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 I): Web-Scale Search — The Inverted Index and Query Processing

Philosophy of Scalable Search

> *"The value of an idea lies in the using of it."*
> *— Thomas Edison*

*Module 2 taught us how to represent data — from high-dimensional geometry to text and graphs — as vectors and structures a machine can operate on. Module 3 now asks the harder question: once billions of documents are represented, how do we actually **find** anything, in milliseconds, at planetary scale?*

Lecture Roadmap

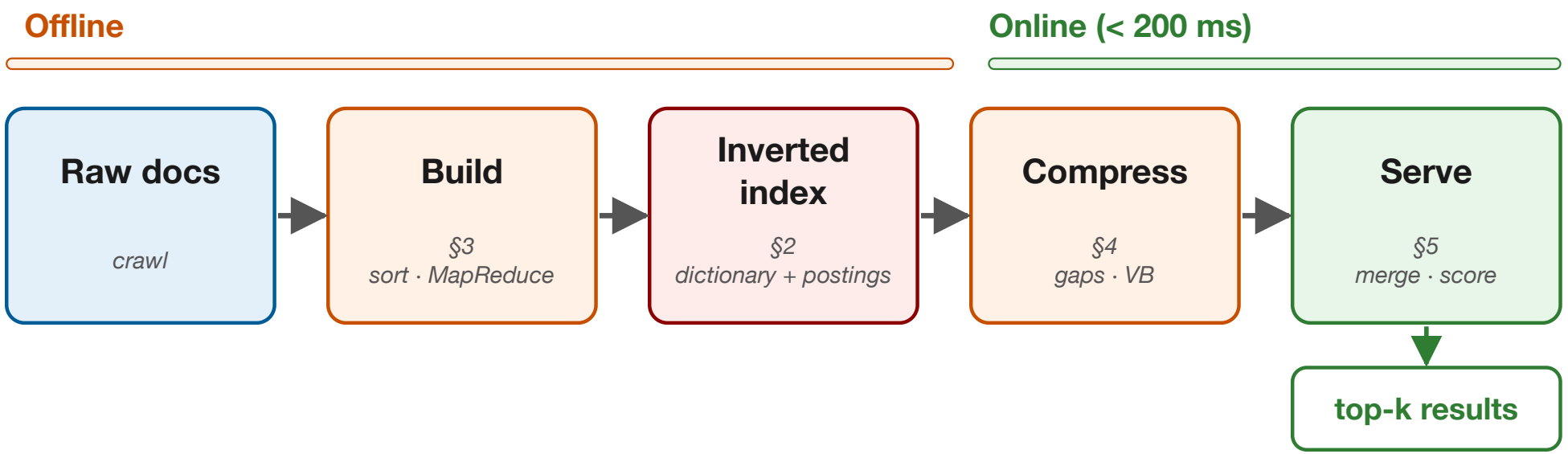
This lecture builds, from first principles, the architecture of large-scale search engines. It is delivered in two parts. **Part I (today)** covers a single machine, in **5 sections**:

1. **The Web-Scale Retrieval Problem** — framing the challenge & latency constraints
2. **The Inverted Index** — the core data structure & incidence-matrix view
3. **Constructing the Inverted Index** — linguistic pipeline & MapReduce sorting
4. **Posting Lists and Compression** — gap encoding & byte-aligned schemes
5. **Query Processing** — Boolean merges, skip pointers, ranked scoring (TAAT vs. DAAT)

Part II then distributes this index across a cluster: document vs. term partitioning (§6–7), routing, caching and replication (§8), and the synthesis (§9).

> *Extended material (derivations, BSBI, Golomb-Rice codes, full TAAT/DAAT traces, etc.) is in the **Appendix** (A.1–A.5): each section divider points to its appendix counterpart.*

Part I in One Picture



Part II distributes this whole pipeline across a cluster

Keep this map in mind: every section of today adds one block to it.

Why This Matters: Opening Module 3

Module 2 closed with graph representations. Module 3 now shifts focus from **representation** to **classical algorithmic tasks** re-engineered to operate at massive scale.

- **The original "big data" problem:** organizing and querying a corpus of tens of billions of documents that no single machine can hold, updated continuously.
- **Foundational data structure:** the inverted index makes sub-linear retrieval tractable, laying the groundwork for link analysis, hashing, and sampling in Module 3.
- **Cross-module synthesis:** directly connects distributed infrastructure from Module 1 (MapReduce, GFS/HDFS) with high-dimensional sparsity and text representations from Module 2 (sparse vectors, TF-IDF, cosine similarity).

§ 1 — The Web-Scale Retrieval Problem

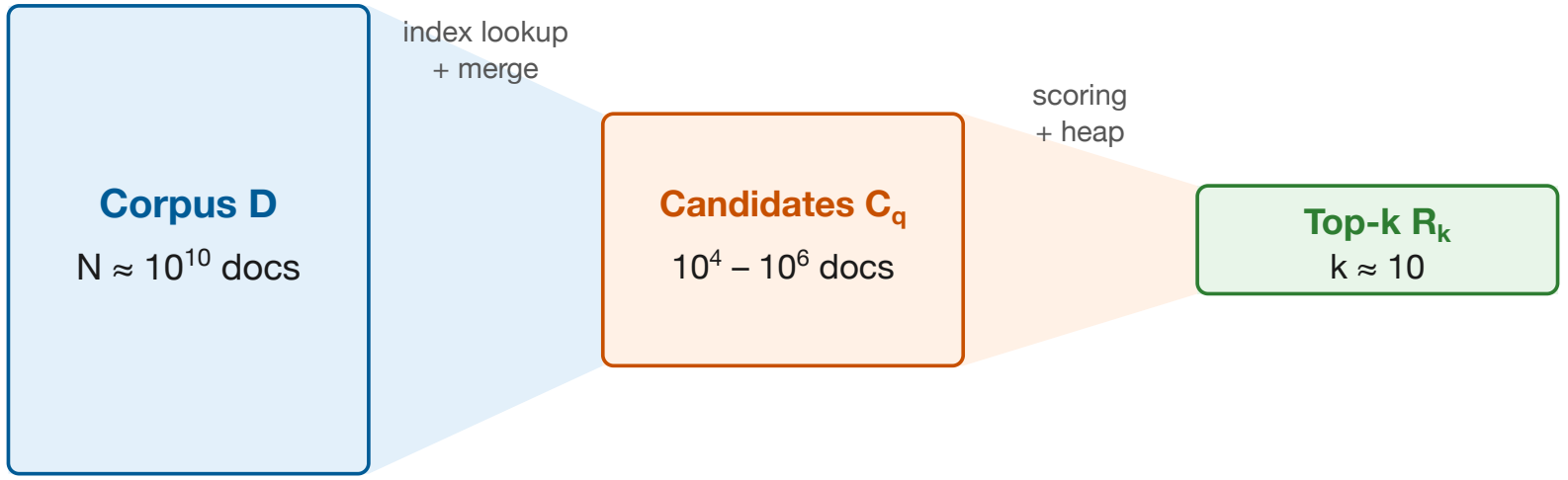
Framing the challenge

📎 *Extended material → Appendix A.1*

Formalizing the Retrieval Task

Corpus D of N documents, query $q = (t_1, \ldots, t_{|q|})$. The engine returns the k most relevant documents, best first:

$$\text{Retrieve}(q, D) \longrightarrow R_k \subseteq D, \qquad k \ll N \text{ (top-10 or top-20)}$$

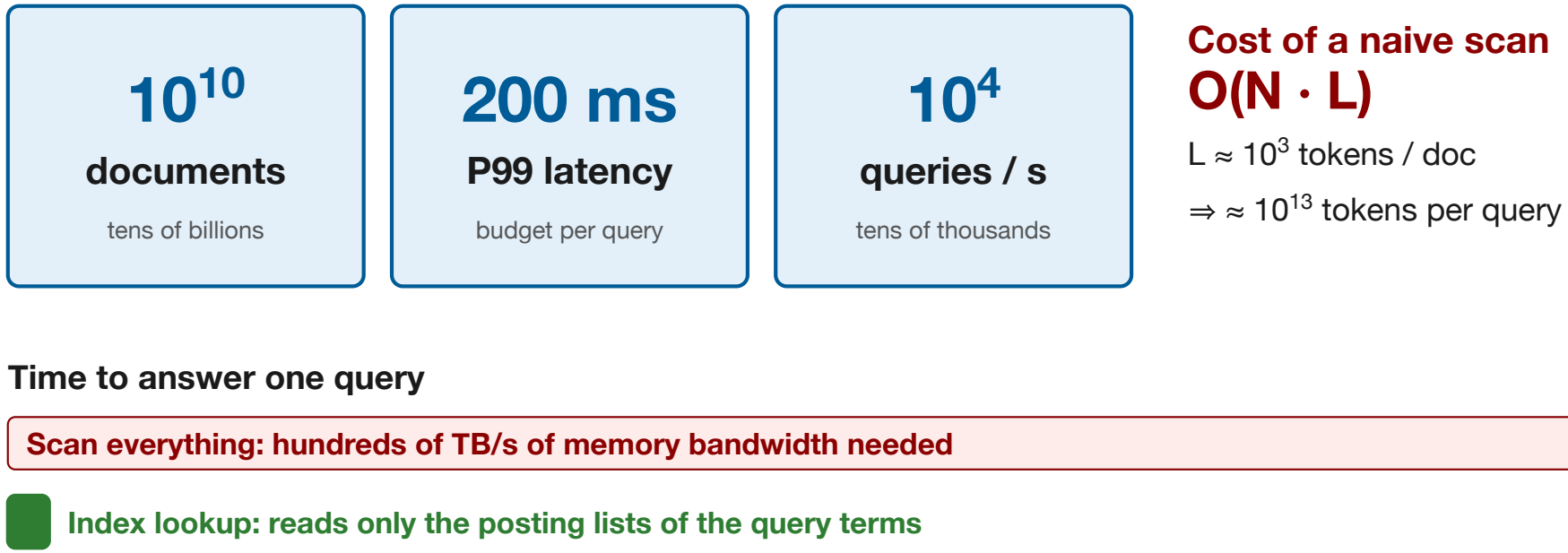


Never touch all N documents: each stage shrinks the set by orders of magnitude

- Without examining all $N \approx 5 \times 10^{10}$ documents: query time must be strictly **sub-linear** in N. *(Formal relevance ordering: Appendix A.1.1)*

Systems Scale & Latency Requirements

Planetary scale imposes strict Service Level Objectives:



- A naive scan costs $O(N \cdot L)$ with $L \approx 10^3$ tokens per document: hundreds of TB/s of memory bandwidth for *one* query.
- *Conclusion: linear scans are infeasible. We must precompute an index.*

