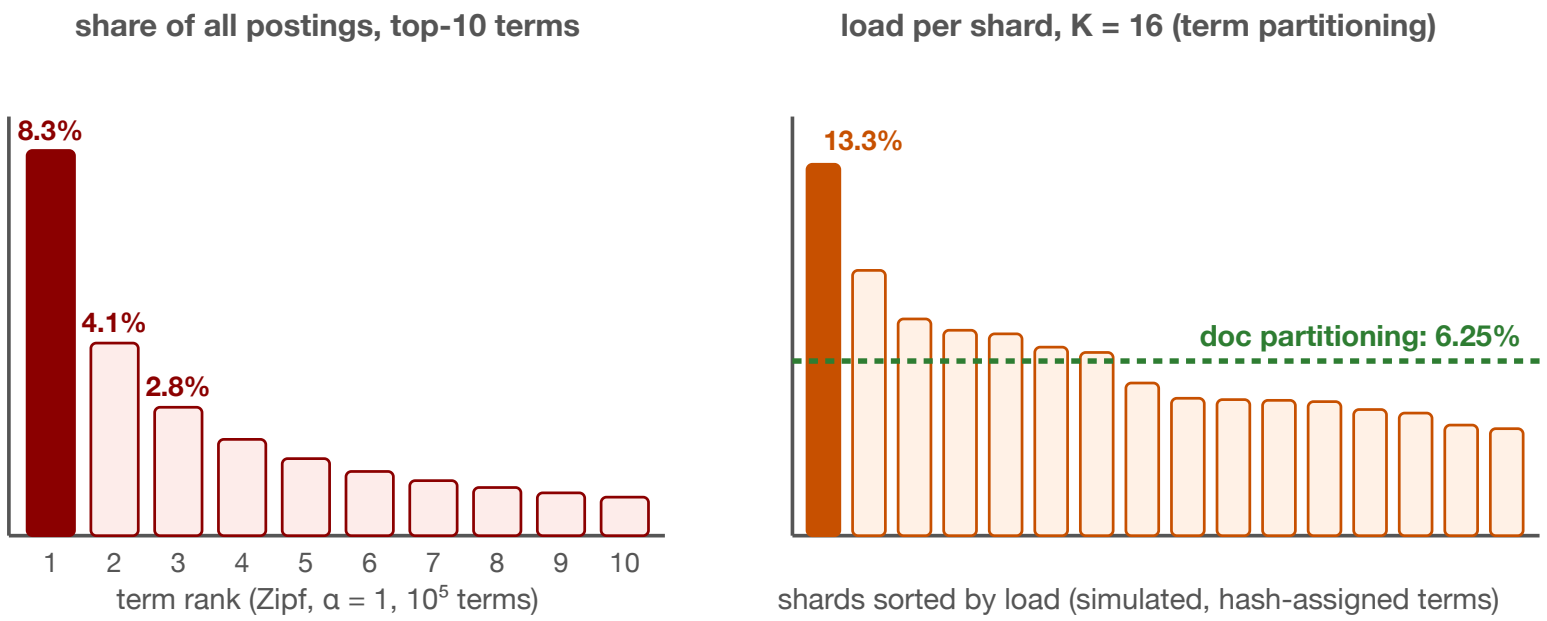
- Apparent advantage: no broadcast, only machines holding query terms are contacted.

The Fatal Flaw: Network Transfer on Intersection



- Multi-term queries need posting lists combined on one machine: millions of docIDs (megabytes) cross the network.
- This destroys the $\leq 200\,\mathrm{ms}$ latency budget. *Numbers: Appendix A.7.2.*

Term Partitioning: The Power-Law Imbalance



- **Zipf's Law:** a few terms (`"web"`, `"news"`) dominate postings and queries $\Longrightarrow$ their nodes become bottlenecks, others sit idle.
- Document partitioning hashes documents $\Longrightarrow$ near-perfect balance. Mirrors graph hub skew (Lecture 2.4, §4).

Comparing Partitioning Paradigms

	Document partitioning	Term partitioning
Partitioning key	doc ID range / hash	term range / hash
Query routing	broadcast to all K shards	only the |q| term shards
Inter-node traffic	small top-k lists	full posting lists
Load balancing	uniform	Zipf-skewed
Fault isolation	lose 1/K of the docs	lose queries with that term
In practice	standard in web search	specialized systems

*Network bandwidth is far scarcer than local CPU/memory bandwidth, so web-scale engines almost universally adopt **Document Partitioning**.*

✅ Checkpoint 7 — Term Partitioning

Test your understanding:

1. Define term partitioning, and contrast it structurally with document partitioning.
2. What hidden network cost does term partitioning incur when processing multi-term Boolean queries?
3. Why is load balancing especially difficult under term partitioning?
4. Why do most large-scale web search engines favor document partitioning in practice?

