Superblock Pruning for SPLADE Retrieval

Jayaprakash Sundararaj, hi@jayaprakash.net

2026-09-22

Contents

This document is the working plan and results for the task: make the exact full-scan SPLADE index substantially faster by implementing Lightweight Superblock Pruning (LSP) from *"Efficient Sparse Retrieval with Lightweight Superblock Pruning"*, and prove the speed/quality tradeoff with a rigorous evaluation. It covers the problem analysis, the task breakdown and what was delivered, implementation design, evaluation methodology, debugging practices, measured results, and reproduction steps.

Summary

What was built. Lightweight Superblock Pruning (LSP/0) on top of the provided exact SPLADE scanner, as a second index and query mode in the same binary: `scibench index-lsp` writes a forward index plus per-term superblock/block maxima, and `search` / `bench --mode lsp` run the four-stage bound-prune-score query path with the exact index as the reference. Everything the paper's method needs was implemented and measured — 8-bit quantised maxima, β-pruned bounds, top-γ selection, block-level pruning with the θ stop rule — and the document ordering the paper relies on (recursive graph bisection) was added after the corpus turned out to be hash-ordered.

Headline numbers (fixed 1000-query sample, recall of the exact top-10, single thread, same machine; Table 12, with Tables 9–11 as the path there):

corpus	exact scan p50	fast point	99 % point	16 threads
50 M docs	2.1 s	0.896 @ 5.6 ms (374×); 0.831 @ 4.7 ms (448×)	0.991 @ 17 ms (123×)	470–890 QPS
5 M slice	177 ms	0.943 @ 1.8 ms (98×)	0.993 @ 4.0 ms (44×)	

What mattered (§6, "What moved the numbers"): the document ordering first — recursive graph bisection lifts recall at fixed γ from 0.55 to 0.87 at 50 M and shrinks the index by a third — then adaptive β, a global block-bound traversal, AVX-512 kernels for the two hot loops, and larger superblocks (b = c = 32), which halve the bound pass and reach the paper's 99.1 % recall level. A follow-up pass (§6, "Follow-up"; log in `docs/experiments/`) tried twelve more ideas and kept four: ordering only the candidate blocks that get visited (a partial select, then a heap), prefetching in the block-bound sweep, and the b = 16, c = 64 geometry — together −30 % to −35 % latency at both operating points, bit-identical results for all but the geometry. Every change was accepted on the same 1000 queries and the same exact ground truth.

Where it differs from the paper (§7): the paper's 4-bit packed dense maxima were implemented and measured neutral here (queries are 5–10× longer than MS MARCO's, so the bound pass is dominated by long per-term lists, not by the packing); quality is matched (99.1 %), latency is 8–13× the paper's per document because of those query lengths and because only the two hot kernels are vectorised.

Limitations and next steps (§7, §6): no relevance labels, so quality is agreement with the exact top-k (a stricter metric); latency is single-threaded (throughput scales 12× to 16 threads). The follow-up also mapped the negative space: the residual recall is in the top-γ candidate ordering, not in block pruning, and every cheaper-bound-pass idea (term caps, entry-level pruning, dropping frequent terms) loses to plain β on this corpus. The largest remaining stage at high γ is still the block-bound sweep (Figure 9), now at ~35 ns per hit even with prefetching.

1. Problem analysis

1.1 What we have

Table 1. Corpus and baseline facts measured on the instance.

Item	Value (measured on the instance)
Corpus	50,000,000 docs, 2.58 B postings, avg 43 nnz/doc (range ~26–43 seen)
Vocab	49,152 term ids (fits `u16`)
Weights	f32, max ≈ 2.7, long tail of tiny values
Doc order	`doc_id` is a hash, files are hash-sorted ⇒ corpus order is random
Queries	10,000 pre-computed SPLADE vectors; often dozens of terms, many with weight < 0.05
Baseline index	19.2 GB `postings.bin` + 2.9 GB `docids.bin`, built in 65 s (32 cores)

Item	Value (measured on the instance)
Baseline latency	top-100: median 5.6 s, p90 12.9 s, 0.2 QPS
Machine	r6i.8xlarge — 32 vCPU Ice Lake (AVX-512), 256 GB RAM, 495 GB disk

Full corpus statistics and an explanation of each figure are in Appendix A.

1.2 Machine & Environment Details

Table 2. Machine and environment.

Component	Specification
Instance Type	AWS r6i.8xlarge
CPU	32 vCPU Intel Xeon (3rd Gen Ice Lake, AVX-512) @ 3.5 GHz
Memory	256 GB DDR4
Storage	495 GB NVMe SSD (EBS gp3)
Network	30 Gbps (Enhanced Networking)
OS	Linux (Ubuntu 20.04 LTS)
Rust	1.75+ (LLVM opt-level=3, target-cpu=native)

1.3 Why the baseline is slow

search.rs:437-462 walks the *entire* posting list of every query term and accumulates `q_w × d_w` into a `HashMap<u32, f32>`. For a common SPLADE term the list has millions of entries, so a query touches tens to hundreds of millions of postings and hash inserts. Everything else (top-k selection, doc-id lookup) is negligible. The fix is therefore purely about visiting a tiny fraction of documents while still finding the true top-k.

1.4 What the paper does (LSP/0, the recommended variant)

1. Partition the corpus into blocks of `b` docs and superblocks of `c` consecutive blocks (paper: `b=16, c=16` for k=10; `b=4, c=16` for k=1000).
2. Store per-term max weight per block and per superblock (quantized).
3. At query time compute `SBMax(X) = Σ_t q_t · maxW(X, t)` for every superblock using the top-`β` fraction of query terms (query pruning, β = 0.33–0.5).
4. Always visit the top-γ superblocks by `SBMax` (γ = 250–500 for k=10), stopping early once `SBMax ≤ θ` (θ = current k-th best score).

5. Inside each visited superblock, compute block bounds, visit blocks in descending bound order, prune when `bound ≤ θ/η` (η = 1.0 ⇒ safe w.r.t. the bound), and score the block's docs from a block-local forward index using the full query.
6. No average-score guard (that was SP's overhead); quantize doc weights to 8 bits, block/superblock maxima to 4–8 bits.

Reported result on MS MARCO (8.8 M docs): ~0.4 ms/query at 99 % of safe recall vs. 2.5 ms for BMP and 75 ms for MaxScore.

1.5 How our setting differs from the paper — and what it implies

Table 3. How our setting differs from the paper's, and what each difference implies.

Difference	Implication
50 M docs vs 8.8 M (5.7×)	With `b·c = 256` we get ~195 k superblocks; the dense `SBMax` pass is ~195 k × (#pruned query terms) adds — still sub-ms but it is the fixed per-query floor. γ probably needs to scale up.
Random doc order vs similarity-clustered blocks	Block/superblock bounds will be loose (many unrelated docs in one block ⇒ max over unrelated terms). Expect LSP to work but with more blocks visited than the paper. Document reordering is the biggest secondary lever.
No relevance judgements	The exact baseline *is* the oracle; quality = agreement with exact top-k.
Long, noisy queries	Query-term pruning (β) should be a large win; also worth checking a min-weight cutoff.
256 GB RAM, 32 cores	Everything fits in memory (forward index ≈ 8 GB, block maxima ≈ 5–8 GB). Index build can be fully parallel. No need for the paper's SIMDBP compression to fit; it is an optional latency experiment.

---

2. Task breakdown and outcome (24 h budget)

Ordered so that every stage left a working, committed, measurable system.

Table 4. Task breakdown and outcome.

#	Task	Where	Status
0	Environment: instance setup, data download, baseline build/index	`splade-index/`, §1.1	done
1	Ground truth for quality measurement	`bench --mode lsp` runs the exact search per sampled query (untimed) and computes recall@k inline	done (inline, not cached to disk)
2	LSP index builder: forward index + block/superblock maxima, own on-disk format	`index-lsp`, `src/lsp_build.rs`	done
3	LSP search (LSP/0): superblock bounds → top-γ → block pruning → forward scoring	`--mode lsp` in `search`/`bench`, `src/lsp.rs`	done
4	Eval harness: recall@k, latency percentiles, QPS, work counters	`bench` (§4)	done as part of `bench`; per-query CSV and trace diagnostics added (§4.5); multi-threaded QPS via `--threads` (Table 12); RBO/τ not built
5	Parameter sweep: γ, β, c, cluster count	§6	done on the 5 M slice; b = c = 32 and, in the follow-up, nine more b×c geometries at both scales (Tables 11–12); η swept in the follow-up (no use)
6	Document reordering and re-measure	`--reorder dominant|kmeans|bp`, `src/lsp_build.rs`, `src/kmeans.rs`, `src/bp.rs`	all three measured (§6): BP (the paper's partitioner) is the final configuration at both 5 M and 50 M

#	Task	Where	Status
7	Micro-optimisations	`src/lsp.rs`, `src/simd.rs`	c = 1 fast path, branchless bound loop, preloaded arrays, batched block bounds, global block order, adaptive β, AVX-512 dot product and block-bound sweep; follow-up: lazy block ordering and a prefetch pipeline in the sweep; packed 4-bit dense maxima measured neutral (default off) (§6)
8	Writeup, reproduction instructions	this document, `EMAIL.md`	done

3. Implementation design

The whole pipeline — inputs, index build, exact baseline, query path and evaluation — is drawn as one flowchart in Appendix D (Figure 11); the sections below describe each box.