The Two Sub-Problems of Retrieval

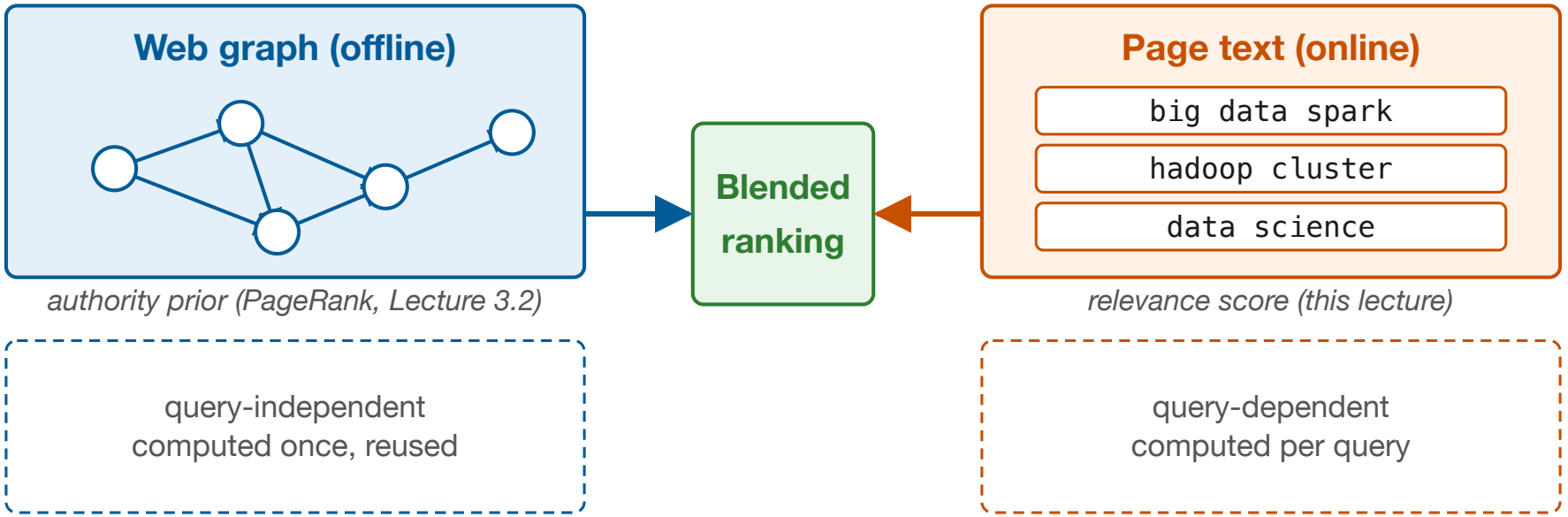
Web-scale retrieval naturally decomposes into two distinct phases:

1. **Candidate Selection (Boolean / Filtering Phase):**
    - *Which documents contain the query terms at all?*
    - Rapidly filters 10^{10} documents down to candidate pool C_q (10^4–10^6 pages).
    - Solved via the **inverted index** (§2–§4) using sorted posting merges (§5).
2. **Relevance Scoring (Ranking Phase):**
    - *Among candidates, which are most relevant to the user?*
    - Computes statistical/geometric relevance scores (TF-IDF, BM25, cosine similarity).
    - Blended with offline graph signals (e.g., PageRank, Lecture 3.2).

The inverted index is uniquely elegant: candidate filtering and relevance scoring share the exact same underlying data structure.

The Web as a Graph: Content vs. Structure

Search engines combine two complementary signals:



- **Structure** (Lecture 2.4, 3.2): query-independent authority, computed offline.
- **Text** (this lecture): query-dependent relevance, computed online by an index that serves queries in milliseconds.

✅ Checkpoint 1 — Framing the Problem

Test your understanding:

1. State the retrieval task formally, in terms of a corpus D, a query q, and a result set R.
2. Why does the naive linear-scan approach fail at web scale?
3. What are the two sub-problems into which web-scale retrieval decomposes?
4. Give two concrete latency/throughput numbers that characterize the systems challenge of web-scale search.

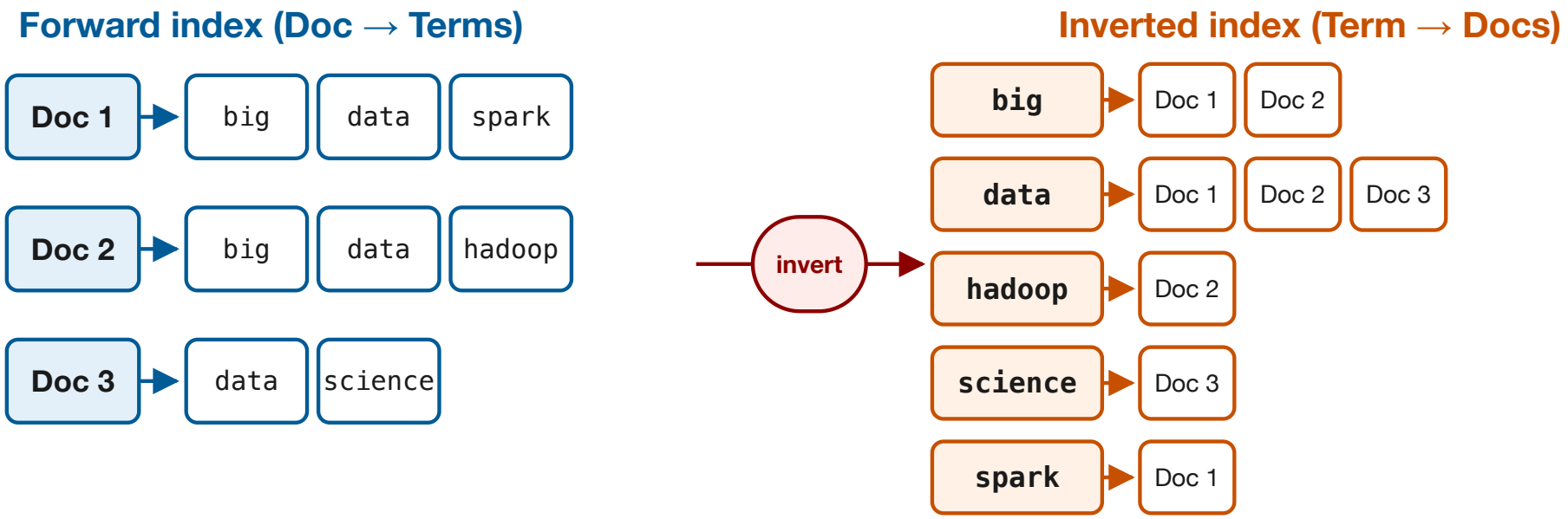
Write down your thoughts before moving to Section 2.

§ 2 — The Inverted Index

The core data structure

📎 *Extended material → Appendix A.2*

Forward Index vs. Inverted Index



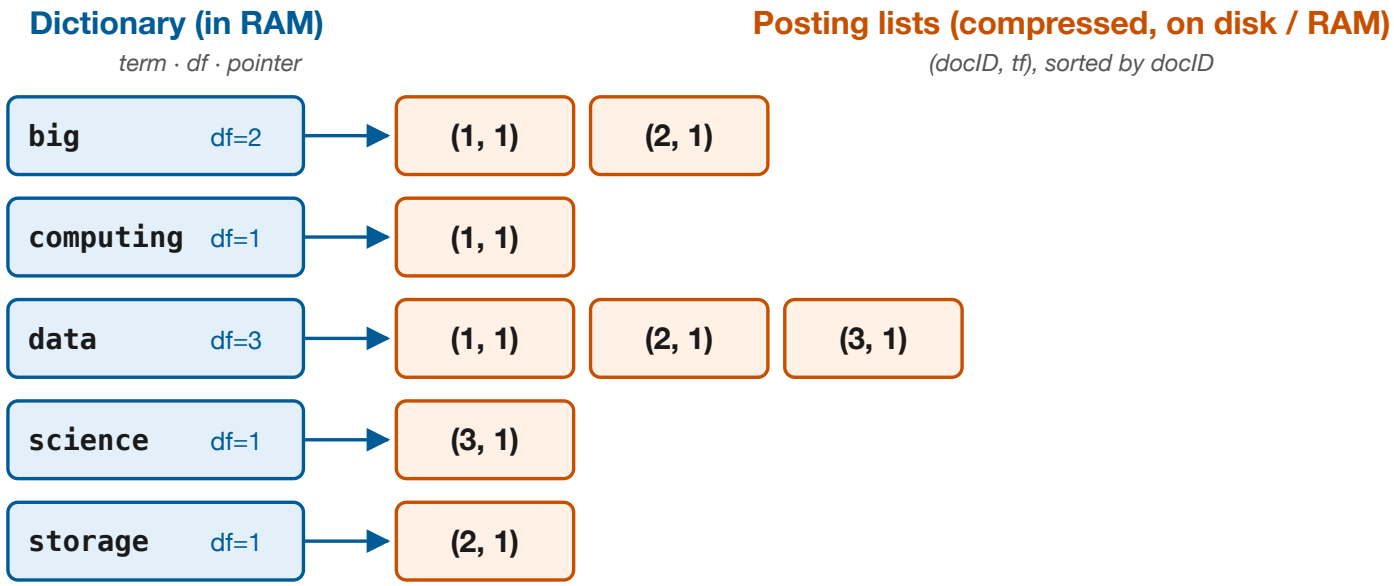
"Which documents contain data?" forward: scan every doc · inverted: one lookup

- **Forward index:** the natural document view. "Which documents contain `data`?" means scanning every document: $O(N \cdot L)$.
- **Inverted index:** maps each term directly to its document list: one dictionary lookup plus a posting-list read.

Anatomy: Dictionary and Posting Lists

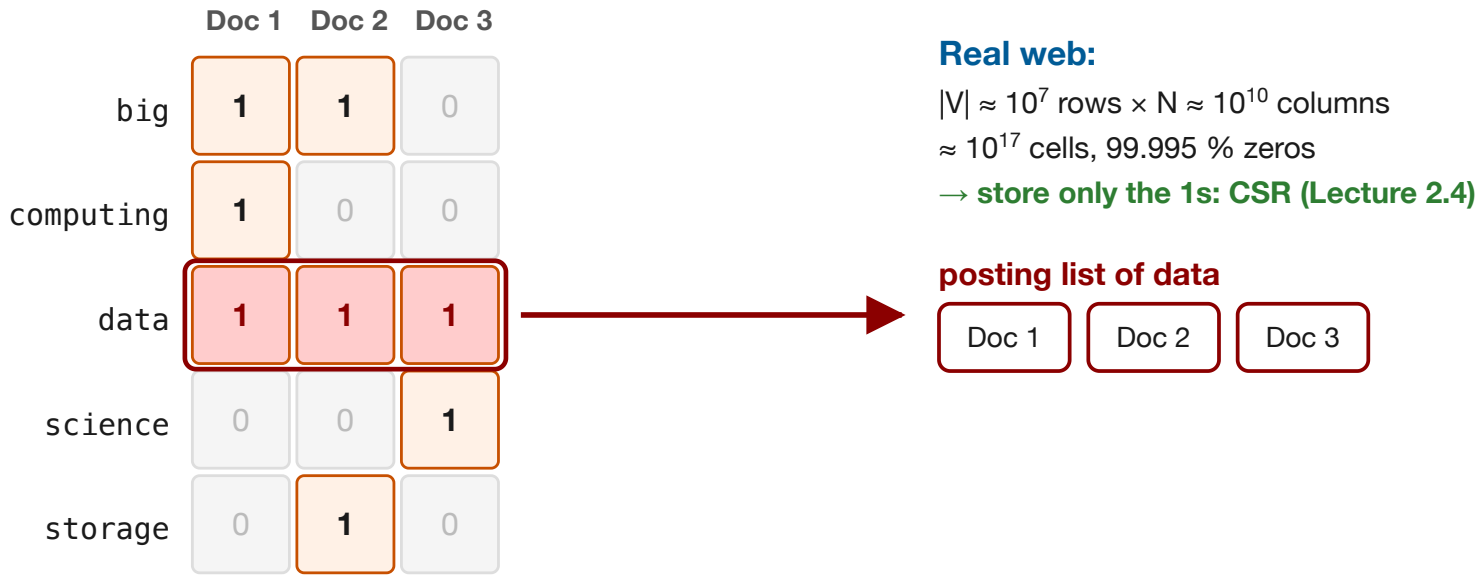
An inverted index has two tightly coupled components:

1. **Dictionary** (vocabulary V): distinct terms, their document frequency df_t and a pointer to the postings. Kept in RAM (hash table or B-tree/trie); grows sub-linearly (Heaps' Law, Appendix A.2.2).
2. **Posting lists:** for each term, postings $(d, \mathrm{tf}_{t,d})$ **strictly ordered by docID**; tf feeds scoring, positions feed phrase queries.