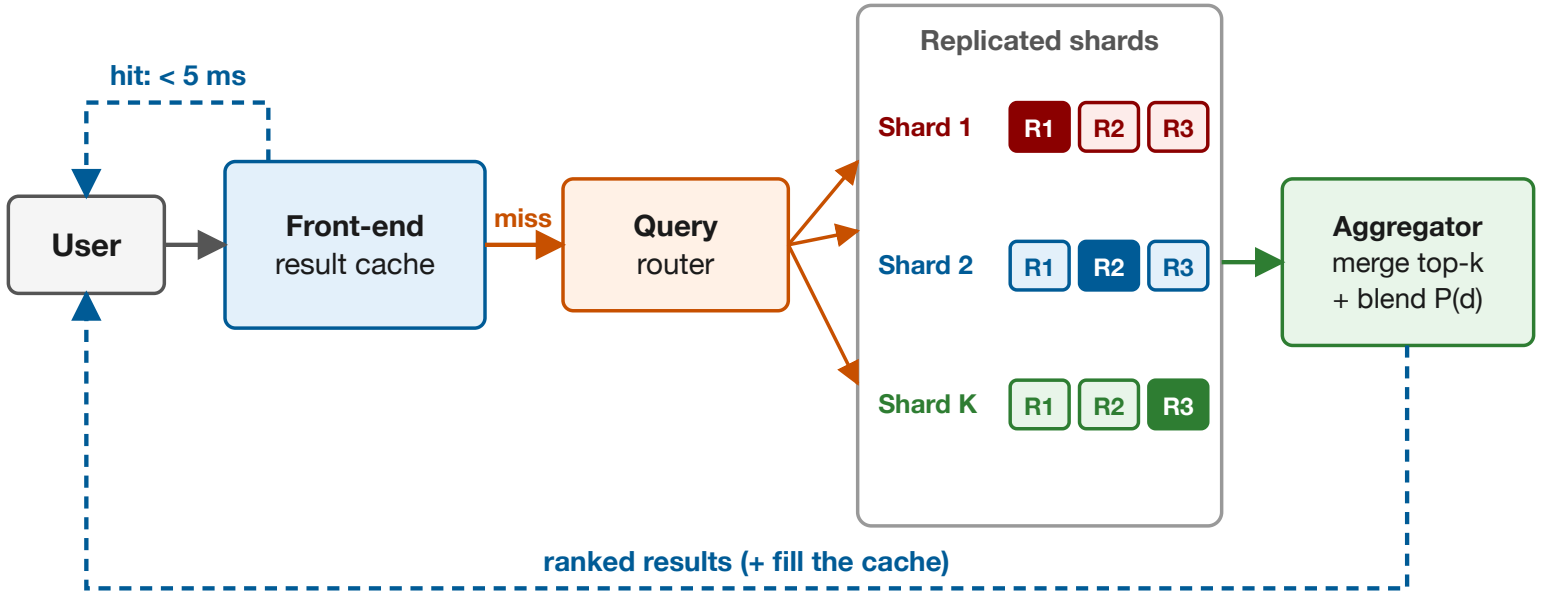
Consolidate your answers before proceeding.

§ 8 — Document Routing & Scale-Out Architecture

Assembling a working system

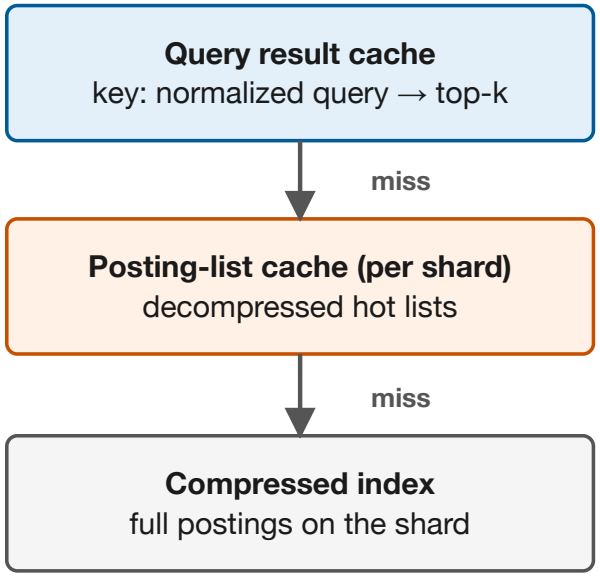
📎 *Extended material → Appendix A.8*

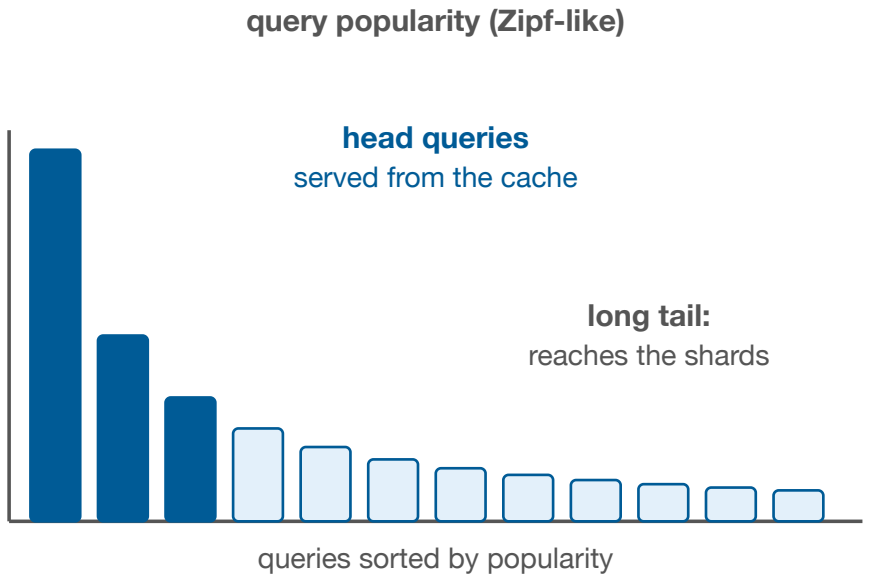
End-to-End Scale-Out Architecture



- Each shard lives in a replication group (load balancing and failover); caches intercept popular queries.