3.1 On-disk LSP index (all little-endian, memory-mapped)

```
lsp-index/
  meta.json          { num_docs, vocab_size, total_postings, b, c, num_blocks,
                       num_superblocks, total_sb_entries, total_blk_entries,
                       weight_scale, dense_threshold,
                       doc_order: "corpus"|"dominant"|"kmeans"|"bp" }
  doc_order.bin      u32[num_docs]  new_pos -> original doc_idx (identity if not reordered)
  docids.bin         same string table as the baseline (in new order)
  fwd_offsets.bin    u64[num_docs+1]  start of each doc's postings in fwd_*.bin
  fwd_terms.bin      u16[total_postings]   term ids (sorted within doc)
  fwd_weights.bin    u8 [total_postings]   quantized weights
  lsp_terms.bin      per term: (sb_start u64, sb_len u32)
  sb_ids.bin         u32   superblock id             ┐ parallel arrays, contiguous
  sb_max.bin         u8    superblock max weight     | per term, sorted by sb id
  sb_blk_off.bin     u64   offset into blk_* arrays  ┘
  blk_local.bin      u8    block index within superblock  ┐ contiguous per (term, sb)
  blk_max.bin        u8    block max weight               ┘
```

```
dense_terms.bin    u64   per term: offset into dense_max.bin, or u64::MAX if sparse
dense_max.bin      packed 4-bit superblock maxima for terms in ≥ dense_threshold of superblocks
```

Design points:

- Sparse per-term lists (not a dense term × block matrix, which would be 49 k × 3 M = way too big). A term's superblock list only contains superblocks where the term appears.
- Index-forest layout: each superblock entry points to the contiguous run of its block entries, so after a superblock is selected, its block maxima for a term are one slice read — no second binary search. This is the paper's "compact data structure for fast superblock pruning" idea without SIMDBP.
- Quantisation: `w_q = round(w / w_max × 255)`, stored `u8`; dequantised once per query into a dense `f32[vocab]` query vector so scoring is `Σ q_dense[t] × w_q` with one multiply-add per posting. Block/superblock maxima are taken over the quantised values, so `bound ≥ score` holds exactly and pruning with η=1 is rank-safe w.r.t. the quantised scores.
- Forward index, flat arrays (the paper's "Flat-Inv"/"Fwd") — no nested `Vec`s (24 B overhead each × 50 M docs).
- Offsets are `u64`; total postings (2.58 B) would fit `u32` but not with headroom.

3.2 Index build (`scibench index-lsp data/v2/ --b 16 --c 16 [--reorder]`)

1. Parallel parse (reuse the baseline's rayon parse) into per-file flat arrays `(doc_offsets, terms u16, weights f32)`; record global `w_max`.
2. Optional reordering pass (see §3.4) producing a permutation.
3. Quantise weights, write forward index in (possibly permuted) order.
4. Block/superblock maxima: process superblocks in parallel chunks. Per chunk, keep a `u8[vocab]` scratch + touched-term list per block, take block max, then superblock max as max over blocks. Emit per-term partial lists per chunk; concatenate chunk lists in order (chunks are in superblock order, so per-term lists stay sorted without a sort).
5. Write `lsp_terms.bin` directory, `meta.json`. Log every phase's time and output size — these are deliverables (build time, index size).

Memory estimate: parse 2.58 B × (2+4) B ≈ 15 GB; block maxima ≈ #(term,block) pairs × 2 B; superblock entries × 13 B. Comfortably < 100 GB.

3.3 Query processing (LSP/0)

```rust
fn search_lsp(query: &Query, k: usize, gamma: usize, beta: f32, eta: f32) -> Vec<(f32, u32)> {
    let q_full: Vec<f32> = dense_from_terms(query, WEIGHT_SCALE);  // vocab length
    let q_prune: Vec<(u32, f32)> = top_k(&q_full, (beta * q_full.len() as f32).ceil() as usize);

    let mut sb_bound = vec![0.0; num_superblocks];  // reused buffer
```

```
    for (t, q_t) in &q_prune {
        for i in term_sb_range(*t) {
            sb_bound[sb_ids[i]] += q_t * sb_max[i];
        }
    }

    let mut cand = select_nth_unstable(&sb_bound, gamma);
    cand.sort_by(|&a, &b| sb_bound[b].partial_cmp(&sb_bound[a]).unwrap());

    let mut heap = BinaryHeap::new();
    let mut theta: f32 = 0.0;

    for sb_idx in &cand {
        if sb_bound[*sb_idx] <= theta { break; }

        let mut blk_bound = vec![0.0; blocks_per_sb];
        for (t, q_t) in &q_prune {
            let i = binary_search(&sb_ids[term_sb_range(*t)], *sb_idx);
            for j in sb_blk_range(i) {
                blk_bound[blk_local[j]] += q_t * blk_max[j];
            }
        }

        let mut block_order: Vec<_> = (0..blk_bound.len()).collect();
        block_order.sort_by(|&a, &b| blk_bound[b].partial_cmp(&blk_bound[a]).unwrap());

        for blk_idx in block_order {
            if blk_bound[blk_idx] <= theta / eta { break; }

            for doc_id in block(blk_idx) {
                let score: f32 = q_full.iter().zip(&fwd_index[doc_id]).map(|(q, w)| q * w).sum();
                heap.push((score, doc_id));
                if heap.len() > k { heap.pop(); }
                theta = heap.peek().map(|(s, _)| *s).unwrap_or(0.0);
            }
        }
    }
    heap.into_sorted_vec()  // best first; pos -> original doc_idx via doc_order -> doc_id
}
```

Cost model (b=16, c=16, γ=250): superblock pass ≈ Σ over pruned terms of their superblock-list length (bounded by 195 k each); block pass 250 × 16 × |q_prune|; scoring ≤ 250 × 256 = 64 k docs × 43 postings ≈ 2.8 M FMAs. Target: ≤ 5 ms p50 single-threaded, i.e. ≥ 1000× over the baseline.

LSP/1 (additionally visiting superblocks with `SBMax > θ/μ`) was not implemented; η at the block level is the only relaxation knob.

3.4 Document reordering

Bounds are only tight when a block's docs share vocabulary. Three orderings, in increasing cost:

1. Dominant-term bucketing (`--reorder dominant`, `src/lsp_build.rs`): sort docs by `(argmax term, 2nd argmax term)`. O(N log N), single pass, no clustering library; groups docs about the same topic surprisingly well for SPLADE (top term is usually the topical word).
2. Spherical k-means on sparse vectors (`--reorder kmeans`, `src/kmeans.rs`): centroids trained on an evenly strided sample (`--kmeans-sample`, default 1 M docs, `--kmeans-iters` Lloyd iterations), all docs assigned in parallel, then option 1 within each cluster. Closer to the paper.
3. Recursive graph bisection (`--reorder bp`, `src/bp.rs`, after PISA's implementation of Dhulipala et al. 2016) — what BMP/SP use. Implemented and unit-tested, but not measured on the real corpus within the time budget.

Measure (before → after): mean blocks visited, mean docs scored, recall@k, latency. Reordering only touches the permutation; the LSP code is unchanged.

3.5 Implementation best practices

- Keep the baseline untouched and byte-identical; the new code lives in `src/lsp.rs`, `src/lsp_build.rs` and `src/kmeans.rs`, selected by `--mode exact|lsp`. This makes the comparison honest and diffs reviewable.
- Unit tests on a tiny generated corpus (a few hundred docs, random vectors): assert LSP with `γ = ∞, β = 1, η = 1` returns exactly the quantised-exact top-k, and that every bound ≥ every score inside its block. This is the single most valuable test — it catches off-by-one errors in offsets, wrong `blk_local` mapping, and permutation bugs immediately.
- `--limit-files 2` dev loop on the instance (2 M docs, index in seconds) before any full-corpus build; full builds run under `nohup` with logs.
- Sampling caveat: `bench` shuffles the query file with a fresh RNG on every run and keeps the first `--num-queries`, so separate runs measure different queries; row-to-row differences of a few percent are sampling noise. A fixed, stored sample would remove this and is a cheap next step.
- Measure what you change: every LSP search returns `LspCounters` (superblocks visited, blocks visited, docs scored) and `bench` reports their mean and p90. These counters are what turn "it got slower" into "block pruning stopped triggering".
- No allocation in the hot loop: `sb_bound`, `blk_bound`, the heap and the dense query vector are per-searcher scratch buffers.
- Use `total_cmp` for float ordering, `u16` / `u8` narrowing only after a checked `assert!` at build time (vocab < 65 536, `c ≤ 256`).

- Commit after each task in §2 with the measured numbers in the commit message, so the history doubles as the decision log.

---

4. Evaluation methodology

4.1 Ground truth

The exact full-scan ranker is the oracle. `bench --mode lsp` opens both the exact index and the LSP index, and for every sampled query runs the exact search first (untimed), then the timed LSP search, and compares the two top-k sets. Nothing is cached to disk; at 50 M docs the exact pass dominates the wall-clock of a bench run but never enters the latency numbers.

4.2 Quality metric (LSP vs exact, per query, then averaged)

Table 5. Quality and efficiency metrics reported by `bench`.

Metric	Definition	Why
Recall@k (k = 10, 100)	`|LSP_k ∩ Exact_k| / |Exact_k|`, matched on doc id	The paper's headline metric ("recall preserved"); the exact ranker is the oracle

`bench` reports the mean, the minimum over queries, and the number of queries with recall < 1. Ties at the exact top-k boundary are not special-cased: a doc that the exact ranker placed at rank k+1 with an equal score counts as a miss, so reported recall is a slight underestimate. Rank-biased overlap, Kendall τ, the θ ratio and a failure-rate count were planned but not built.