Sparse Matrix View: Term-Document Incidence

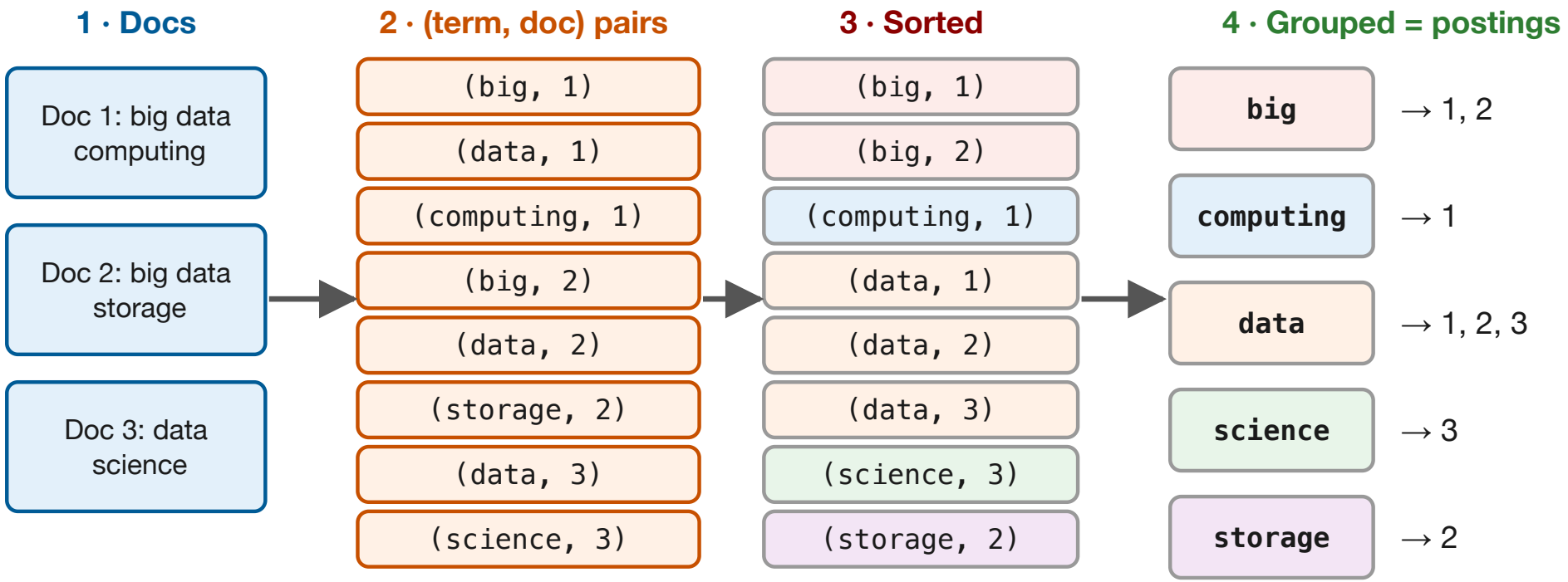
View the collection as a **term-document incidence matrix** (1 if term t occurs in document d):



- Almost every cell is 0 (density $\approx 0.005\%$, derivation in Appendix A.2.1): a dense matrix is petabytes.
- A posting list stores only the 1s of a row: the inverted index is **CSR** (Lecture 2.4) applied to this matrix.

Worked Toy Example: Building the Index

Three toy documents: build the index in three moves: **extract** (term, docID) pairs, **sort**, **group**.



Dictionary is sorted for prefix queries; postings are sorted for fast linear merging.

✅ Checkpoint 2 — The Inverted Index

Test your understanding:

1. Contrast the forward index and the inverted index, and explain why the inversion enables sub-linear term lookup.
2. What are the two components of an inverted index, and what does each store?
3. How does the term-document incidence matrix relate to the sparse adjacency-list ideas from Lecture 2.4?
4. Build, by hand, the inverted index for two additional tiny documents of your choice.

Discuss with your colleague or write down your answers.

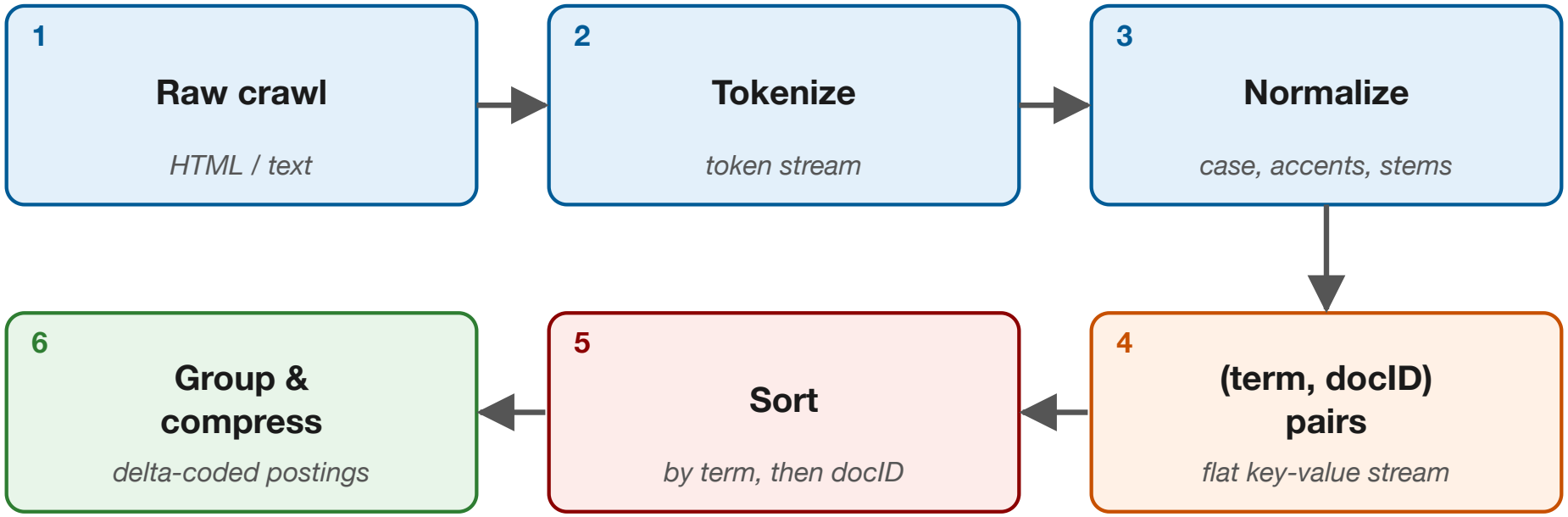
§ 3 — Constructing the Inverted Index

From raw documents to postings

📎 *Extended material → Appendix A.3*

Index Construction Pipeline

Transforming a raw web crawl into an inverted index follows a clear pipeline:



The bottleneck is the Sort stage: $K \approx 10^{12}$ pairs

Linguistic Processing: Tokens & Normalization

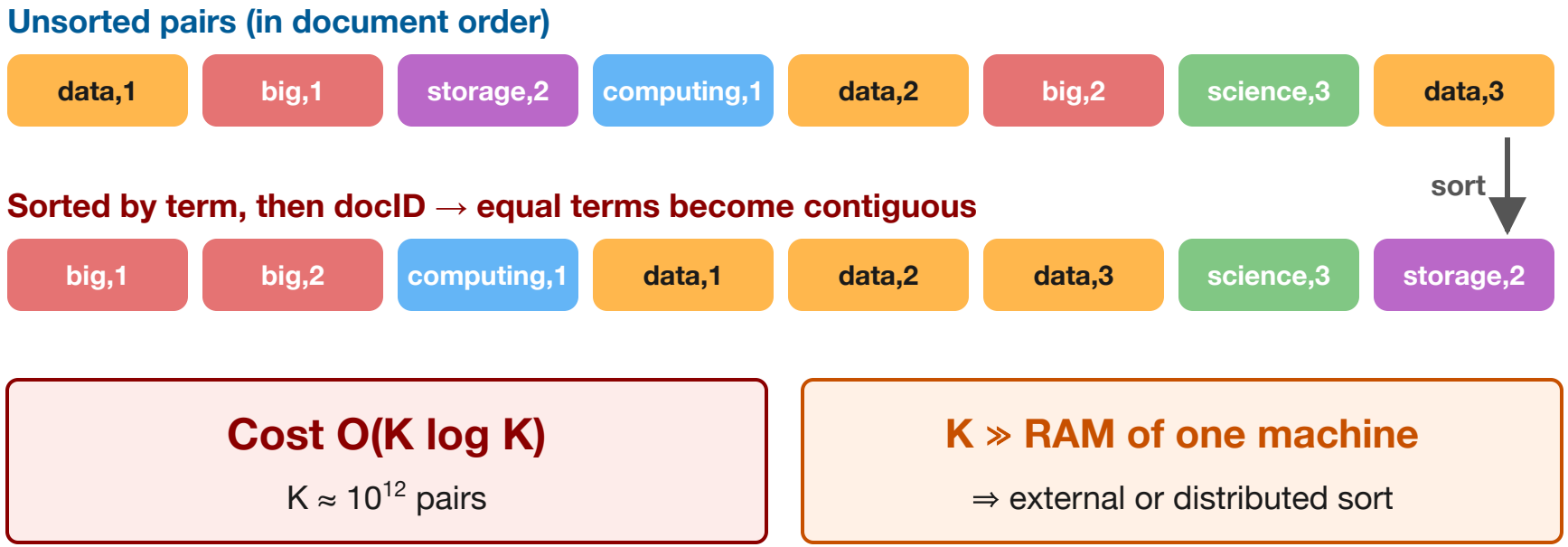
Raw text must be converted into standardized search terms:

- **Tokenization:**
  - Segments text on whitespace and punctuation: `"Big Data Computing!"` → `["Big", "Data", "Computing"]`.
  - Edge cases: email addresses, hyphenated words (`state-of-the-art`), URLs.
- **Normalization & Stemming:**
  - **Case Folding:** maps uppercase to lowercase (`"Big"` → `"big"`).
  - **Stemming (e.g., Porter Stemmer):** chops suffixes to conflate morphological variants (`"computing"`, `"computed"` → `"comput"`).
  - **Stopword Handling:** frequent words (`"the"`, `"in"`) can be filtered or compressed.

Normalization prevents vocabulary explosion and fragmentation across equivalent variants.