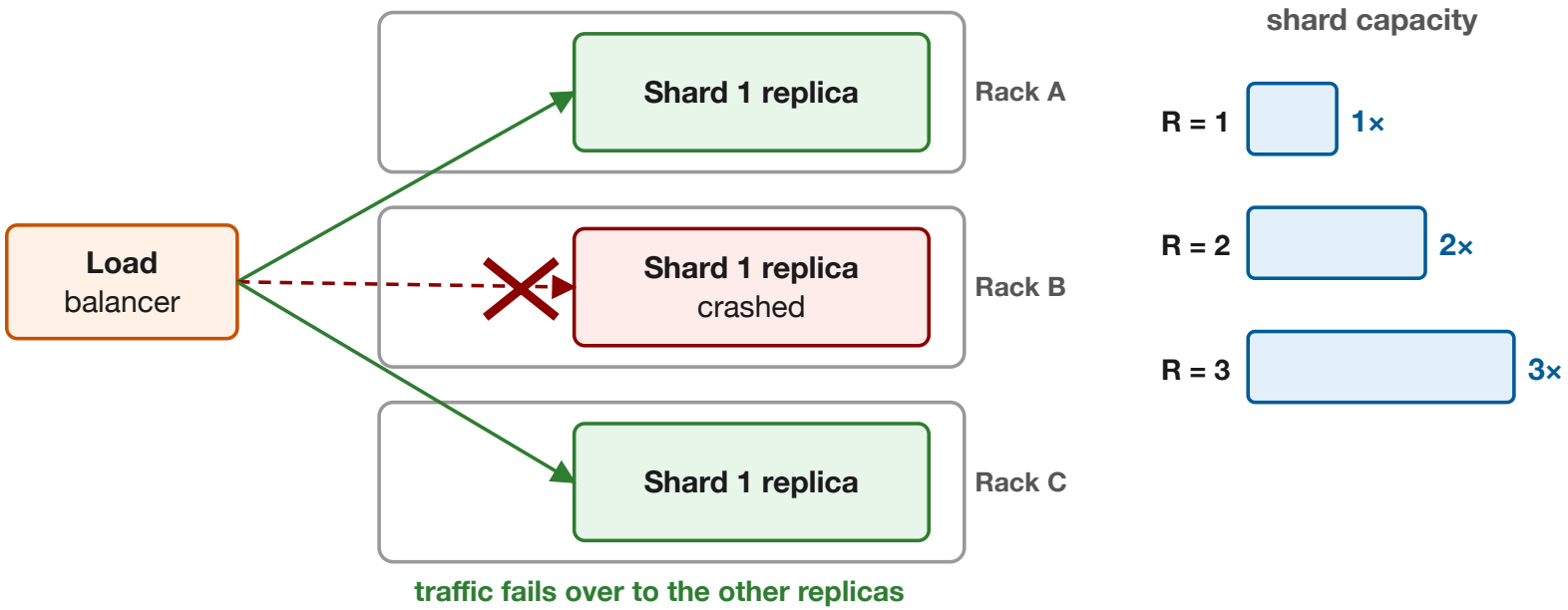
Caching: Exploiting Skew





- Heavily skewed traffic (breaking news, navigational queries) makes caches very effective.
- **Query result cache** serves 30–50% of traffic in $< 5\,\mathrm{ms}$; **posting list cache** keeps hot lists in RAM.

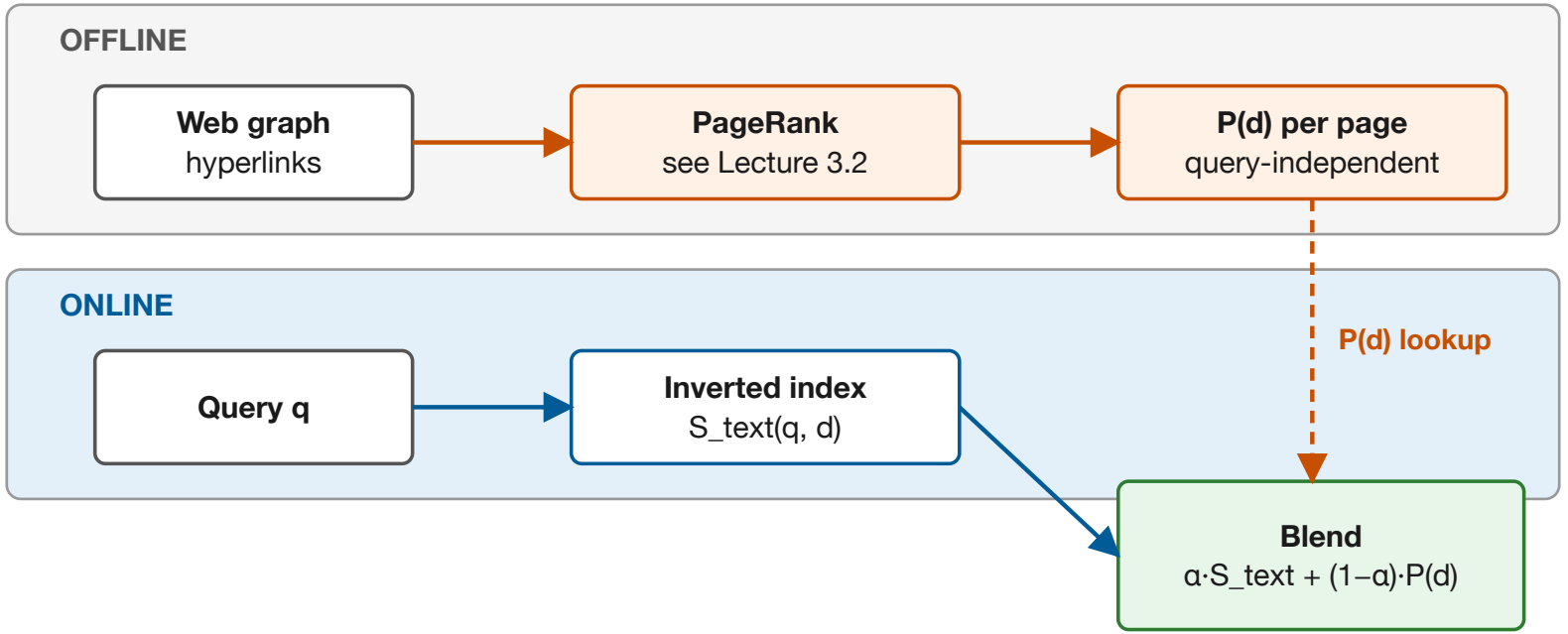
Replication: Throughput and Availability