4.3 Efficiency metrics

- Latency: single-threaded, one process per configuration; `bench` reports min / mean / median / p90 / p95 / p99 / max and QPS = N / total time (the full profile of every headline configuration is Table 13). Cold runs measure page-fault overhead on the memory-mapped index, so the first pass after a build is discarded. Multi-threaded QPS: `bench --threads N` replays the sample on N threads (independent queries, one scratch buffer each) after the timed single-thread pass (Table 12).
- Work counters (LSP mode): mean and p90 of superblocks visited, blocks visited and docs scored — these explain the latency numbers and are what changes with reordering.
- Index: `index-lsp` logs per-phase wall-clock, total build time and on-disk size per file. Peak RSS of the query process is measured with `/usr/bin/time -v` (Table 13).

4.4 Sweep performed

Each run = one configuration, one row, at k = 10 and k = 100:

Table 6. Parameter sweep performed.

Param	Values
γ	250, 1000 (50 M); 1000, 4000, 8000, 16000 (5 M slice)
β	0.5, 1.0
η	1.0 only
(b, c)	(16, 16), (16, 4), (16, 1) — each a separate index; b was not varied
ordering	corpus, dominant-term, k-means (k = 4096 and 16384)

The γ×β sweep was done on the default index first (same index, cheap), then c, then ordering. §6 reports the frontier and the two configurations that matter: the fastest one at ≈ 99 % recall and the rank-safe one.

4.5 Slices for analysis

Averages hide where pruning breaks. Slice every metric by:

Table 7. Per-query slices used in the analysis (results: Table 14, Figures 2–4).

Slice	How to derive	What it tells us
Query length (nnz): ≤ 10 / 11–30 / 31–60 / > 60	from the query vector	Long queries have looser bounds and benefit most from β; very short ones may lose recall under β
Query weight concentration: top-3 terms' share of Σq	from the query vector	Peaked queries prune well; flat ones are the hard case
Posting-list length of the heaviest query term: short / medium / long (terciles of df)	from `lsp_terms.bin`	Common terms make `SBMax` loose (the term is in every superblock)
Exact-score gap: $(s_1 - s_{10})/s_1$ small vs large	from ground truth	When the top-k is tightly bunched, small quantisation error swaps ranks
Top-k score magnitude: low-θ vs high-θ queries	from ground truth	Low absolute θ means the stop rule $\mathrm{SBMax} \le \theta$ rarely fires ⇒ latency tail

Slice	How to derive	What it tells us
Latency tail: the 5 % slowest queries	from the run	Inspect their counters: are they visiting all γ superblocks? no block pruning?
Rank position: recall of positions 1–3 vs 4–10 vs 11–100	from both lists	Does pruning lose the head or the tail of the list?
Result-set failures: queries with < k results	from the run	Which queries and why (all terms rare? β too low?)

`bench --per-query-csv` writes one row per query (query, nnz, top-3 weight share, heaviest term's superblock frequency, θ and score gap from the returned list, recall overall and by exact rank band, result count, latency, work counters, superblocks touched, entries read and the four stage timers), from which the slice tables above are derived. Results are in §6 (Table 14, Figures 2–4). `bench --trace-out` (with `--exact-cache`) is the second diagnostic: per query it records the bound-rank of the superblock holding each exact top-k doc and, for the first `--trace-full` queries, every visited superblock's bound and θ; Figures 7–8 are drawn from it and Figure 10 from the per-query CSVs.

---

5. Debugging playbook

Symptoms → first thing to check:

Table 8. Debugging playbook: symptom → first check.

Symptom	Check
Recall@k < 1 even with γ=∞, β=1, η=1	Quantisation only? Compare with an f32-rescored variant. If still < 1: bound < score somewhere ⇒ assert `blk_max ≥ w` in a debug build over the whole index.
Results are the right docs with wrong ids	`doc_order` permutation applied twice or not at all; test on `--limit-files 1` by comparing `docids` of exact vs LSP for one query.
Great recall@10, poor recall@100	γ too small for the depth, or the stop rule fires on θ for k=10 heap — make sure the heap size is the requested k.
Latency flat regardless of γ	The dense superblock pass dominates ⇒ increase `c` (fewer superblocks), lower β, or SIMD the pass. Counters will show `superblocks_visited` ≪ γ.

Symptom	Check
Latency tail (p99 ≫ p50)	Slice by max-df: queries dominated by a very common term. Consider capping per-term superblock list contribution or a min-weight cutoff.
Zero results for some queries	LSP failure mode: all superblocks had `SBMax ≤ θ` — should be impossible with θ starting at 0; if it happens the bound arrays are misaligned.
Build OOMs / is slow	Check `RAYON_NUM_THREADS`; per-chunk partial lists must be flushed, not accumulated per file.

Tooling: the unit tests in `src/lsp_build.rs` and `src/lsp.rs` cover the first two rows directly — `every_block_and_superblock_max_equals_the_true_max` walks a built index and asserts every stored maximum dominates its docs, and `lsp_without_pruning_matches_quantised_exact_topk` / `rank_safe_search_is_permutation_invariant` check that γ = all, β = 1, η = 1 reproduces the quantised exact top-k on every ordering. A per-query `--explain` mode that prints bounds against the actual best score was planned but not built.

---

6. Results (AWS r6i.8xlarge)

All runs use the v2 SPLADE corpus on the instance from §1.2, single-threaded queries, η = 1.0, `b = 16`, quality = recall of the exact top-k (the exact search runs untimed for every sampled query). Zero-weight padding postings (16.3 % of the raw index, Appendix A) are dropped by the LSP builder in every configuration. Figure 1 summarises the k = 10 rows of Tables 9–10 as a recall–latency frontier; the tables follow.

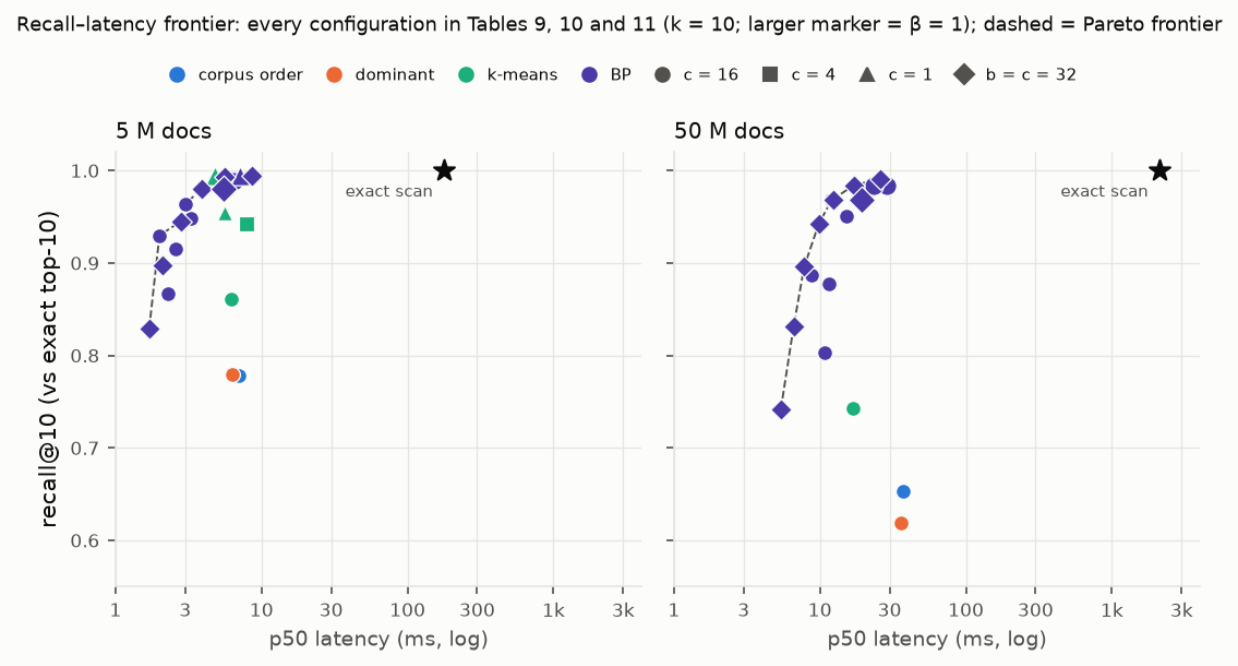

Figure 1. Recall–latency frontier: every k = 10 configuration in Tables 9, 10 and 11 (colour = ordering, marker = superblock size c, diamond = b = c = 32, larger marker = β = 1), the exact scan as the reference at recall 1.0, and the Pareto frontier dashed. Each successive ordering moves the frontier up and to the left; every frontier point at both scales is a BP index — c = 16 up to about 0.93, b = c = 32 above it.

5 M-document slice (every 10th doc; 1000 queries per row)

Every row uses the same 1000 queries (`--seed 42`) with the exact top-k cached (`--exact-cache`). Exact scan p50: 177 ms (k = 10), 138 ms (k = 100). Rows are in the order the work was done; each row is the best point of that configuration.