Sort-Based Construction & The Bottleneck

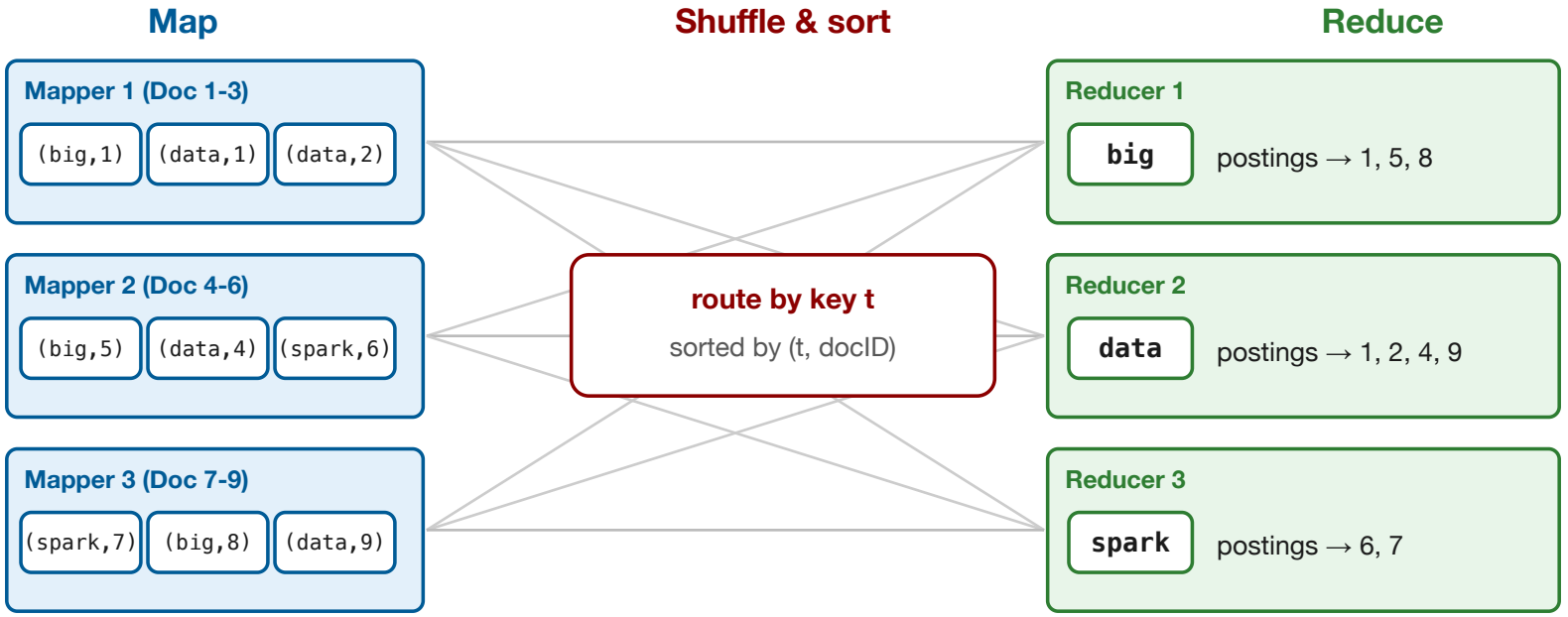
Once tokens are paired into (t, d) tuples, index creation is fundamentally a **sorting problem**:



- Generating the pairs is one linear pass over the text. **Sorting** them is the cost: $O(K \log K)$ with $K \approx 10^{12}$.
- K exceeds one machine's RAM by orders of magnitude, so the sort must be external or distributed.

MapReduce for Scalable Index Construction

Index construction maps directly onto **MapReduce** (Lecture 1.2):



Infrastructure solves the sort bottleneck; our code is only Map and Reduce

- **Map:** each mapper parses its local blocks (HDFS/GFS, Lecture 1.1) and emits (t, d) pairs.
- **Shuffle & sort:** the framework routes all pairs of a term t to one reducer, sorted by docID.
- **Reduce:** each reducer writes the posting list of its terms. *(Formal signatures and pseudocode: Appendix A.3.3–A.3.4)*

✅ Checkpoint 3 — Constructing the Inverted Index

Test your understanding:

1. List, in order, the stages of the index construction pipeline, from raw documents to posting lists.
2. Describe the sort-based index construction algorithm in your own words.
3. Map the sort-based construction algorithm onto MapReduce's Map, Shuffle, and Reduce phases.
4. Why is index construction considered a canonical, textbook example of a MapReduce application?

Discuss with your colleague or write down your answers.

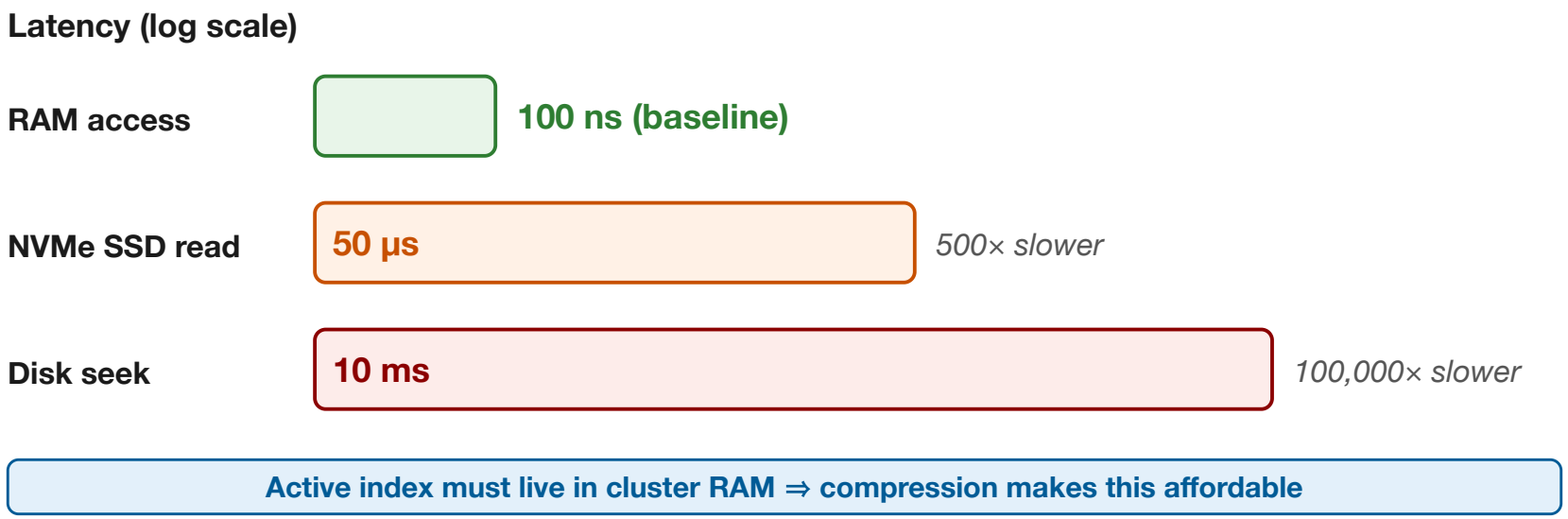
§ 4 — Posting Lists and Compression

Making the index small enough to serve

📎 *Extended material → Appendix A.4*

Why Compression Matters at Web Scale

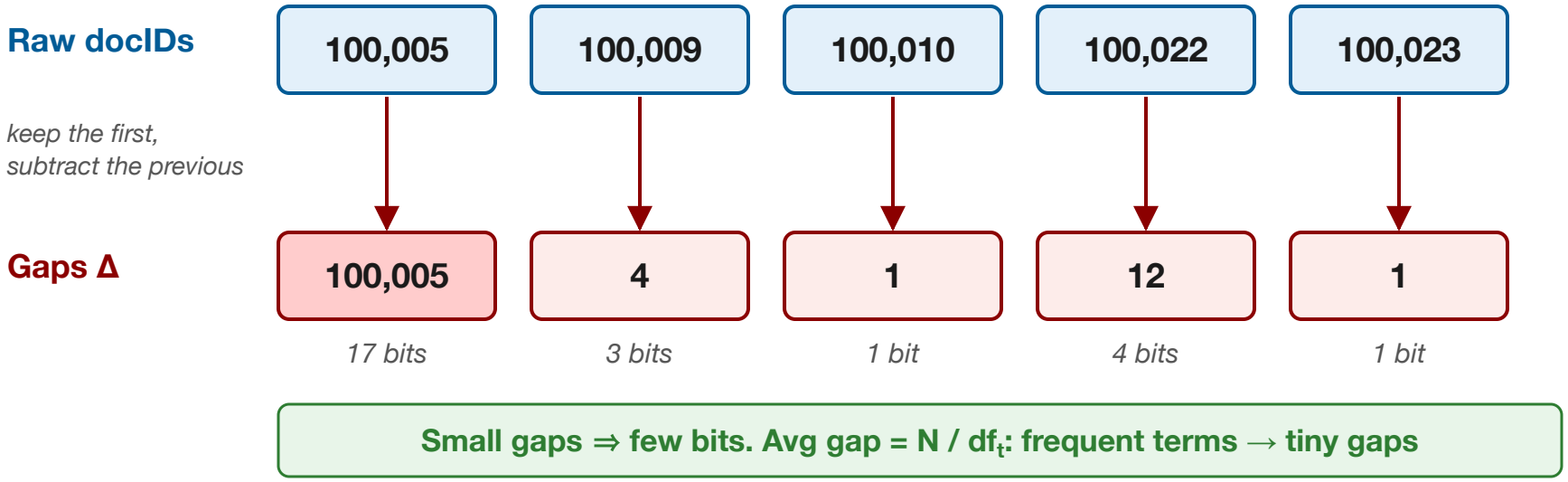
With $N \approx 10^{10}$ documents and 10^{12} postings, raw 64-bit docIDs alone cost $10^{12} \times 8\,\mathrm{B} = 8\,\mathrm{TB}$, and tf plus positions push the index to tens of TB.



To meet the 200 ms budget the active index must sit in cluster RAM. Compression makes that economically viable.

Gap Encoding (Delta Encoding)

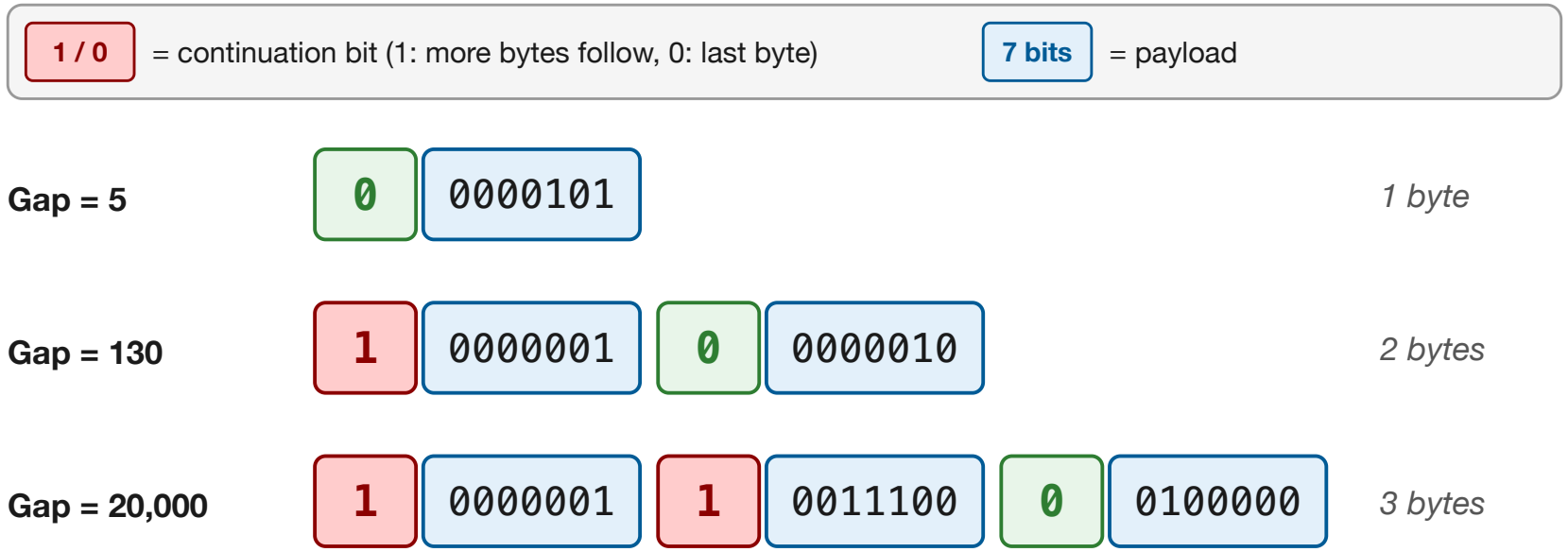
Posting lists are **sorted**, so store the **gaps** between consecutive docIDs instead of the IDs: $\Delta_1 = d_1$ and $\Delta_i = d_i - d_{i-1}$.