- Every shard is copied R times across racks: throughput $\approx R\times$ shard capacity.
- If a server crashes, traffic fails over to another replica with no downtime.

Echoes GFS/HDFS chunk replication (Lecture 1.1), now at the index-serving layer.

Blending Content Relevance with Graph Authority



- **Offline (Lecture 3.2):** PageRank gives a query-independent authority prior $P(d)$.
- **Online (§5):** the inverted index gives $S_{\text{text}}(q, d)$; the two are blended: $\alpha S_{\text{text}} + (1 - \alpha)P(d)$.

Heavy graph analytics run offline; low-latency scoring runs per query.

✅ Checkpoint 8 — Document Routing & Scale-Out Architecture

Test your understanding:

1. Describe the end-to-end query path, from query router through to the result aggregator.
2. Why must the result aggregator merge, rather than simply concatenate, the local top-k lists from each shard?
3. How does query result caching exploit query-workload skew?
4. Explain why replication in this architecture serves both a load-balancing and a fault-tolerance purpose.

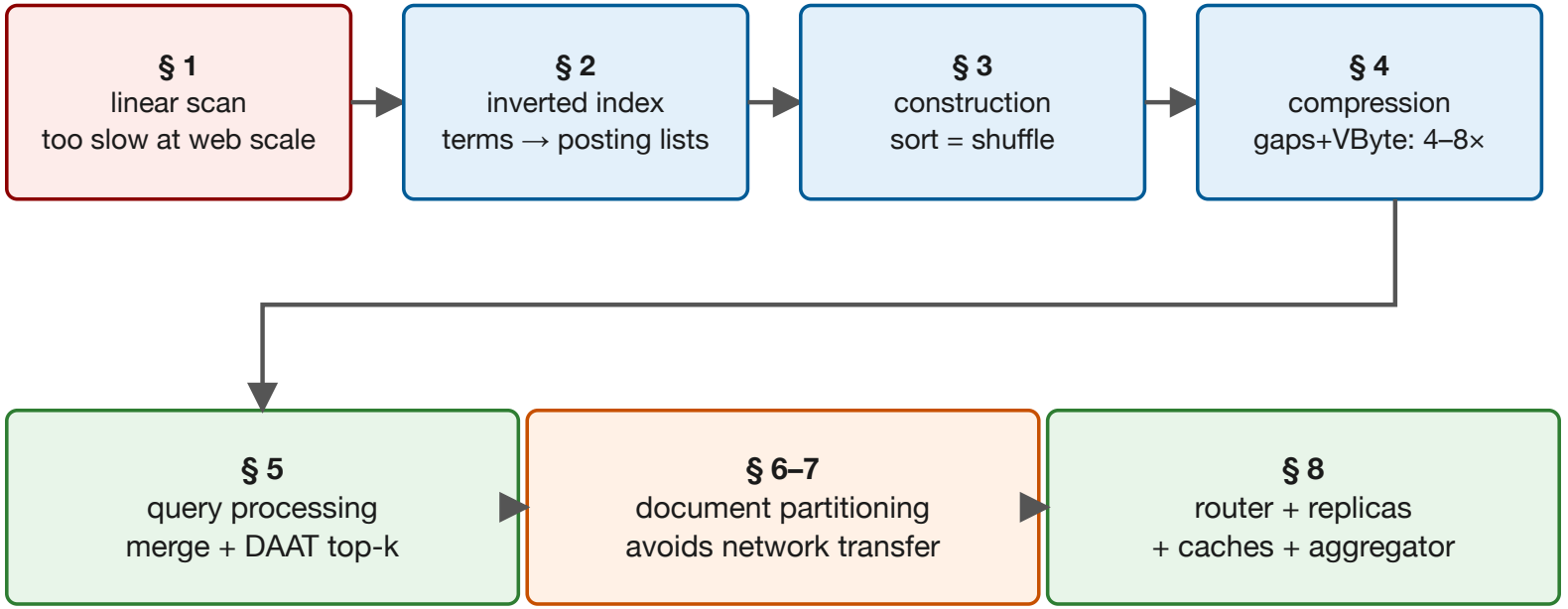
Consolidate your answers before the final synthesis.

§ 9 — Synthesis & Wrap-up

The full pipeline, end to end

📎 *Extended material → Appendix A.9*

The Chain of Results Recap



Summary of Key Takeaways

1. **Inverted Index is King:** Replaces $O(N)$ document scans with $O(1)$ dictionary lookups and efficient posting list traversals.
2. **Sorting is the Computational Core:** Building indices at scale is fundamentally a distributed sorting problem, cleanly solved by MapReduce.
3. **Sortedness Enables Compression & Merging:**
    - Sorted docIDs enable delta gap encoding (sub-10-bit postings).
    - Sorted docIDs enable $O(n_1 + n_2)$ linear Boolean intersections.
4. **DAAT Prevents Memory Explosions:** Evaluating documents concurrently in docID order bounds memory to $O(|q| + k)$ via min-heaps.
5. **Document Partitioning Triumphs at Web Scale:** Paying the broadcast fan-out penalty is far cheaper than streaming multi-megabyte posting lists across network switches.