Table 9. 5 M-document slice: recall@k and latency per configuration (1000 queries per row).

ordering	c	k	γ	β	recall@k	p50	p90	speedup
corpus order	16	10	1000	0.5	0.778	7.0 ms	26 ms	25×
dominant term	16	10	1000	0.5	0.780	6.3 ms	24 ms	28×
k-means 4096	16	10	1000	0.5	0.861	6.2 ms	22 ms	29×
k-means 4096	4	10	4000	0.5	0.942	7.9 ms	28 ms	22×

ordering	c	k	γ	β	recall@k	p50	p90	speedup
k-means 16384	1	10	8000	0.5	0.954	5.6 ms	12 ms	32×
k-means 16384	1	10	8000	1.0	0.994	4.8 ms	15 ms	37×
BP	16	10	250	0.5	0.867	2.3 ms	8 ms	76×
BP	16	10	500	0.5	0.915	2.6 ms	10 ms	67×
BP	16	10	1000	0.5	0.948	3.3 ms	12 ms	54×
BP	16	10	2000	1.0	0.988	6.4 ms	24 ms	28×
BP	1	10	8000	1.0	0.994	7.1 ms	18 ms	25×
BP + adaptive β, global blocks, SIMD	16	10	500	0.5	0.930	2.0 ms	7 ms	89×
BP + adaptive β, global blocks, SIMD	16	10	1000	0.5	0.964	3.0 ms	11 ms	59×
k-means 4096	16	100	1000	0.5	0.844	5.0 ms	18 ms	28×
k-means 16384	1	100	8000	1.0	0.995	6.6 ms	24 ms	21×
BP	16	100	250	0.5	0.798	3.4 ms	10 ms	41×
BP	16	100	500	0.5	0.876	4.0 ms	13 ms	34×
BP	16	100	2000	0.5	0.967	6.1 ms	21 ms	23×
BP	1	100	8000	1.0	0.995	7.3 ms	25 ms	19×

Rank-safe ceiling (γ = all, β = 1): 0.997 / 0.995 — the residual is 8-bit quantisation. The exact-scan baseline is a separate 300-query sample.

50 M documents (1000 queries per row)

Same 1000-query sample and cached exact top-k. Exact scan p50 2.1 s, p90 12.3 s (100-query sample). Index builds: corpus order 198 s / 33.5 GB, dominant 171 s / 27.0 GB, k-means 399 s / 25.0 GB, BP 30 min / 20.9 GB. All rows are solo runs except the two marked ‡ (measured 4-way concurrent while the exact cache was being built).

Table 10. 50 M documents: recall@k and latency per configuration (1000 queries per row).

ordering	c	k	γ	β	recall@k	p50	p90	speedup
corpus order	16	10	1000	0.5	0.653	36.9 ms	154 ms	57×
dominant term	16	10	1000	0.5	0.619	35.7 ms	140 ms	59×
k-means 4096	16	10	1000	0.5	0.743	16.8 ms	62 ms	125×
BP	16	10	250	0.5	0.803	10.7 ms	38 ms	196×
BP	16	10	500	0.5	0.877	11.6 ms	41 ms	181×
BP ‡	16	10	2000	0.5	0.951	15.3 ms	53 ms	137×
BP	16	10	4000	1.0	0.984	28.5 ms	108 ms	74×
BP + adaptive β, global blocks, SIMD	16	10	500	0.5	0.887	8.8 ms	31 ms	239×
BP + adaptive β, global blocks, SIMD	16	10	2000	0.5	0.968	12.0 ms	40 ms	175×
BP + adaptive β, global blocks, SIMD	16	10	4000	1.0	0.984	23.5 ms	86 ms	89×
corpus order	16	100	1000	0.5	0.426	34.3 ms	112 ms	61×

ordering	c	k	γ	β	recall@k	p50	p90	speedup
k-means 4096	16	100	1000	0.5	0.684	19.6 ms	70 ms	107×
BP	16	100	250	0.5	0.684	12.7 ms	41 ms	165×
BP	16	100	500	0.5	0.786	14.2 ms	46 ms	148×
BP ‡	16	100	4000	1.0	0.963	36.2 ms	135 ms	58×

Larger blocks and superblocks: b = c = 32 (1000 queries per row)

The geometry sweep in Tables 9–10 only went *down* from c = 16. Going up — b = 32 docs per block, c = 32 blocks per superblock, i.e. 1024-doc superblocks and 4× fewer of them (48,829 at 50 M, 4,883 at 5 M) — was measured last, with the BP ordering reused from the c = 16 index (`--reorder from-index`, so the build is 100 s at 50 M instead of 30 min; BP is coherent at every scale, so 32-doc blocks are unions of adjacent 16-doc leaves). Same 1000-query sample, adaptive β, global block order, SIMD; the last column is the c = 16 row of Tables 9–10 at the same γ where one exists. Index size 17.1 GB at 50 M (20.9 GB at c = 16) and 1.8 GB at 5 M.

Table 11. b = c = 32 with the BP ordering, k = 10 (1000 queries per row; speedup = exact-scan p50 / p50).

corpus	γ	β	recall@10	p50	p90	speedup	c = 16 at the same γ (Tables 9–10)
5 M	125	0.5	0.829	1.7 ms	7 ms	102×	—
5 M	250	0.5	0.897	2.1 ms	9 ms	83×	
5 M	500	0.5	0.945	2.8 ms	12 ms	64×	0.930 @ 2.0 ms
5 M	1000	0.5	0.980	3.9 ms	14 ms	46×	0.964 @ 3.0 ms
5 M	2000	0.5	0.993	5.6 ms	18 ms	32×	0.988 @ 6.4 ms (β = 1, pre-SIMD)
5 M	4000	0.5	0.994	8.6 ms	24 ms	21×	

corpus	γ	β	recall@10	p50	p90	speedup	c = 16 at the same γ (Tables 9–10)
5 M	1000	1.0	0.980	5.5 ms	24 ms	32×	β = 1 adds nothing
50 M	125	0.5	0.742	5.4 ms	19 ms	389×	
50 M	250	0.5	0.831	6.6 ms	23 ms	321×	
50 M	500	0.5	0.896	7.8 ms	30 ms	270×	0.887 @ 8.8 ms
50 M	1000	0.5	0.942	9.8 ms	37 ms	215×	
50 M	2000	0.5	0.969	12.3 ms	46 ms	171×	0.968 @ 12.0 ms
50 M	4000	0.5	0.984	17.0 ms	58 ms	123×	0.984 @ 23.5 ms (β = 1)
50 M	8000	0.5	0.991	26.1 ms	82 ms	81×	—
50 M	2000	1.0	0.969	19.2 ms	80 ms	110×	β = 1 adds nothing

Three things change with 1024-doc superblocks:

- The bound pass halves and the top-γ selection becomes negligible: 2.4 M superblock entries per query instead of 4.8 M (Appendix C), 3.0 ms instead of 6.3 ms at 50 M, and a select over 49 k instead of 195 k candidates.
- Scoring grows: the bounds of a 1024-doc superblock are looser, so more blocks survive — the mean query scores 79 k docs at γ = 500 (0.16 % of the corpus) against 16 k with c = 16, and the visit stage goes from 1.5 to 5.5 ms. At γ = 8000 the two effects balance at 26 ms.
- β = 1 for every query stops helping (0.9412 vs 0.9421 at γ = 1000; 0.9687 vs 0.9689 at γ = 2000), where at c = 16 it was worth +1–2 points over adaptive β. Adaptive β itself (β = 1 only for queries of ≤ 60 terms) still matters: the same sweep without it is 1–1.6 points lower (0.968 instead of 0.984 at γ = 4000, Figure 6). For a long query the heavy terms already decide the order of 1024-doc superblocks; adding the light half raises every bound by about the same amount and never tightens θ enough to stop earlier — it just doubles the bound pass.

Net: at the fast point the two geometries are within a point and a millisecond of each other (0.896 @ 7.8 ms vs 0.887 @ 8.8 ms at 50 M); at the 99 % point b = c = 32 wins clearly — 0.984 at 17.0

ms where c = 16 needed 23.5 ms and β = 1, and 0.991 at 26 ms, the paper's recall level, which c = 16 did not reach at any measured setting. The same holds at 5 M (0.993 @ 5.6 ms vs 0.988 @ 6.4 ms). Figure 1 and Figure 6 include these points.

Follow-up: lazy block ordering, prefetching and b = 16, c = 64

A second pass over the query path after the report was first written (`docs/experiments/2026-09-21-overnight.md` has every run; 12 ideas, 4 adopted). Same sample, same caches, same machine; every row is a solo warm run under `/usr/bin/time -v`. Three changes (the first one in two steps) are now the defaults:

1. Lazy block ordering. The global traversal (item 5b below) sorted every candidate block by bound before visiting — γ × c entries, 256 k at b16c64 / γ = 4000 — of which a few hundred are visited before the stop rule fires. The top 8192 blocks are now selected (quickselect) and sorted, and the rest are heapified only if the stop rule has not fired by then. Same traversal order, bit-identical results; −10 % to −40 % p50 depending on γ × c.
2. Prefetch pipeline in the block-bound sweep (`src/simd.rs`). A hit in the sweep costs two dependent cache misses (its run-directory entry, then its block run); a two-stage ring prefetches both 16 hits ahead. Bit-identical; −9 % to −17 % on that stage.
3. b = 16, c = 64. With ordering no longer penalising large c, the geometry that keeps b = c = 32's cheap bound pass (48 k superblocks, ~3 ms at 50 M) but c = 16's tight 16-doc blocks (22 k docs scored per query instead of 120 k) is on or above every other curve from γ = 500 up (Figure 1's frontier moves left by 1.5–2.5 ms at 50 M).