- Frequent terms have small average gaps ($\approx N/\mathrm{df}_t$): for a term in 10^8 of 10^{10} documents, about 100, i.e. 7 bits. *(Derivation: Appendix A.4.2)*
- Rare terms have larger gaps, but their lists are short anyway.

Variable-Byte (VB) Encoding

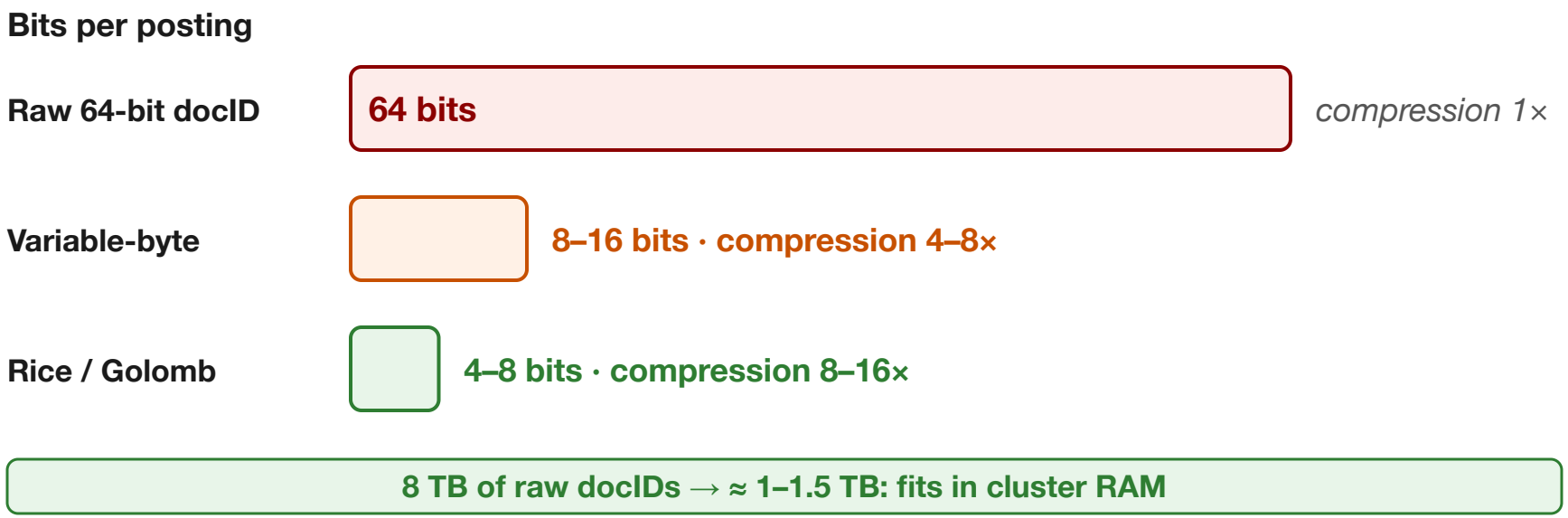
Variable-byte encoding writes each gap as a sequence of bytes: **7 payload bits** plus **1 continuation bit** (`1` = more bytes follow, `0` = last byte).



Byte-aligned ⇒ fast decoding. Frequent terms: 8 bytes → 1 byte per posting (8×)

- Byte alignment means fast CPU decoding, no bit shifts. *(Worked derivations: Appendix A.4.3)*

Compression Ratios & Serving Trade-Offs



- **Storage payoff:** 8 TB of raw postings shrinks to about 1–1.5 TB and fits in cluster RAM.
- **Latency payoff:** decoding costs microseconds (VB: very low; Rice/Golomb: bit shifts, moderate), dwarfed by avoiding disk I/O.
- *Comparison table: Appendix A.4.5.*

✅ Checkpoint 4 — Posting Lists and Compression

Test your understanding:

1. Why does compression matter so much for web-scale inverted indices?
2. Explain gap (delta) encoding, and compute the gap sequence for the posting list $\langle 5, 6, 15, 16, 17 \rangle$.
3. Why does gap encoding provide greater relative benefit for frequent terms than for rare terms?
4. Contrast variable-byte encoding with Golomb/Rice coding, at a conceptual level.

Consolidate your answers before proceeding.

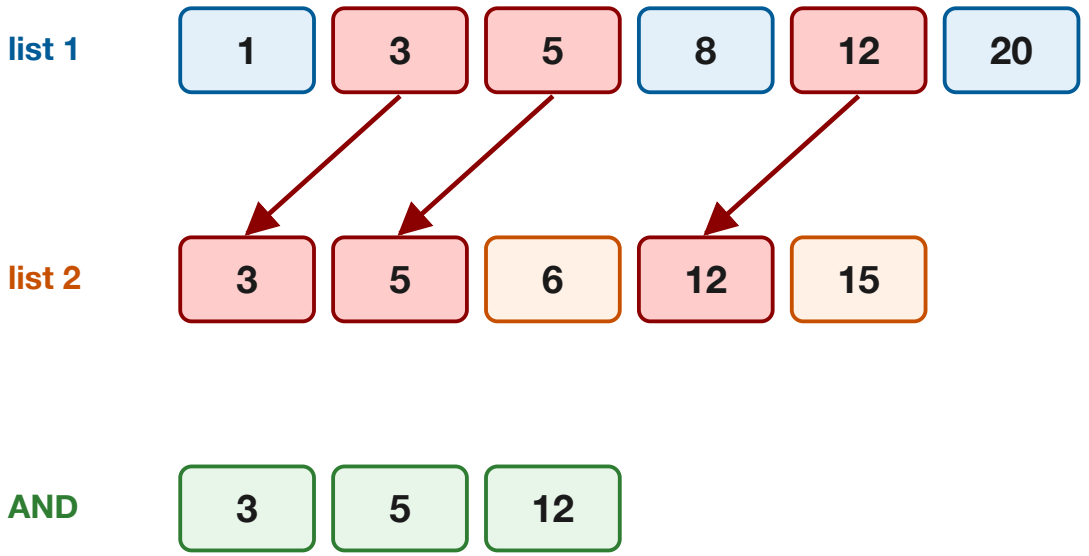
§ 5 — Query Processing over Inverted Indices

Boolean retrieval and ranked retrieval

📎 *Extended material → Appendix A.5*

Boolean Retrieval: Posting List Merges

AND is the *intersection* of the query terms' posting lists, **OR** their *union*. Because lists are sorted, intersection is a two-pointer **linear merge**:



Both lists sorted ⇒
one pass, never back up

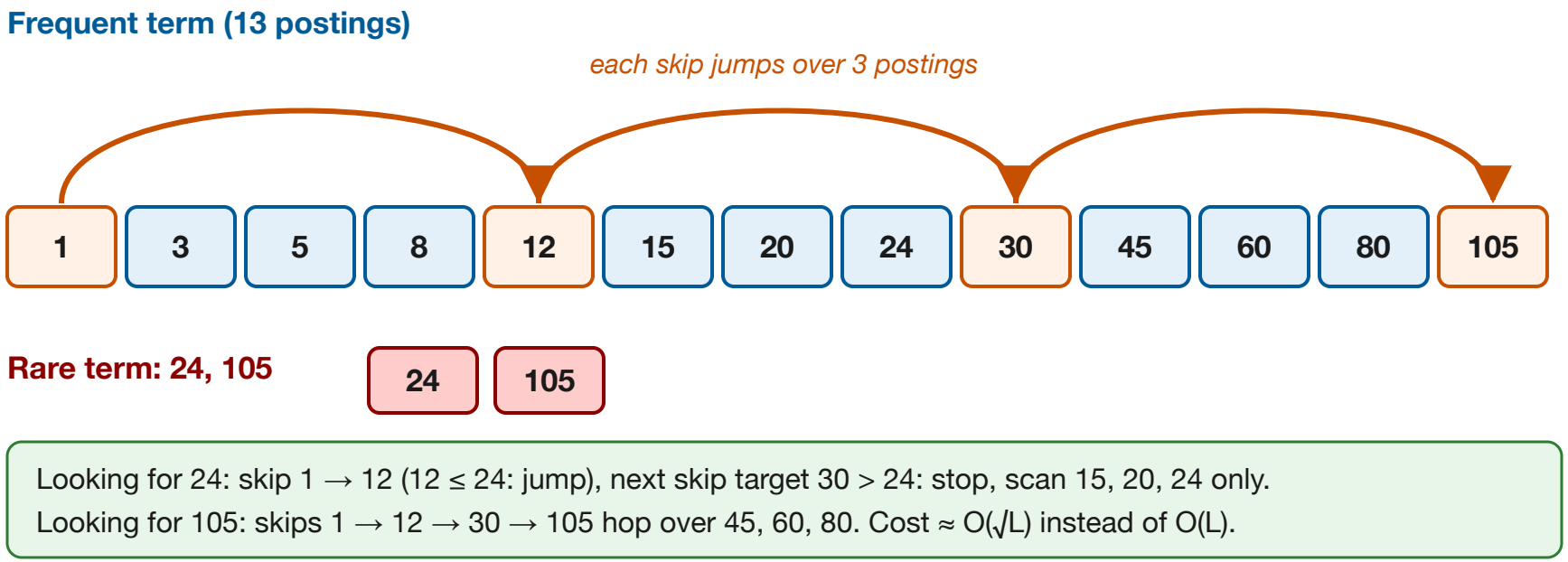
Cost: O(n_1 + n_2)

Rule at each step:
equal → output, advance both
smaller → advance that pointer

- Far superior to the nested-loop intersection, $O(n_1 \cdot n_2)$. *(Formulas and code: Appendix A.5.1, A.5.7)*

Accelerating Intersection: Skip Pointers

When list lengths differ wildly (rare ∩ frequent), reading the whole long list is wasteful. **Skip pointers**, placed every $\approx \sqrt{L}$ postings, let the merge jump ahead:



- If the skip target is $\leq$ the search value, jump; otherwise scan the short block. Cost drops from $O(L)$ to about $O(\sqrt{L})$. *(Optimal interval: Appendix A.5.2)*

From Boolean to Ranked Retrieval

Boolean retrieval returns an **unranked set**: a web query may match 10^7 documents, yet users read the top 10. We must **score** and rank.

Reconnecting to Lectures 2.1 and 2.3: represent $\mathbf{q}$ and $\mathbf{d}$ as TF-IDF vectors (weight $w_{t,d} = \mathrm{tf}_{t,d} \cdot \mathrm{idf}_t$) and rank by cosine similarity. For a fixed query this reduces to a dot product, normalised by the document length:

$$\mathrm{score}(q, d) \;\propto\; \sum_{t \in q \cap d} w_{t,q} \cdot w_{t,d}$$

Key insight: $\mathrm{tf}_{t,d}$ is in the posting list and df_t is in the dictionary: the inverted index stores everything a cosine score needs. (Full TF-IDF and cosine formulas: Appendix A.5.3.)