Key Terms Glossary

Term	Meaning
Inverted index	Term → document-list mapping; dictionary + posting lists
Posting list	Sorted list of documents (and metadata) containing a term
Gap encoding	Storing differences between consecutive sorted docIDs
Variable-byte	Byte-aligned code: 7 payload bits + 1 continuation bit
Skip pointers	Shortcuts accelerating sorted posting-list intersection
TAAT / DAAT	Term-at-a-time (accumulators) vs. document-at-a-time (lockstep + heap) scoring
Document partitioning	Corpus split across machines; each holds a full local index
Term partitioning	Vocabulary split across machines; each holds full postings of its terms
Query fan-out	Broadcasting a query to all shards under document partitioning

Lecture 3.1 Complete

Recommended Reading

- Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*, Cambridge University Press. Chapters 1–4, 6, and 20.
- Dean, J., & Ghemawat, S. (2004). *MapReduce: Simplified Data Processing on Large Clusters*. OSDI.

Contact & Q&A

- **Course:** Big Data Computing (2026/2027)
- **Lecturer:** Gabriele Tolomei — tolomei@di.uniroma1.it
- **Office Hours:** Drop me a message, and we'll arrange a time to meet, either in person or online! 🙂

Thank you for your attention!

§ Appendix

Appendix Map

Section	Extends	What you will find	Slides
A.6	§ 6	Sharding functions (with figure), query-execution diagram, tail-latency amplification (with curve)	5
A.7	§ 7	Routing mechanics, network bandwidth arithmetic (with latency-budget figure), Zipfian load skew	4
A.8	§ 8	End-to-end query life cycle, caching/replication economics, rank fusion with PageRank	3
A.9	§ 9	Worked problems: compression, skip pointers, stragglers	2

Supplementary material for deeper study. Not required during the main lecture.

📎 A.6 — Sharding & Tail Latency

A.6.1 · Partitioning Fundamentals and Sharding Functions

Let cluster consist of K physical indexing nodes (shards). Total corpus D (N documents) must be distributed across nodes:

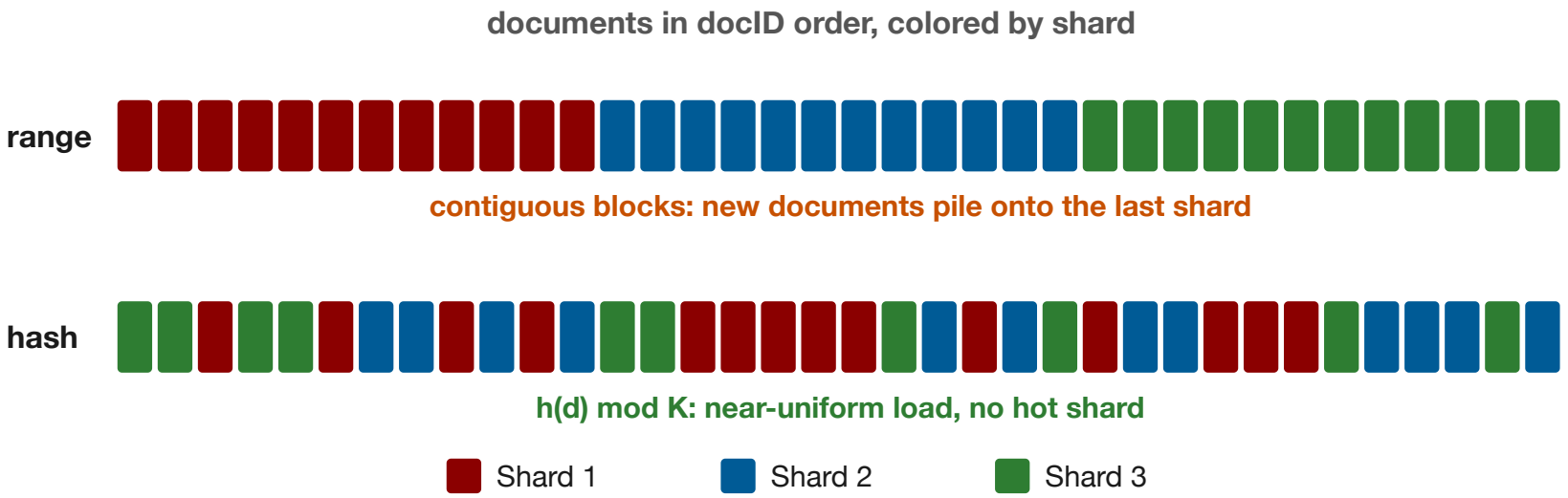
Document Assignment Functions:

- **Range Partitioning:** shard k stores docIDs $\left[\frac{(k-1)N}{K} + 1, \ldots, \frac{kN}{K}\right]$. Vulnerable to data ingestion imbalance.
- **Hash Partitioning:** document d is assigned to shard $h(d) \pmod K$.
    - Generates uniform document distribution across shards with high probability (by Chernoff bounds, Lecture 2.1).
    - Normalizes posting list length variance across shards.