Table 12. Final configurations with the follow-up defaults (k = 10, 1000 queries, single thread; speedup = exact-scan p50 / p50, 2.1 s at 50 M and 177 ms at 5 M). Throughput: the same sample replayed on N threads, independent queries.

corpus	index	γ	recall@10	p50	p90	p99	speedup	docs scored	peak RSS	QPS 1 / 16 / 32 thr
50 M	b16c64	250	0.831	4.7 ms	18.0	30.1	448×	16.3 k	13.3 GB	
50 M	b16c64	500	0.896	5.6 ms	21.0	35.9	374×	19.1 k	13.4 GB	
50 M	b16c64	1000	0.942	6.1 ms	23.0	38.3	344×	20.8 k	13.5 GB	
50 M	b16c64	2000	0.968	7.8 ms	28.6	47.6	269×	21.6 k	13.5 GB	

corpus	index	γ	recall@10	p50	p90	p99	speedup	docs scored	peak RSS	QPS 1 / 16 / 32 thr
50 M	b16c64	4000	0.983	10.9 ms	39.9	64.5	193×	21.8 k	13.5 GB	
50 M	b16c64	8000	0.990	16.1 ms	57.2	90.8	130×	21.9 k	13.5 GB	
50 M	b32c32	500	0.896	7.8 ms	29.8	45.9	269×	78.5 k	14.5 GB	
50 M	b32c32	4000	0.984	12.7 ms	54.6	97.3	166×	120.7 k	14.6 GB	39 / 470 / 628
50 M	b32c32	8000	0.991	17.1 ms	69.7	126	123×	123.3 k	14.6 GB	
50 M	b32c64	4000	0.987	13.8 ms	57.5	110	152×	122.4 k	12.3 GB	
50 M	c16	500	0.887	8.4 ms	30.0	48.3	250×	15.6 k	18.2 GB	74 / 888 / 1077
50 M	c16	2000	0.968	9.9 ms	35.3	56.9	213×	20.7 k	18.3 GB	
5 M	b16c64	250	0.897	1.33 ms	5.6	9.4	133×	9.4 k	1.8 GB	
5 M	b16c64	500	0.943	1.81 ms	7.0	11.7	98×	9.9 k	1.8 GB	
5 M	b16c64	1000	0.978	2.35 ms	8.5	13.9	75×	9.8 k	1.8 GB	
5 M	c16	500	0.930	1.76 ms	7.0	11.3	101×	9.0 k	2.2 GB	
5 M	c16	2000	0.985	2.91 ms	10.2	16.4	61×	9.8 k	2.2 GB	

corpus	index	γ	recall@10	p50	p90	p99	speedup	docs scored	peak RSS	QPS 1 / 16 / 32 thr
5 M	b32c64	500	0.966	2.73 ms	12.9	24.4	65×	45.4 k	1.7 GB	
5 M	b32c64	1000	0.989	3.36 ms	14.4	27.7	53×	46.7 k	1.7 GB	
5 M	b32c32	500	0.945	2.50 ms	12.3	21.7	71×	42.7 k	1.8 GB	
5 M	b32c32	2000	0.993	3.99 ms	16.8	30.8	44×	47.0 k	1.8 GB	
5 M	b32c64	2000	0.9935	4.52 ms	17.9	33.3	39×	47.2 k	1.7 GB	

Against Tables 11 and 13: the 50 M fast point moves from 0.896 @ 7.8 ms to 0.896 @ 5.6 ms (and 0.831 @ 4.7 ms is a new point), the 99 % point from 0.991 @ 26 ms to 17 ms; at 5 M the 99 % point from 0.993 @ 5.6 ms to 4.0 ms and the fast point from 0.945 @ 2.8 ms to 0.943 @ 1.8 ms. Peak RSS here is lower than Table 13's because the block-entry files stay memory-mapped and only the touched pages count. Throughput is linear to 16 threads (the 16 physical cores: per-thread latency 13.1 → 14.3 ms from 4 to 16 threads, so memory bandwidth is not binding) and the hyperthreads add another 1.3×.

Measured and not adopted in the follow-up, each on the same sample (details and tables in the experiment log): progressive block bounds (the superblock bound is too loose to shed candidates after a first phase; +20 % on the sweep), a larger γ for short queries only (+0.8 pt for +30 % p50; raising γ for everyone dominates), η < 1 (−3 pt for −10 % latency) and η > 1 or all-term block bounds (+0.01 pt for +40–150 % latency — the block-level prune is *not* where the residual recall goes), impact-ordered superblock lists with entry-level pruning (the id order is load-bearing: both sweeps read entries in address order, and sorting by max costs more locality than the skipped entries save), and skipping frequent terms in the bound pass (5.7 → 2.5 ms but 0.968 → 0.916: frequent SPLADE terms still discriminate). Every cheaper-bound-pass idea has now lost to plain β on this corpus.

Full latency profile and memory (final re-run)

Tables 9, 10 and 11 report p50 / p90 because that is what moved during development. For the record, every headline configuration was re-run once more at the end, solo, under `/usr/bin/time -v`, with the complete latency distribution, throughput and peak resident memory. Same 1000-

query sample (seed 42) and cached exact top-10; the exact scans use the first 100 (50 M) and 300 (5 M) queries of the same sample, so their medians differ from the random-sample figures quoted earlier (2.1 s and 177 ms) — the speedups in this table are within-table. QPS is single-threaded (1 / mean latency); the machine has 32 vCPUs and the query path shares nothing, so throughput scales with threads until memory bandwidth binds.

Table 13. Full latency profile, throughput and peak RSS of the headline configurations (k = 10, single thread).

corpus	configuration	queries	recall@10	p50	p90	p95	p99	QPS	speedup (p50)	peak RSS
50 M	exact scan (baseline)	100	1.000 (reference)	2.30 s	10.84 s	11.45 s	16.45 s	0.3	1×	7.9 GB
50 M	LSP c = 16, γ = 500, β = 0.5	1000	0.887	8.9 ms	30.9 ms	38.3 ms	50.6 ms	74.4	258×	17.4 GB
50 M	LSP c = 16, γ = 2000, β = 0.5	1000	0.968	11.9 ms	39.4 ms	48.1 ms	63.9 ms	56.9	193×	17.5 GB
50 M	LSP c = 16, γ = 4000, β = 1	1000	0.984	24.0 ms	88.2 ms	103.6 ms	140.2 ms	27.8	96×	17.8 GB
50 M	LSP b = c = 32, γ = 500, β = 0.5	1000	0.896	7.9 ms	30.0 ms	36.9 ms	46.1 ms	81.8	292×	16.6 GB
50 M	LSP b = c = 32, γ = 4000, β = 0.5	1000	0.984	16.9 ms	57.6 ms	73.8 ms	97.2 ms	39.3	136×	16.6 GB
50 M	LSP b = c = 32, γ = 8000, β = 0.5	1000	0.991	26.0 ms	82.2 ms	100.5 ms	132.2 ms	26.8	88×	16.6 GB
5 M	exact scan (baseline)	300	1.000 (reference)	106.2 ms	460.6 ms	621.8 ms	862.4 ms	5.2	1×	1.6 GB
5 M	LSP c = 16, γ = 500, β = 0.5	1000	0.930	2.0 ms	7.3 ms	8.9 ms	11.6 ms	317.5	53×	2.1 GB

corpus	configuration	queries	recall@10	p50	p90	p95	p99	QPS	speedup (p50)	peak RSS
5 M	LSP c = 16, γ = 1000, β = 0.5	1000	0.964	2.7 ms	8.9 ms	11.0 ms	14.3 ms	250.9	40×	2.1 GB
5 M	LSP c = 16, γ = 2000, β = 1	1000	0.988	5.6 ms	19.4 ms	23.7 ms	31.4 ms	121.5	19×	2.2 GB
5 M	LSP b = c = 32, γ = 500, β = 0.5	1000	0.945	2.8 ms	11.9 ms	16.0 ms	19.6 ms	212.2	38×	1.8 GB
5 M	LSP b = c = 32, γ = 2000, β = 0.5	1000	0.993	5.6 ms	18.3 ms	23.1 ms	31.5 ms	122.5	19×	1.8 GB

Reading it:

- Tail vs median. p99 is 5–6× p50 in every LSP row (Figure 3: latency is query length), and 7× for the exact scan; the exact scan's p99 is 16 s. The pruned ranker's 99th percentile at the 99 % recall point (132 ms) is below the exact scan's *median* by 17×.
- Memory. The LSP query process holds the pruning structure and the forward index on the heap (they are preloaded, §6 lever 4), so peak RSS ≈ index size: 16.6–17.8 GB at 50 M, 1.8–2.2 GB at 5 M, and b = c = 32 is ~1 GB smaller than c = 16 (Appendix C). The exact scan memory-maps its 19 GB posting file and only pages in what 100 queries touch (7.9 GB); a long-running exact service would converge on the full file. Both fit the 247 GB machine many times over; a 32 GB box would still hold the 50 M LSP index.
- Index size and build time (§6, Appendix C): 33.5 GB / 198 s in corpus order, 20.9 GB / 32 min with BP (the ordering is the cost), 17.1 GB / 100 s for b = c = 32 when the ordering is reused; the exact index is 22.1 GB / 65 s.

Slice analysis

Per-query slices (same 1000-query sample) for the two 50 M configurations, BP order, c = 16, k = 10. Recall is per slice; latency is p50.