Efficient Scoring via Sparse Postings

A dense dot product over $\mathbb{R}^{|V|}$ with $|V| \approx 10^7$ is hopeless. **Sparsity saves us:**

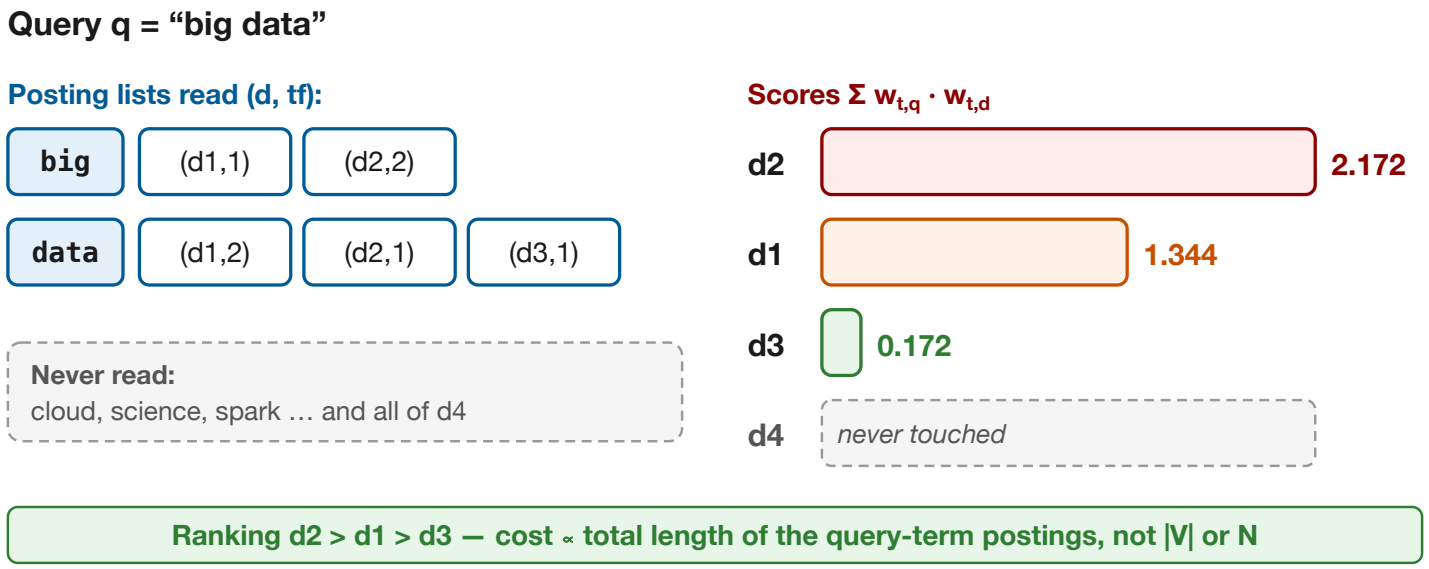
- Only terms present in **both** q and d contribute: every other product is exactly zero.
- So we read **only the posting lists of the query terms**: the work is the total length of those lists, not $|V|$ and not N.

*How should the engine accumulate these partial scores? Two paradigms exist: **TAAT** and **DAAT**.*

Worked Example: Ranked Scoring over the Index

Toy corpus, $N = 4$, $\mathrm{idf}_t = \log_2(N/\mathrm{df}_t)$: d_1 `"data big data spark"`, d_2 `"big data big"`, d_3 `"data science"`, d_4 `"cloud spark"`. Query `"big data"`:

Term	df_t	idf_t	Postings (d, tf)
`big`	2	1.000	$(d_1, 1), (d_2, 2)$
`data`	3	0.415	$(d_1, 2), (d_2, 1), (d_3, 1)$



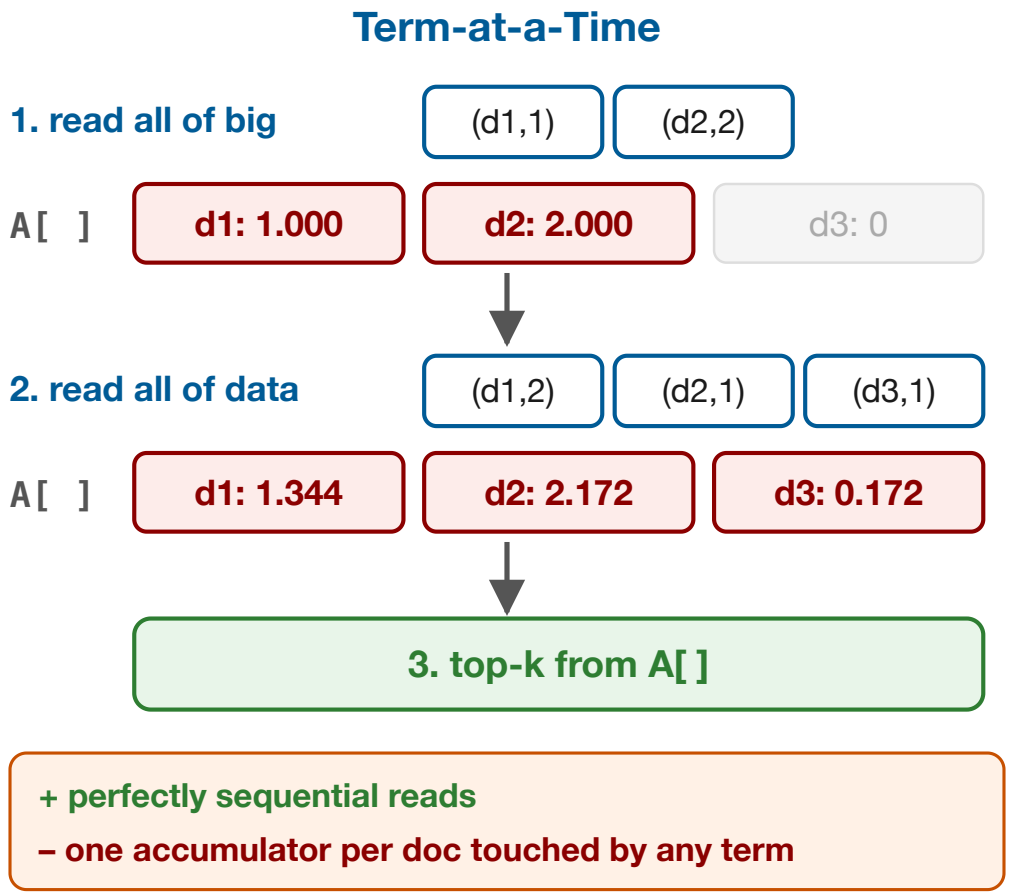
Full weights and length normalisation: Appendix A.5.4.

Term-at-a-Time (TAAT) Scoring

TAAT: the **outer loop runs over query terms**.

1. Keep an accumulator $A[d]$ per document, all starting at 0.
2. For each query term, read its **entire** posting list and add each document's partial score to $A[d]$.
3. After the last list, normalise by $\|\mathbf{d}\|_2$ and extract the top-k.

- **Strength:** perfectly sequential list access: ideal for disk and cache streaming.
- **Weakness:** one accumulator per candidate document: memory can explode for common terms.

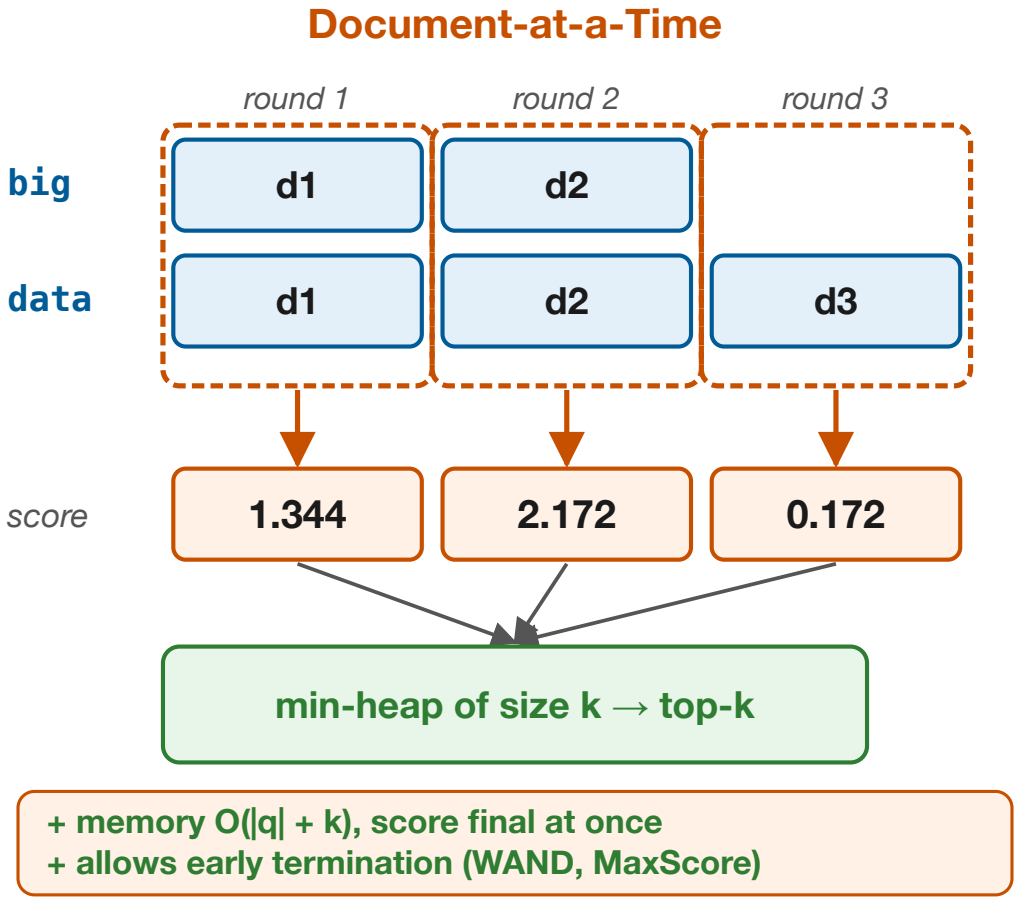


Document-at-a-Time (DAAT) Scoring

DAAT: the **outer loop runs over documents**; all lists advance in lockstep.

1. Keep one pointer per query term.
2. Take the **smallest current docID**, advance every pointer on it, and compute its **final** score at once.
3. Keep a **min-heap of size** k: if the score beats the heap minimum, insert it, then forget the document.

- **Strength:** memory $O(|q| + k)$; early termination (WAND).



Summary Comparison: TAAT vs. DAAT

Both compute the same scores: they differ in loop order, memory and early termination.

Dimension	Term-at-a-Time (TAAT)	Document-at-a-Time (DAAT)
Memory Footprint	$O(\lvert\text{Candidates}\rvert)$ accumulators	$O(\lvert q\rvert + k)$ pointers & top-k heap
I/O Access Pattern	Perfectly sequential per list	Synchronized concurrent reads
Score Completion	Only after last term finishes	Completed immediately per docID
Early Termination	Difficult	Highly effective (WAND / MaxScore)
Industry Adoption	Specialized retrieval engines	**Dominant in web-scale search**

✅ Checkpoint 5 — Query Processing

Test your understanding:

1. Which set operation on posting lists implements AND, and which implements OR?
2. Explain why sorted posting lists allow $O(n_1 + n_2)$ intersection, and describe the merge algorithm.
3. What are skip pointers, and when do they provide the greatest benefit?
4. Using the worked example, compute $\mathbf{q} \cdot \mathbf{d}_3$ for the one-term query $q =$ `"data"`: which posting entry is needed, and what is its TF-IDF weight?
5. Trace TAAT and DAAT for a two-term query over a tiny index of your own, and verify that both give the same ranking.