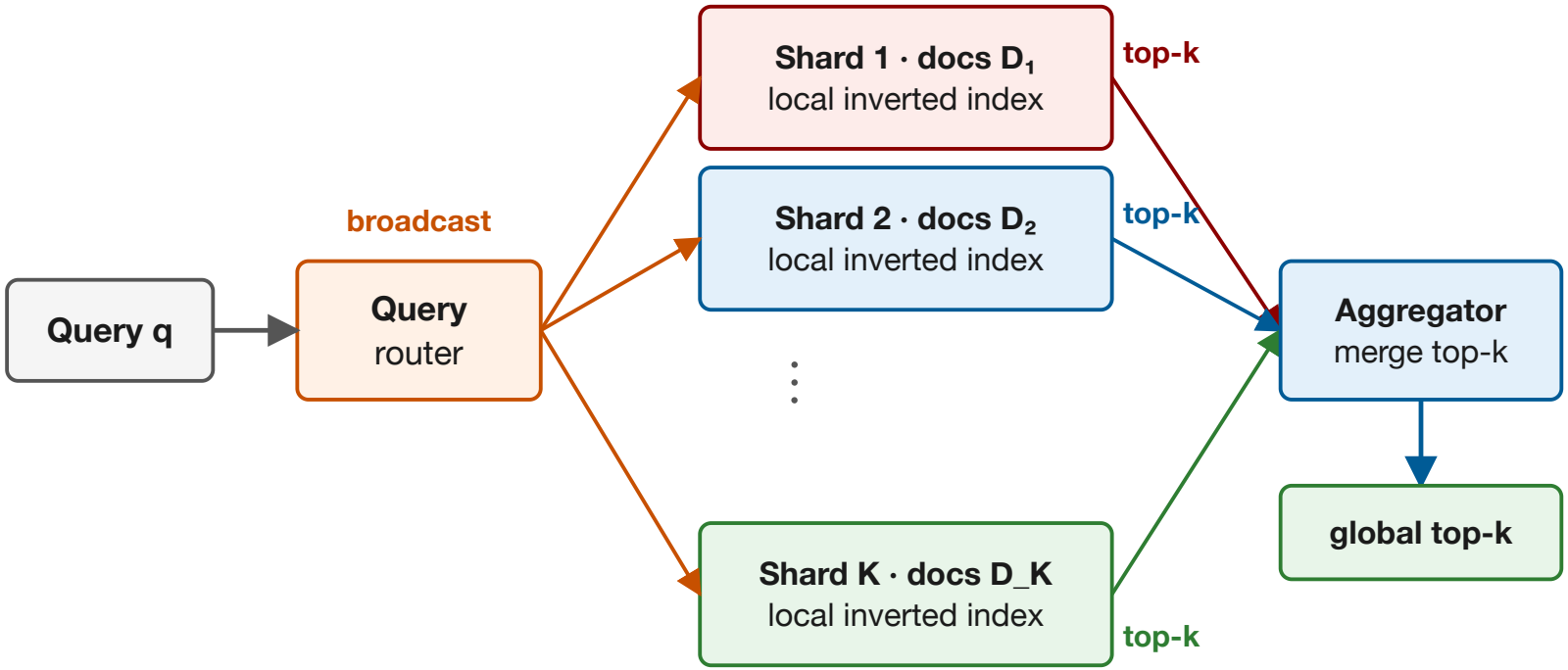
Local Index Independence:

- Each shard S_k builds its own internal dictionary V_k and compressed posting lists for documents D_k.
- Shard S_k requires no external knowledge of the existence or contents of other shards during index creation.

A.6.1 · Sharding Functions at a Glance



A.6.2 · Document Partitioning: Query Execution



A.6.3 · Tail Latency Amplification

The Straggler Bottleneck (Tail Latency Amplification):

- Let T_i be the response time of shard i. Query response time is:

$$T_{\text{query}} = \max(T_1, T_2, \ldots, T_K) + T_{\text{aggregation}}$$

- If each shard has independent probability $p = 0.01$ of experiencing a 1-second latency spike (due to garbage collection, background compaction, or disk contention), the probability that the overall query experiences a spike is:

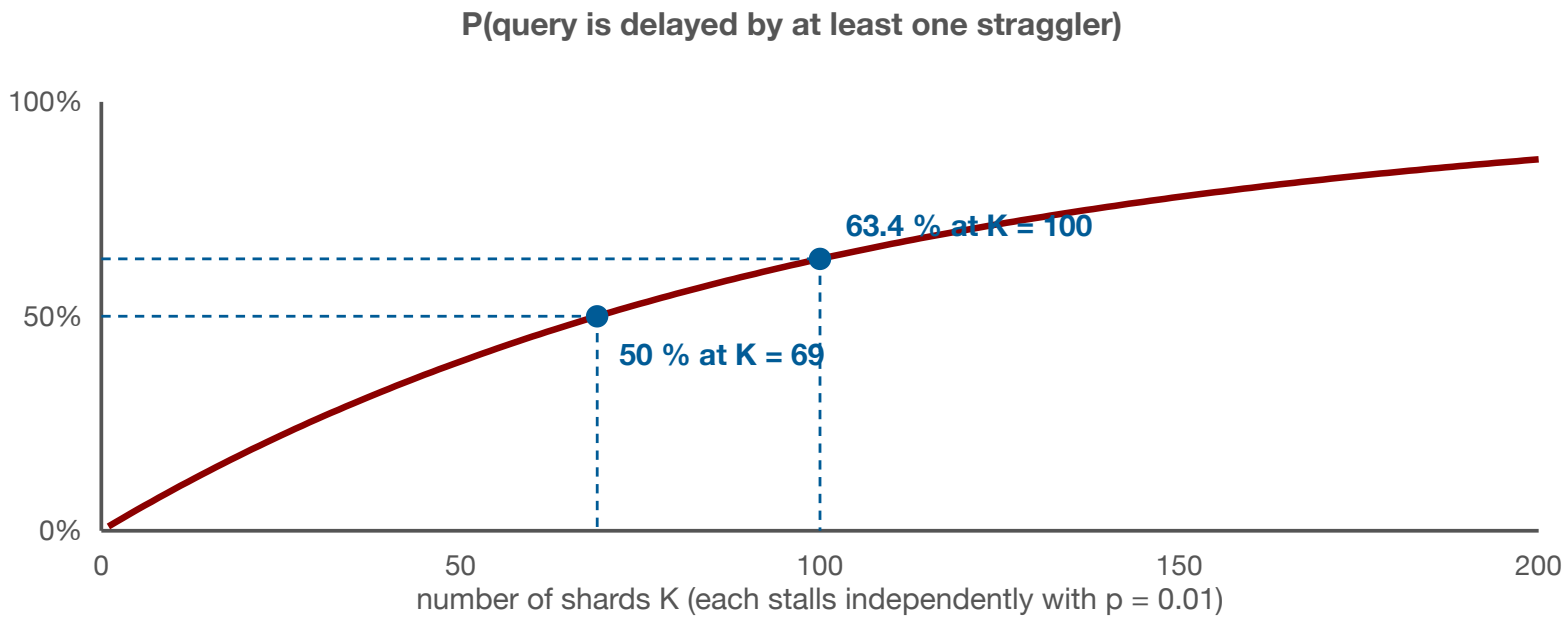
$$P(\text{Spike}) = 1 - (1 - p)^K$$

- For $K = 100$ shards:

$$P(\text{Spike}) = 1 - (0.99)^{100} = 1 - 0.366 = \mathbf{63.4\%}$$

- *Result:* More than 63% of user queries would be delayed by stragglers!
- Systems mitigations: hedge requests, aggressive replication (§8), and tie-breaker timeouts.

A.6.3 · Tail Latency Amplification: The Curve



Even a 1% per-shard spike rate hits half of all queries at $K = 69$.

📎 A.7 — Term Partitioning: Network & Skew

A.7.1 · Term Partitioning: Routing Mechanics

In term partitioning, vocabulary V is partitioned into K disjoint subsets $V_1, \ldots, V_K$:

- Shard k holds posting lists $P(t)$ for all $t \in V_k$.
- Every posting list in shard k covers the **entire corpus** D ($N = 10^{10}$ documents).

Single-Term Query:

- Query $q = (t_1)$: Router consults vocabulary map, contacts only the single shard holding t_1. Zero broadcast fan-out! Zero network merge cost!

Multi-Term Query Execution Bottleneck:

- Query $q = (t_1 \text{ AND } t_2)$ where $t_1 \in V_1$ and $t_2 \in V_2$.
- The search engine must execute one of the following:
    i. Transfer entire $P(t_1)$ from Shard 1 to Shard 2.
    ii. Transfer entire $P(t_2)$ from Shard 2 to Shard 1.
    iii. Stream both $P(t_1)$ and $P(t_2)$ to the Query Router and merge there.

A.7.2 · Network Bandwidth Analysis

Quantifying Network Transfer Costs:

- Let query be `"distributed" \text{ AND } "systems"`.
- Suppose $\mathrm{df}(\text{"distributed"}) = 5 \times 10^7$ postings; $\mathrm{df}(\text{"systems"}) = 10^8$ postings.
- Even compressed with Variable-Byte ($1.5\,\mathrm{bytes/posting}$):

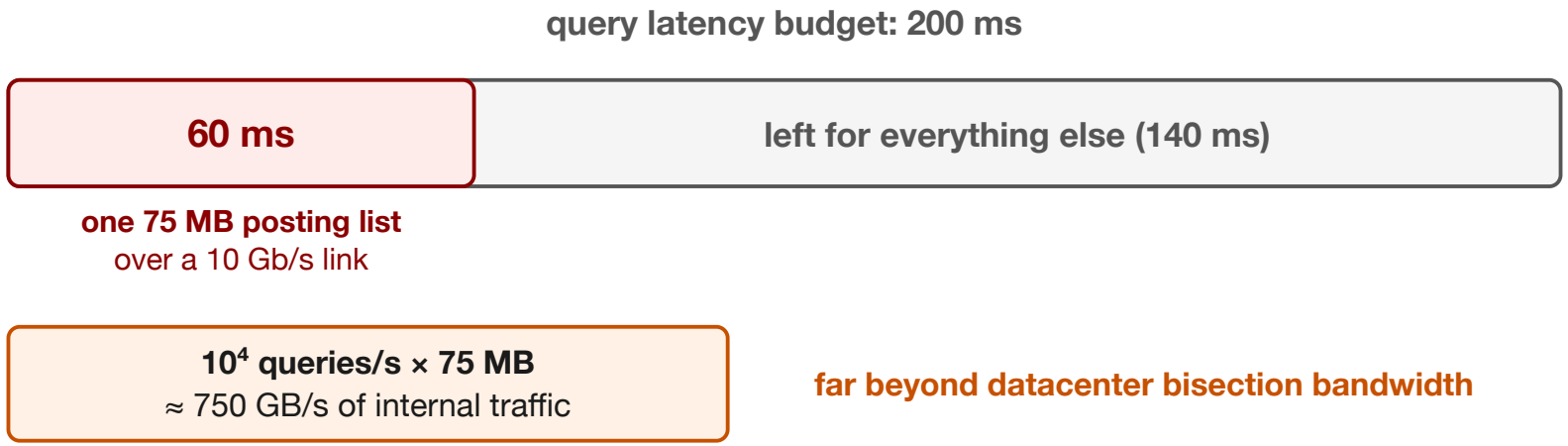
$$\mathrm{Size}(P(\text{"distributed"})) \approx 5 \times 10^7 \times 1.5\,\mathrm{B} = 75\,\mathrm{MB}$$