Table 14. Slice analysis at 50 M docs, k = 10: recall per slice @ p50 latency for the two operating points.

slice	queries	γ = 4000, β = 1 (overall 0.984, 27.6 ms)	γ = 500, β = 0.5 (overall 0.877, 11.4 ms)
query length ≤ 10 terms	2 %	0.990 @ 3.7 ms	0.695 @ 1.9 ms
11–30	18 %	0.990 @ 5.0 ms	0.879 @ 2.6 ms
31–60	13 %	0.984 @ 11.5 ms	0.866 @ 4.8 ms
61–200	24 %	0.978 @ 23.4 ms	0.884 @ 9.5 ms
201–500	23 %	0.983 @ 51.3 ms	0.892 @ 19.9 ms
> 500	20 %	0.987 @ 107.7 ms	0.876 @ 40.4 ms
weight concentration: flat / medium / peaked	⅓ each	0.984 / 0.981 / 0.988 @ 82 / 28 / 6.8 ms	0.880 / 0.890 / 0.862 @ 32 / 11 / 3.2 ms
θ (k-th score): low / medium / high	⅓ each	0.981 / 0.984 / 0.987 @ 10 / 27 / 81 ms	0.824 / 0.902 / 0.905 @ 4.3 / 11 / 32 ms
score gap $(s_1 - s\_k)/s_1$: bunched / spread	⅓ / ⅓	0.986 / 0.983	0.901 / 0.851
slowest 5 % of queries (mean 1,160 terms)	5 %	0.986 @ 149 ms	0.836 @ 56 ms
recall of exact positions 1–3 / 4–10	—	0.994 / 0.980	0.928 / 0.856
queries with < k results or recall 0	—	0 / 0	0 / 0

What the slices say:

- Latency is query length. p50 rises from 3.7 ms at ≤ 10 terms to 108 ms at > 500 terms (γ = 4000) — a power law, latency ∝ nnz^0.75 (Figure 3); the slowest 5 % average 1,160 terms. Docs scored rise with it (2 k → 107 k) because more terms mean higher θ but also looser bounds, so more blocks survive pruning. Weight concentration is the same effect seen from the other side: peaked queries (top-3 terms ≥ 23 % of Σq) run 12× faster than flat ones.
- At the 99 % operating point recall is uniform (0.978–0.991 across every slice); there is no query class that pruning fails on. The loss sits at the tail of the list (positions 4–10: 0.980) rather than the head (1–3: 0.994).

- At the fast operating point the loss is structured. Very short queries (≤ 10 terms, 0.695), low-θ queries (0.824) and spread score distributions (0.851) lose most: with β = 0.5 a 10-term query keeps 5 terms, and a low k-th score means the stop rule `SBMax ≤ θ` never fires, so the γ budget alone decides recall. These are the queries for which β = 1 or a larger γ is worth its cost; a per-query rule (β = 1 when nnz ≤ 30) would recover most of the gap at no cost for the long queries that dominate latency.
- The heaviest-term slice is uninformative on this corpus: 85 % of queries contain a term present in ≥ 72 % of superblocks, so the terciles collapse.

The 5 M slice behaves the same way (γ = 500, β = 0.5: ≤ 10-term queries 0.720, slowest 5 % 0.842, positions 1–3 vs 4–10 0.937 / 0.906; γ = 2000, β = 1: 0.98–0.995 everywhere).

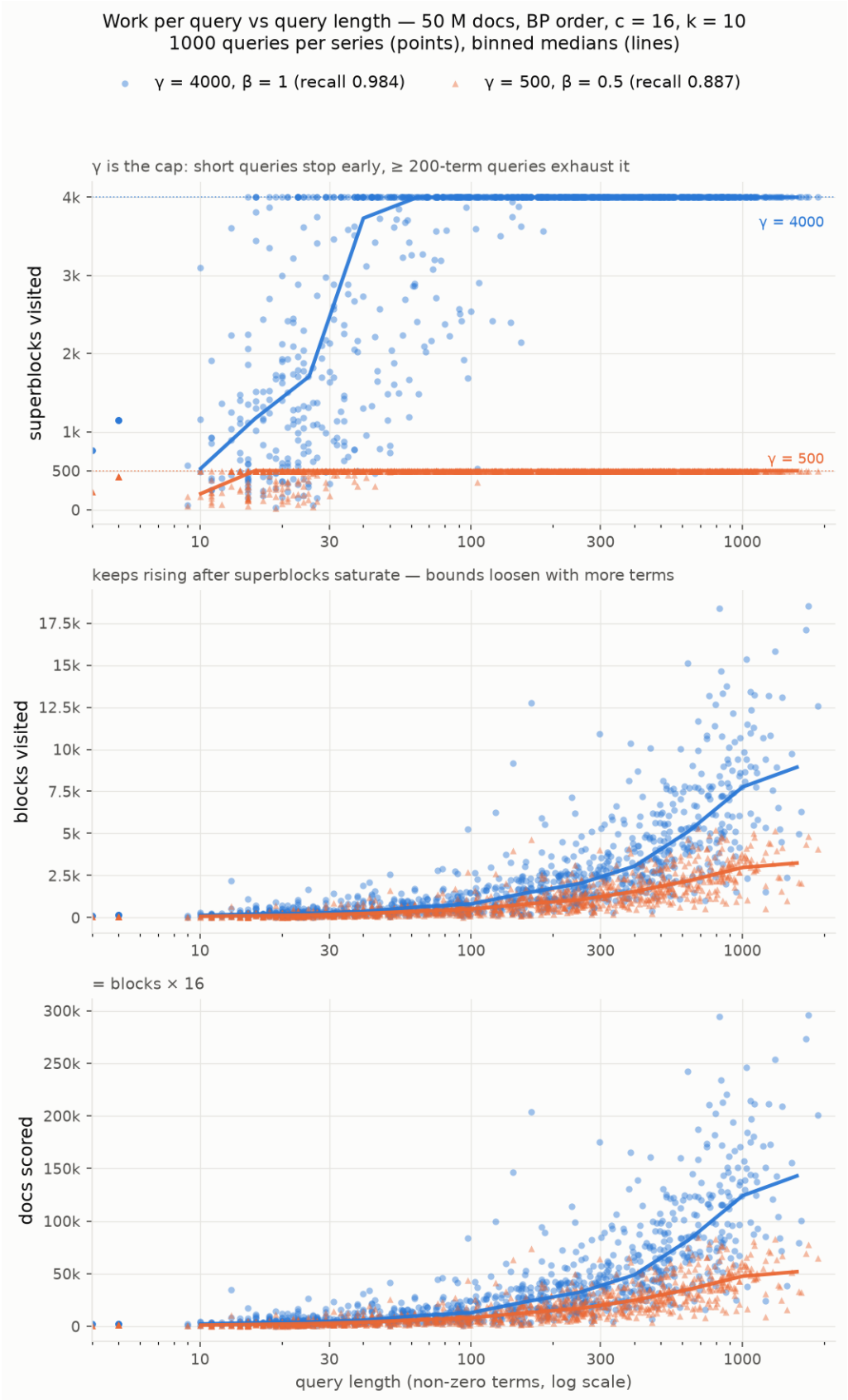
Work per query vs query length — 50 M docs, BP order, c = 16, k = 10
1000 queries per series (points), binned medians (lines)
γ = 4000, β = 1 (recall 0.984)
γ = 500, β = 0.5 (recall 0.887)
γ is the cap: short queries stop early, ≥ 200-term queries exhaust it
superblocks visited
4k
3k
2k
1k
500
0
γ = 4000
γ = 500
10
30
100
300
1000
keeps rising after superblocks saturate — bounds loosen with more terms
blocks visited
17.5k
15k
12.5k
10k
7.5k
5k
2.5k
0
= blocks × 16
docs scored
300k
250k
200k
150k
100k
50k
0
query length (non-zero terms, log scale)

Figure 2. Per-query work vs query length (log scale), 50 M docs, BP order, c = 16, k = 10, 1000 queries per series; lines are binned medians. Superblocks visited hit the γ cap at ~50 terms (γ = 4000) or ~15 terms (γ = 500) — the θ stop rule only ends the search early for short queries — while blocks visited and docs scored keep rising with query length because the block bound (a sum over query terms) loosens relative to θ, so fewer blocks are pruned inside each visited superblock. Spearman ρ with query length: superblocks 0.71, blocks 0.88, docs 0.88, latency 0.99 (γ = 4000). The vertical spread at a given length is weight concentration: peaked queries sit at the bottom of the cloud, flat ones at the top.

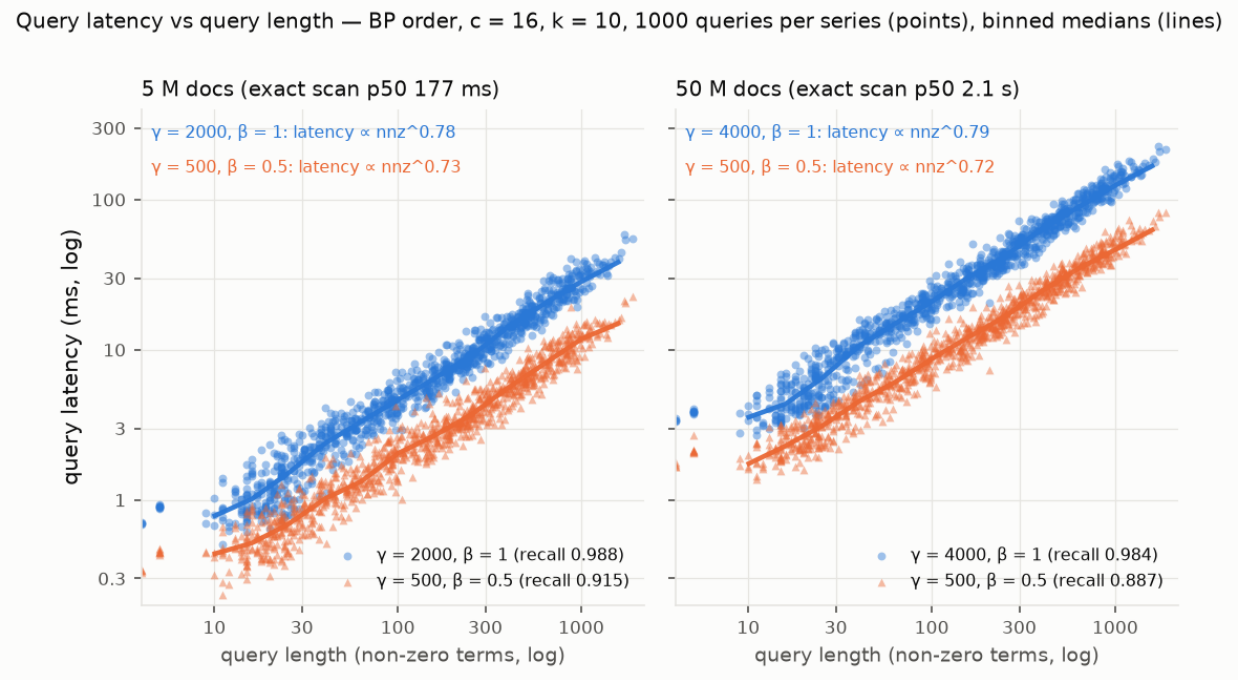

Figure 3. Query latency vs query length on log–log axes, 5 M (left) and 50 M (right), same runs. Both operating points are straight lines: latency ∝ nnz^0.72–0.79, i.e. sub-linear in query length, with the two operating points separated by a constant ~3× at every length. Going from 5 M to 50 M docs shifts the curve up ~4× (not 10×): the superblock/block work grows with the corpus, the per-block scoring does not. The residual spread at a given length (~2× between the bottom and top of the cloud) is weight concentration.