Consolidate your answers before proceeding.

Summary of Part I

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.

Key Terms of Part I

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

Next: Part II — Distributed Search at Scale

Everything so far fits on **one machine**. The index of the whole web does not: it is **terabytes** of postings, and a query must still be answered in **milliseconds**.

- **How do we distribute the index across a cluster?**
- Split by **documents** or by **terms**? What does each choice cost in network traffic and load balance?
- How do routing, caching and replication turn many machines into one search engine?

Part II answers these questions (§6–§9).

Lecture 3.1 (Part I) 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.1	§ 1	Scale arithmetic behind linear-scan infeasibility; candidate retrieval vs. fine-grained ranking	2
A.2	§ 2	Sparsity derivation, Heaps' Law, dictionary data structures, formal definitions	4
A.3	§ 3	Tokenization and stemming rules, BSBI external sorting, MapReduce pseudocode and formal signatures	4
A.4	§ 4	Why compression speeds up queries, expected gap sizes, VB worked encodings, Golomb-Rice codes, ratio table	5
A.5	§ 5	Multi-term ordering, skip-pointer optimality, TF-IDF/cosine derivation, full TAAT and DAAT traces, Boolean formulas	7

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

📎 A.1 — Scale Arithmetic & Ranking Stages

A.1.1 · Formalizing the Task and Scale Parameters

Let $D = \{d_1, d_2, \ldots, d_N\}$ denote a corpus of N documents over vocabulary V. A user query $q = (t_1, t_2, \ldots, t_{|q|})$ is an ordered sequence of terms where $t_i \in V$.

The search engine computes an operator:

$$\text{Retrieve}(q, D) \longrightarrow R_k = \langle d_{(1)}, d_{(2)}, \ldots, d_{(k)} \rangle$$

where $k \ll N$ (typically $k \in \{10, 20\}$) and documents satisfy:

$$\text{Relevance}(q, d_{(1)}) \geq \text{Relevance}(q, d_{(2)}) \geq \cdots \geq \text{Relevance}(q, d_{(k)})$$

The Systems Challenge In Numbers:

- $N \approx 5 \times 10^{10}$ indexed pages; average length $L \approx 10^3$ tokens $\implies$ $T \approx 5 \times 10^{13}$ tokens.
- Target Service Level Objective (SLO): 99th percentile latency $P_{99} \leq 200\,\text{ms}$ at $50,000\,\text{queries/second}$.
- Scanning 5×10^{13} tokens in $0.1\,\text{sec}$ demands memory bandwidth of $500\,\text{TB/s}$ per query — physically impossible on commodity servers without indexing.

A.1.2 · Candidate Retrieval vs. Fine-Grained Ranking

Decomposition of Web Search:

1. **Candidate Retrieval (High Recall):**
    - Quickly reduces corpus from $N = 10^{10}$ to candidate pool $C_q = \{d \in D : d \cap q \neq \emptyset\}$, where $|C_q| \approx 10^4 - 10^6$.
    - Primary metric: computational efficiency and zero false negatives on term presence.
2. **Fine-Grained Ranking (High Precision at k):**
    - Scores candidates in C_q using multi-stage rankers (BM25, vector similarities, neural cross-encoders, and graph authority).

📎 A.2 — Sparsity, Heaps' Law & Dictionaries

A.2.1 · Incidence Matrix Sparsity: Derivation

Incidence matrix $M \in \{0,1\}^{|V| \times N}$: $M_{i,j} = 1$ iff term t_i occurs in document d_j.

- Let $L_d \approx 200$–500 be the average number of distinct terms per web page.
- Non-zeros: $\text{nnz}(M) = \sum_{j=1}^{N} |\{t \in d_j\}| = N \cdot L_d$.
- Density:

$$\rho = \frac{\text{nnz}(M)}{|V| \cdot N} = \frac{L_d}{|V|} = \frac{500}{10^7} = 5 \times 10^{-5} = 0.005\%$$

- The matrix is 99.995% zeros: a dense array wastes almost all of its space.

The inverted index is exactly **Compressed Sparse Row (CSR)** storage of M (cf. Lecture 2.4).

A.2.2 · Heaps' Law: Sub-Linear Vocabulary Growth

Why can the search engine fit the dictionary in memory as the web grows?
Heaps' Law models vocabulary size $|V|$ as a function of total corpus tokens T:

$$|V| = k \cdot T^{\beta}, \qquad 0 < \beta < 1 \quad (\beta \approx 0.4 - 0.6)$$

- Because $\beta < 1$, doubling the corpus size does **not** double the vocabulary.
- For $T = 10^{11}$ tokens, $|V| \approx 10^{7}$ distinct terms.
- Even at web scale, the dictionary occupies tens to hundreds of megabytes — easily cached in server RAM!
- Meanwhile, the posting lists grow linearly with T, dominating storage.

A.2.3 · Dictionary Data Structures

Heaps' Law in numbers. With $k \in [10, 100]$ and $\beta \approx 0.5$, a corpus of $T = 10^9$ tokens has $|V| = kT^{\beta} \approx 3 \times 10^5$–$3 \times 10^6$ distinct terms: the dictionary fits in RAM even when the postings need terabytes.

Dictionary structures compared:

- **Hash table:** $O(1)$ average-case lookup; cannot support wildcard/prefix queries (`comput*`).
- **Sorted array / B-tree / trie:** $O(\log |V|)$ or $O(\text{term length})$ lookup; naturally supports prefix queries, autocomplete, and range scans.
- Production systems often combine a front-end trie with a contiguous in-memory array of posting pointers.

A.2.4 · Formal Definitions: Index, Postings, Incidence Matrix

- **Incidence matrix:** $M \in \{0,1\}^{|V| \times N}$ with $M_{t,d} = 1 \iff t \in d$.
- **Posting list:** $\mathrm{PostingList}(t) = \langle (d_1, \mathrm{tf}_{t,d_1}), (d_2, \mathrm{tf}_{t,d_2}), \ldots \rangle$ with $d_1 < d_2 < \cdots$, i.e. the non-zero entries of row $M_{t,*}$.
- **Document frequency:** $\mathrm{df}_t = |\mathrm{PostingList}(t)|$.
- **Index size:** $\sum_{t \in V} \mathrm{df}_t = \mathrm{nnz}(M)$, the total number of postings.
- **Naive scan vs. index:** $\mathrm{Cost}(\mathrm{scan}) = O(N \cdot L)$; with the index, a one-term query costs $O(1)$ dictionary lookup $+O(\mathrm{df}_t)$ posting reads.

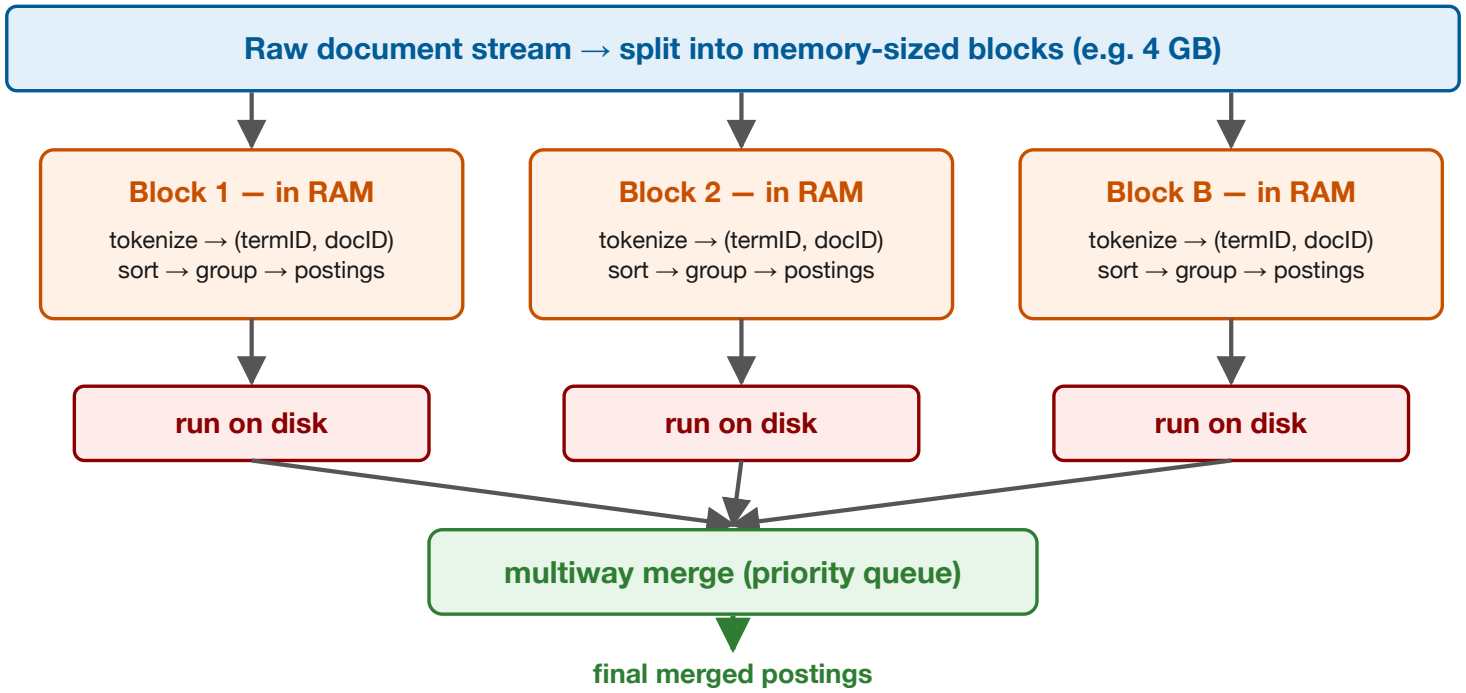
📎 A.3 — Linguistics, BSBI & MapReduce Code

A.3.1 · Tokenization, Normalization, and Stemming

1. **Parsing & tokenization:** strip markup (HTML, JavaScript); decide on hyphens (`state-of-the-art`), apostrophes (`user's`), URLs and e-mail addresses.
2. **Normalization:** case folding (`Apple` → `apple`); accent stripping (`résumé` → `resume`).
3. **Stemming vs. lemmatization:**
    - *Porter stemmer:* rule-based suffix chopping (`computing`, `computed` → `comput`).
    - *Lemmatization:* dictionary-based reduction to the lemma (`better` → `good`).
4. **Stopwords:** classical IR dropped `the`, `is`, `at`; modern engines keep them for phrase queries (`to be or not to be`) in compressed auxiliary structures.

A.3.2 · External Sort-Based Indexing (BSBI)

When the collection of (t, d) pairs exceeds RAM, single-machine indexing uses **Blocked Sort-Based Indexing (BSBI)**:



- **I/O Complexity:** $O\left(\frac{T}{M}\log\frac{T}{M}\right)$ disk operations, where T is total postings and M is memory buffer capacity.

A.3.3 · Distributed Index Construction: MapReduce Pseudocode

```
# MapReduce Specification for Inverted Index Construction

def map(document_id d, document_text text):
    terms = tokenize_and_normalize(text)
    term_counts = collections.Counter(terms)
    for term, count in term_counts.items():
        emit_intermediate(term, (d, count))

# Framework Shuffle & Sort:
# Groups all values by term key, sorts docIDs within each key partition

def reduce(term t, list_of_pairs postings):
    posting_list = []
    doc_freq = 0
    for doc_id, tf in postings:
        posting_list.append((doc_id, tf))
        doc_freq += 1
    compressed_postings = compress(posting_list)
    emit(t, (doc_freq, compressed_postings))
```

A.3.4 · MapReduce Index Construction: Formal Signatures

$$\text{Map}(d) \longrightarrow \{(t, d) : t \in d\}$$

$$\text{Reduce}\big(t, \langle d_1, d_2, \ldots\rangle\big) \longrightarrow \big(t, \text{PostingList}(t) = \langle d_1, d_2, \ldots\rangle\big)$$

- Shuffle & sort groups by key t and orders values by docID, so each reducer receives a **sorted** list: no sorting code in `reduce`.
- Pseudocode with term frequencies: previous slide (A.3.3).