- Transferring $75\,\mathrm{MB}$ across a 10 Gbps datacenter link takes:

$$T_{\text{transfer}} = \frac{75 \times 8 \times 10^6\,\text{bits}}{10 \times 10^9\,\text{bits/s}} = 60\,\mathrm{ms}$$

- Just transferring *one* posting list burns 30% of the entire $200\,\mathrm{ms}$ query latency budget!
- Under $10,000\,\mathrm{qps}$, aggregate internal switch traffic would require $750\,\mathrm{GB/s}$ — far exceeding datacenter network bisection capacity.

A.7.2 · The 200 ms Latency Budget



A.7.3 · Zipfian Skew and Load Imbalance

Zipf's Law and Node Load Imbalance:

- Term frequency follows a power-law distribution:

$$f(r) \propto \frac{1}{r^{\alpha}}, \quad \alpha \approx 1$$

where r is the frequency rank of the term.

- Top 1% of terms account for $> 50\%$ of all token occurrences in the corpus.
- The shards assigned popular terms (e.g., `"web"`, `"search"`, `"news"`) receive massive query load and storage burdens, while shards holding rare terms remain almost idle.
- Parallels graph hub imbalances from Lecture 2.4, §4.

📎 A.8 — Query Life Cycle, Caching & Rank Fusion

A.8.1 · End-to-End Query Life Cycle

```
1. User types query: "big data computing"
2. Front-end Web Server checks In-Memory Query Result Cache (LRU).
   └── If hit: return precomputed HTML results (< 5 ms).
3. On cache miss: send query to Query Router.
4. Query Router consults cluster load balancer; broadcasts query to 1 replica
   of each of the K document shards.
5. Each Shard executes local DAAT query processing using in-memory compressed postings:
   - Traverses posting lists for "big", "data", "computing".
   - Maintains min-heap of local top-k documents (e.g., k = 100).
   - Returns local top-k (docID, local_score) to Result Aggregator.
6. Result Aggregator receives K local top-k lists (K × 100 entries):
   - Merges into global top-100 list using priority queue.
   - Blends scores with precomputed static PageRank priors.
7. Snippet Generation Engine fetches titles and text snippets for the top 10 docs.
8. Front-end updates Result Cache and renders response to user.
```

A.8.2 · Multi-Tier Caching and Replication Economics

Multi-Tier Caching Strategy:

1. **Tier 1 — Query Result Cache:**
    - Keys: normalized query string. Values: top-k document IDs and snippets.
    - Hit rate: ≈ 30–50% due to heavy query workload skew (Zipfian query popularity).
2. **Tier 2 — Posting List Cache:**
    - Caches decompressed posting lists of frequent terms in RAM on individual shards.
    - Avoids repetitive delta/VB decompression on popular terms.

Replication Economics:

- Shard is replicated across R physical machines (typically $R \geq 3$).
- **Throughput Scaling:** aggregate query capacity scales linearly with R:

$$Q_{\max} \propto R \times (\text{capacity of 1 shard})$$

- **High Availability:** System survives failure of up to $R - 1$ replicas without data loss.

A.8.3 · Ranking Fusion with Graph Authority

Ranking Fusion (Marrying Content Relevance & Authority):

$$\text{FinalScore}(q, d) = \alpha \cdot S_{\text{text}}(q, d) + (1 - \alpha) \cdot \log(\text{PageRank}(d) + \epsilon)$$

- $S_{\text{text}}(q, d)$: Dynamic query-dependent similarity computed online via cosine/BM25 over the inverted index (§5).
- $\text{PageRank}(d)$: Static query-independent structural authority precomputed offline via distributed graph algorithms (Lecture 3.2).
- Logarithmic scaling ensures PageRank powers (which vary across orders of magnitude) do not overwhelm term matching.

📎 A.9 — Worked Problems

A.9.1 · Problem 1: Gap + Variable-Byte Encoding

Compress $\langle 100005, 100009, 100010, 100022, 100023 \rangle$ with VB, and compare with 64-bit docIDs.

```
Gaps:        100005   4    1    12    1
VB bytes:       3     1    1     1    1      (7-bit chunks: <128 -> 1 B,
                                               <16384 -> 2 B, <2^21 -> 3 B)
Total: 7 bytes  vs  5 x 8 = 40 bytes   ->  about 5.7x smaller
```

Warm-up: $\langle 5, 6, 15, 16, 17 \rangle$ gives gaps $5, 1, 9, 1, 1$, all < 128, hence 5 bytes vs. 40 bytes, i.e. $8\times$.

A.9.2 · Problem 2: Skip Pointers and Stragglers

(a) A posting list has $L = 10^6$ entries. Optimal skip interval and worst-case cost?

$$S^* = \sqrt{L} = 10^3, \qquad C(S^*) = \frac{L}{S^*} + S^* = 2 \times 10^3 \ll 10^6$$

(b) Each shard independently stalls with probability $p = 0.01$. How many shards K before more than half of the queries hit a stall?

$$1 - (1-p)^K > \tfrac{1}{2} \iff K > \frac{\ln 0.5}{\ln 0.99} \approx 69 \Rightarrow K \geq 69$$

At $K = 100$: $1 - 0.99^{100} \approx 63.4\%$, which motivates replication and hedged requests (§ 8).