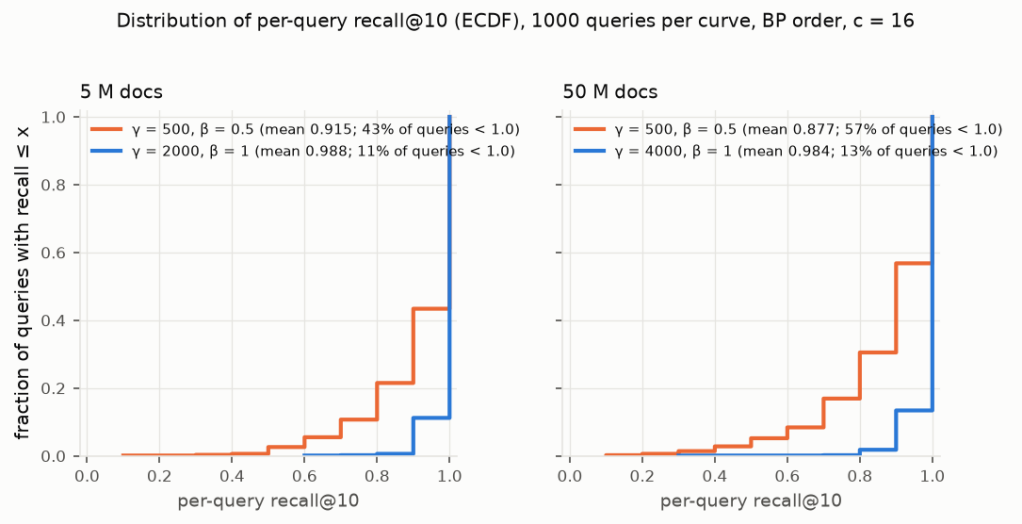

Figure 4. Distribution of per-query recall@10 (ECDF) behind the means in Table 10. At the fast operating point (γ = 500, β = 0.5) 43 % (5 M) and 57 % (50 M) of queries miss at least one of their top-10 documents, most of them exactly one; at the 99 % point 87–89 % of queries are perfect and no query falls below 0.7.

What moved the numbers

1. Reordering (corpus → dominant → k-means → BP). With unclustered blocks the superblock bound `SBMax`, summed over 141–282 query terms and 256 unrelated docs, is nearly flat, so the top-γ superblocks are largely misses — recall 0.64–0.79 however large γ is. Recursive graph bisection (the paper's block assignment, `src/bp.rs`, 129 s at 5 M / 30 min at 50 M) makes 256-doc superblocks coherent: 0.79 → 0.95 at the same γ and the smallest index (81 M vs 104 M superblock entries at 5 M).

Figure 5. Recall@10 vs γ for each ordering (c = 16, β = 0.5, fixed 1000-query sample). Ordering shifts the whole curve: at the paper's γ = 250–500 (shaded) BP reaches 0.86–0.91 (5 M) and

0.80–0.87 (50 M) where corpus order reaches 0.56–0.67 and 0.46–0.55. All curves are still rising at γ = 4000; with β = 1 (not shown) BP reaches 0.99.

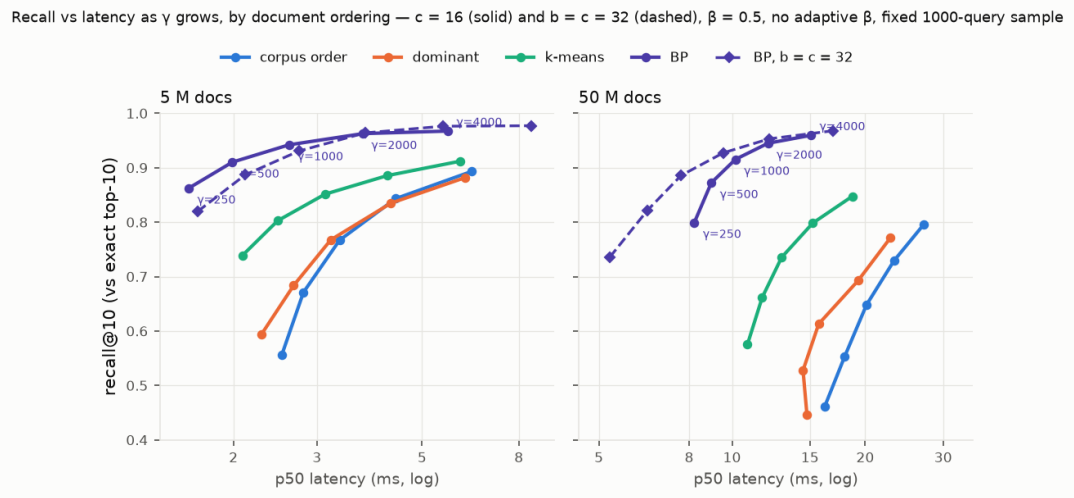

Figure 6. The same sweep as recall against p50 latency, γ = 250 → 4000 along each curve (dashed: the b = c = 32 index of Table 11 under the same protocol, γ = 125 → 4000). Ordering moves the whole curve, not one point: at 50 M, BP at γ = 250 (0.80 @ 8.2 ms) matches corpus order at γ = 4000 (0.80 @ 27.2 ms) at 3.3× lower latency, and at 5 M BP at γ = 500 (0.91 @ 2.0 ms) is above corpus order's best (0.89 @ 6.4 ms). Latency also falls with ordering at equal γ, because coherent superblocks have shorter per-term lists. The b = c = 32 curve sits on or just above the c = 16 curve at equal latency; its advantage in Table 11 comes at the top end, with adaptive β and γ ≥ 4000.

1. Superblock size. Before BP, shrinking c (16 → 4 → 1) was the only way to tight bounds: 0.86 → 0.94 → 0.96 at matched doc budget. With BP, c = 16 is both the fastest (bound pass over 19.5 k superblocks ≈ 1 ms) and within 0.2 points of c = 1.

2. β = 1. Partial bounds (β < 1) make θ-pruning lossy; β = 1 closes the last 3–4 points to 0.99 for +1–2 ms.

3. Query-path engineering, each verified bit-identical by the exactness tests: a c = 1 fast path (block bound = superblock bound), a branchless bound loop with heap-preloaded arrays (4.3 → 1.6 ns per entry), and batched block bounds (one sweep routing entries to candidate slots instead of ~560 k binary searches per query: 37 → 5 ms at c = 16).

4. Query-time refinements from the slice analysis. (a) *Adaptive β*: queries with ≤ 60 terms use all their terms for bounds (the slices showed β < 1 loses most on short, low-θ queries while long queries are insensitive): +1.5–2 points at the fast operating point for no latency cost (50 M, γ = 2000: 0.951 → 0.968). (b) *Global block order*: with c > 1, all candidate blocks are visited in one descending block-bound order (BMP's traversal over the candidate set) with the stop rule on the block bound; recall-neutral, 10–15 % fewer docs scored. (c) *Capping bound terms* at the 64 heaviest was tried and rejected: p90 falls 4× but recall drops 6–10 points — the low-weight

tail of a 282-term query carries real bound mass, so the cap is lossy in the same way $\beta < 1$ is. All three are flags; (a) and (b) are the defaults.

5. AVX-512 kernels (`src/simd.rs`, runtime-dispatched, scalar fallback): the per-doc dot product (16-lane gather of the dense query by term id, FMA, reduce) and the block-bound sweep (gather 16 candidate slots, compare, process only the hits). Bounds are bit-identical; scores differ only by summation order (the tests' brute force uses the same kernel). 11–20 % end to end: 50 M, $\gamma = 2000$: 14.1 → 12.0 ms; $\gamma = 500$: 10.9 → 8.8 ms. The remaining block-bound cost is in the hit entries' block runs (scalar, data-dependent), not in scanning the misses.

6. Ordering refinements — measured, none adopted. BP with 40 swap iterations (−1 recall point: the log-gap objective is not the max-bound objective), leaf size 8 (identical), $c = 8$ (+1.6 points at equal block budget, +20 % latency). BP at $c = 16$ is at its ceiling for this corpus.

7. Larger geometry, $b = c = 32$ (Table 11): 4× fewer superblocks halve the bound pass and make $\beta = 1$ unnecessary beyond adaptive β, at the price of scoring 5× more docs. Neutral at the fast point, +6.5 ms saved at the 99 % point (0.984 @ 17.0 ms) and a new best of 0.991 @ 26 ms at 50 M. Item 2's "$c = 16$ is fastest" was true of the bound pass only; once the block-bound sweep and scoring are vectorised, the balance point moves to bigger superblocks.