📎 A.4 — Compression: Theory & Worked Encodings

A.4.1 · The Compression Imperative: Hardware Context

Hardware Latency Hierarchy Context:

- CPU L1 cache access: $\approx 1\,\text{ns}$
- Main Memory (RAM) access: $\approx 100\,\text{ns}$
- NVMe SSD random access: $\approx 50\,\mu\text{s} = 50,000\,\text{ns}$ ($500\times$ slower than RAM)
- Mechanical disk seek: $\approx 10\,\text{ms} = 10,000,000\,\text{ns}$ ($100,000\times$ slower than RAM)

Why Compression Accelerates Query Processing:

- Storing uncompressed 64-bit docIDs requires huge memory bandwidth: $\text{Throughput} = \frac{\text{Bytes transferred}}{\text{Bus Bandwidth}}$.
- Decompressing Variable-Byte or SIMD-PForDelta encodings takes ≈ 1–2 CPU cycles per posting on modern superscalar architectures.
- The time to decompress a compressed posting in CPU registers is significantly smaller than the latency required to fetch the uncompressed 64-bit word from main memory over PCIe/DDR buses.
- *Compression is therefore a net speedup for query latency, not an overhead.*

A.4.2 · Gap Encoding: Expected Gap Size

Let posting list $P(t) = \langle d_1, d_2, \ldots, d_{\mathrm{df}_t} \rangle$ have document frequency df_t drawn from corpus size N.

Expected Gap Size Analysis:

- Assuming a uniform distribution of terms across documents, the document IDs act as order statistics:

$$\mathbb{E}[\Delta] = \mathbb{E}[d_i - d_{i-1}] \approx \frac{N}{\mathrm{df}_t}$$

- **Frequent term case:** $\mathrm{df}_t = 10^8$, $N = 10^{10} \implies \mathbb{E}[\Delta] = 100$.
  - An integer of magnitude 100 requires only $\lceil \log_2(100) \rceil = 7$ bits!
  - Compared to 64 bits, theoretical compression potential is $64/7 \approx 9.1\times$.
- **Rare term case:** $\mathrm{df}_t = 10^2$, $N = 10^{10} \implies \mathbb{E}[\Delta] = 10^8$. Requires ≈ 27 bits.
  - However, list length is only 100 postings $\implies$ total space is negligible anyway.
- *Conclusion:* Delta encoding naturally allocates minimal bits to the longest posting lists that dominate aggregate index size.

A.4.3 · Variable-Byte Encoding: Worked Examples

Variable-Byte Algorithm:

- Extract 7-bit chunks of the integer from least to most significant.
- In the final byte (lowest significance), set continuation bit to `0`.
- In preceding continuation bytes, set continuation bit to `1`.

Worked Example 1: Gap = 5

- Binary: `0000101` (7 bits) $\Longrightarrow$ fits in 1 byte: `[00000101]` (`0x05`).

Worked Example 2: Gap = 130

- Binary: `10000010` (8 bits $\Longrightarrow$ requires 2 bytes).
- Lower 7 bits: `0000010` (terminal byte $\rightarrow$ bit 7 is `0`): `[00000010]`.
- Upper bits: `1` $\rightarrow$ padded to 7 bits: `0000001` (continuation byte $\rightarrow$ bit 7 is `1`): `[10000001]`.
- Encoded stream: `10000001 00000010` (`0x81 0x02`).

A.4.4 · Bit-Level Compression: Golomb-Rice Codes

Variable-byte encoding cannot compress below 8 bits per posting. For ultra-dense posting lists ($\mathbb{E}[\Delta] < 8$), bit-level codes achieve higher compression:

Golomb Coding (Optimal for Geometric Distributions):

- Let parameter $M \approx 0.69 \cdot (N/\mathrm{df}_t)$.
- For gap Δ:
  - Quotient $q = \lfloor (\Delta - 1)/M \rfloor$ encoded in **unary** (q ones followed by a zero).
  - Remainder $r = (\Delta - 1) \pmod{M}$ encoded in **truncated binary** ($\lfloor \log_2 M \rfloor$ or $\lceil \log_2 M \rceil$ bits).

Rice Coding (Powers of 2):

- Restricts $M = 2^k$.
- Quotient: $q = (\Delta - 1) \gg k$ (bitwise shift).
- Remainder: $r = (\Delta - 1)\ \&\ (2^k - 1)$ (bitwise mask, exactly k bits).
- Enables fast bitwise operations while achieving 4–6 bits per posting on frequent terms.

A.4.5 · Compression Ratios and Serving Trade-Offs

Encoding Scheme	Bits / Posting	Compression Ratio	CPU Decode Cost
Raw 64-bit DocID	64 bits	1× (baseline)	None
Variable-Byte (VB)	8–16 bits	**4× – 8×**	Very low (byte-aligned)
Bit-level (Rice/Golomb)	4–8 bits	**8× – 16×**	Moderate (bit shifts)

- **Storage:** an 8 TB raw posting set compresses to $\approx 1\text{–}1.5\,\mathrm{TB}$, easily fitting in distributed cluster RAM.
- **Latency:** CPU decompression is microsecond-scale, dwarfed by the cost of avoiding disk I/O and bus transfers.

📎 A.5 — Query Processing: Proofs & Traces

A.5.1 · Boolean Intersection and Multi-Term Queries

```
def intersect(list1, list2):
    answer, p1, p2 = [], 0, 0
    while p1 < len(list1) and p2 < len(list2):
        if list1[p1] == list2[p2]:
            answer.append(list1[p1]); p1 += 1; p2 += 1
        elif list1[p1] < list2[p2]: p1 += 1
        else:                       p2 += 1
    return answer
```

Comparisons $\leq |list_1| + |list_2|$, so the cost is $\Theta(|list_1| + |list_2|)$.

Multi-term AND ($m > 2$): sort terms by increasing df and intersect left to right.

- Example: $\mathrm{df}(t_3) = 10^3 < \mathrm{df}(t_1) = 5 \cdot 10^5 < \mathrm{df}(t_2) = 2 \cdot 10^6$.
- After $P(t_3) \cap P(t_1)$ at most 10^3 candidates survive, so every later merge is against a tiny list.

A.5.2 · Skip Pointers: Optimal Interval

Posting list of length L, skip pointers every S postings (L/S of them).

- Worst case: scan the skip level (L/S steps), then at most S steps inside one block:

$$C(S) = \frac{L}{S} + S$$

- Minimize: $\frac{dC}{dS} = -\frac{L}{S^2} + 1 = 0 \implies S^* = \sqrt{L}.$

> ***Result.*** *The optimal skip interval is $\sqrt{L}$, giving worst-case cost $2\sqrt{L} \ll L$.*

A.5.3 · Vector Space Model and Cosine Similarity

Weights: $w_{t,d} = \mathrm{tf}_{t,d} \times \mathrm{idf}_t$, with $\mathrm{idf}_t = \log \dfrac{N}{\mathrm{df}_t}$, for $\mathbf{d}, \mathbf{q} \in \mathbb{R}^{|V|}$.

$$\mathrm{score}(q, d) = \frac{\mathbf{q} \cdot \mathbf{d}}{\|\mathbf{q}\|_2 \|\mathbf{d}\|_2} = \frac{\sum_{t \in q \cap d} w_{t,q} \cdot w_{t,d}}{\sqrt{\sum_{t \in q} w_{t,q}^2} \cdot \sqrt{\sum_{t \in d} w_{t,d}^2}}$$

- For a fixed query, $\|\mathbf{q}\|_2$ is the same for all documents and can be dropped when ranking.
- $\|\mathbf{d}\|_2$ is precomputed offline and stored in the document table.

A.5.4 · Worked Scoring Example: Weights and Normalization

Corpus (as in § 5), $\log_2$: $\text{idf}(\text{big}) = 1.000$, $\text{idf}(\text{data}) = \log_2(4/3) \approx 0.415$, $\text{idf}(\text{spark}) = 1.000$, $\text{idf}(\text{science}) = 2.000$.

Doc	Weights $w_{t,d} = \text{tf} \cdot \text{idf}$	$\lvert\mathbf{d}\rvert_2$	$\mathbf{q} \cdot \mathbf{d}$	$\mathbf{q} \cdot \mathbf{d} \,/\, \lvert\mathbf{d}\rvert_2$
d_1	big 1.000, data 0.830, spark 1.000	1.640	1.344	0.820
d_2	big 2.000, data 0.415	2.043	2.172	1.063
d_3	data 0.415, science 2.000	2.043	0.172	0.084

- Query weights ($\text{tf}_{t,q} = 1$): $w_{\text{big},q} = 1.000$, $w_{\text{data},q} = 0.415$.
- $\lVert\mathbf{q}\rVert_2$ is the same for every document, so it can be dropped when ranking.
- Here length normalization leaves the ranking unchanged: $d_2 > d_1 > d_3$.

A.5.5 · TAAT Execution Trace

Query `"big data"`; accumulators $A[\cdot]$ start at 0.

```
Read PostingList("big")  = [(d1,1), (d2,2)]
  A[d1] += 1.000·1.000 = 1.000
  A[d2] += 1.000·2.000 = 2.000

Read PostingList("data") = [(d1,2), (d2,1), (d3,1)]
  A[d1] += 0.415·0.830 -> 1.344
  A[d2] += 0.415·0.415 -> 2.172
  A[d3] += 0.415·0.415 -> 0.172      (new accumulator)

Scores final only after the LAST list: extract top-k from A[].
```

Peak memory: one accumulator per document touched by **any** query term.

A.5.6 · DAAT Execution Trace

Pointers: `big` $\rightarrow [d_1, d_2]$, `data` $\rightarrow [d_1, d_2, d_3]$. Min-heap of size k.

```
Round 1: min docID = d1 (in both lists)
  score = 1.000·1.000 + 0.415·0.830 = 1.344   -> push to heap
Round 2: min docID = d2 (in both lists)
  score = 1.000·2.000 + 0.415·0.415 = 2.172   -> push to heap
Round 3: min docID = d3 (only in "data")
  score = 0.415·0.415 = 0.172                 -> push / discard if heap full
Lists exhausted: heap holds the top-k, ranked d2 > d1 > d3.
```

Each score is final the moment its docID is processed; memory stays at $O(|q| + k)$. Because scores are complete early, DAAT supports **early termination** (e.g., WAND, MaxScore).

A.5.7 · Boolean Queries and Skip Pointers: Formulas

$$\text{Retrieve}(t_1 \text{ AND } t_2) = \text{PostingList}(t_1) \cap \text{PostingList}(t_2)$$

$$\text{Retrieve}(t_1 \text{ OR } t_2) = \text{PostingList}(t_1) \cup \text{PostingList}(t_2)$$

- Sorted lists: AND and OR both cost $O(n_1 + n_2)$ with a two-pointer merge; nested loops cost $O(n_1 \cdot n_2)$.
- Skip pointers every $S = \sqrt{L}$ postings: worst case $\frac{L}{S} + S = 2\sqrt{L}$ comparisons (derivation: A.5.2).
- TF-IDF weight and cosine score: A.5.3.