8. Lazy block ordering and prefetching (follow-up): ordering only the blocks that get visited (quickselect of the top 8192, heap for the rest) and prefetching the block-bound sweep's hit runs; bit-identical, −10 % to −40 % p50 (Table 12).

9. $b = 16$, $c = 64$ (follow-up): the cheap bound pass of 1024-doc superblocks with 16-doc blocks; the new fast point at 50 M (0.896 @ 5.6 ms, 0.831 @ 4.7 ms).

Why reordering works. Both operating points are gated by where the exact answers sit in the bound order. With `--trace-out` (a diagnostic that records, per query, the bound-rank of the superblock holding each exact top-10 doc) the picture is direct: under BP 89 % of the answers are inside the first 500 superblocks and 99 % inside the first 4000; under corpus order 60 % and 87 % (Figure 7). The remainder is what γ cannot recover, and it is why the $\beta = 1$ / $\gamma = 4000$ point reaches 0.98 with BP and 0.87 without it. Figure 8 shows the mechanism on one short query: with coherent superblocks the answers surface early, θ rises fast, and the stop rule fires at 471 superblocks — before $\gamma = 500$ — with all ten answers found; in corpus order the same query finds seven of them inside $\gamma = 500$ and runs on for 2692 superblocks.

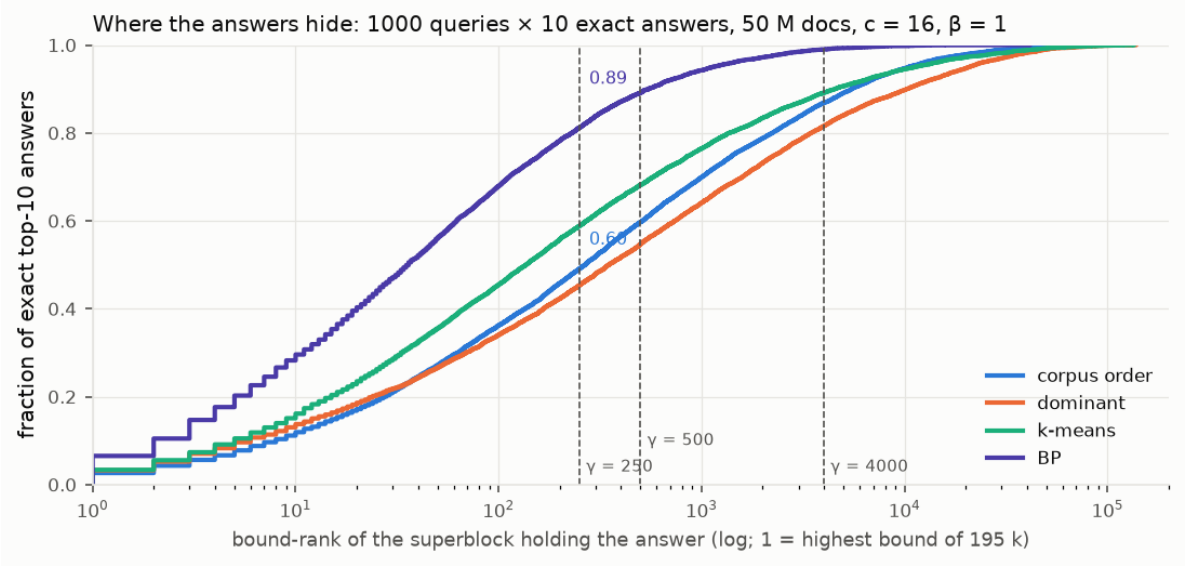

Figure 7. Where the answers hide: CDF of the bound-rank of the superblock holding each exact top-10 doc, over 1000 queries × 10 answers (50 M, c = 16, β = 1 so every answer is touched). Fraction of answers inside γ = 250 / 500 / 4000: BP 0.81 / 0.89 / 0.99, k-means 0.59 / 0.68 / 0.89, corpus order 0.49 / 0.60 / 0.87, dominant 0.45 / 0.55 / 0.82.

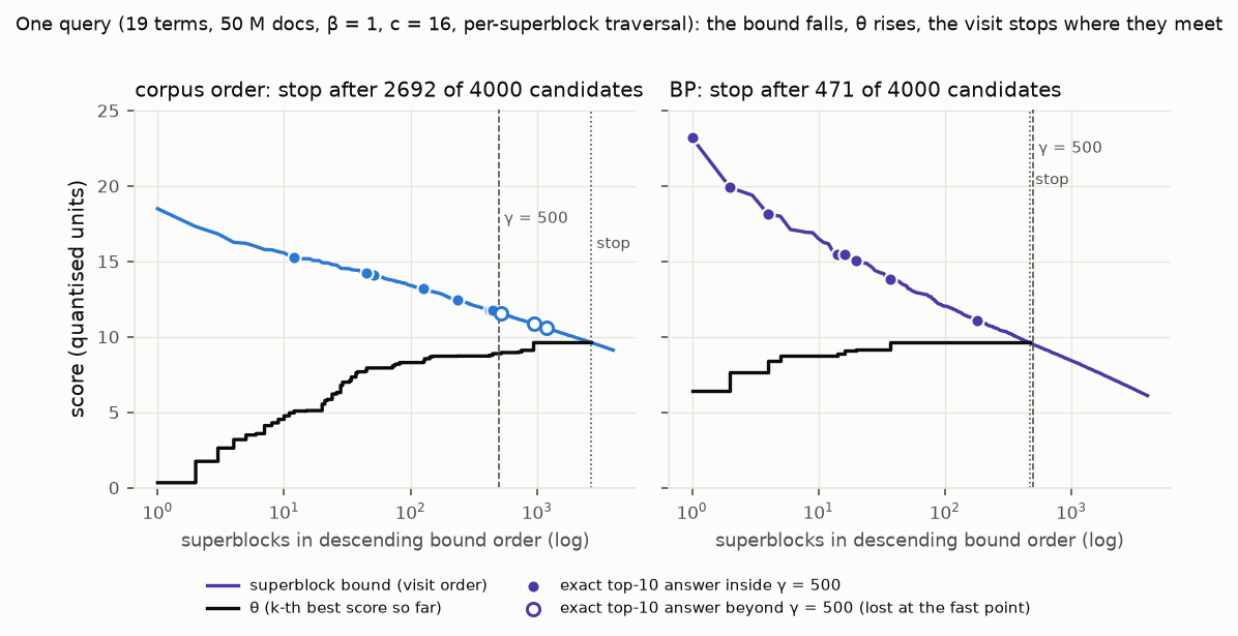

Figure 8. The stop rule on one query (19 terms, 50 M docs, β = 1, γ = 4000, per-superblock traversal): superblock bounds in visiting order (falling), θ = current 10th-best score (rising), and the superblock of each exact top-10 answer (filled inside γ = 500, hollow beyond). BP: all ten answers within the first 180 superblocks, visit stops at 471. Corpus order: answers spread to rank 1191, three of them beyond γ = 500, and the bound stays above θ until superblock 2692.

Measured and not adopted. The paper's packed 4-bit dense maxima (superblock rows swept with an integer-accumulating vectorised tile loop; block rows for index-forest random access; `--dense-threshold`, rank-safe via `17 × max4 ≥ max8`) were implemented and are neutral here: a dense sweep costs ~40 µs per term at 312 k superblocks, the same as the sparse lists it replaces, and block rows are neutral at threshold 0.5 and harmful at 5 % (half the vocabulary becomes dense at c = 16). The flag defaults to off. They pay off in the paper's regime — 34 k superblocks and ~13 pruned query terms — not in ours.

Where the remaining time goes (stage timers, 50 M, k = 10, γ = 4000, β = 1, SIMD): bound pass 11 ms over 195 k superblocks, block-bound sweep 18 ms, scoring 5 ms. Both sweeps are linear in query terms × list length for 282-term queries (5–10× longer than MS MARCO's); the term cap that would shorten them was shown to be lossy (item 5c). What is left is structural: a positional layout for sparse terms' block entries would remove the re-sweep, and shorter queries (the paper's regime) would shrink everything proportionally.

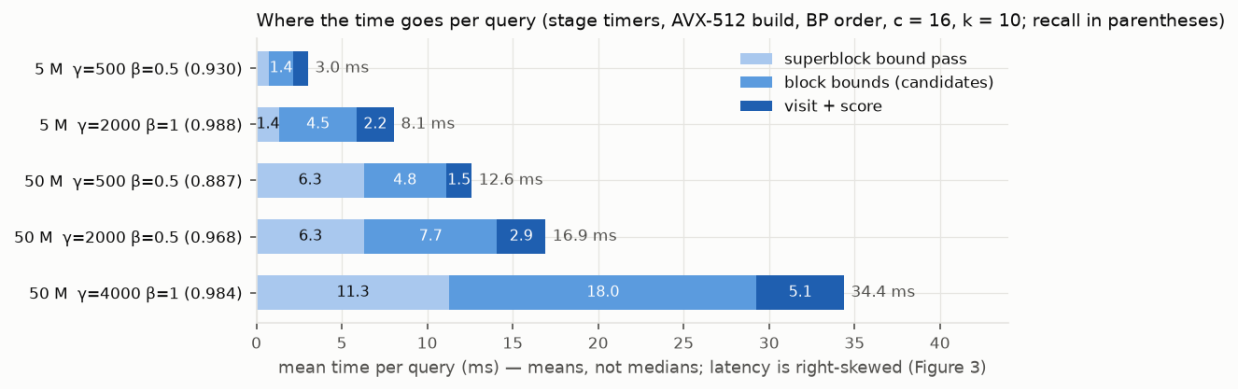

Figure 9. Per-query time by stage (means from the stage timers; the medians in Tables 9–10 are lower because latency is right-skewed, Figure 3). The superblock bound pass is ~1 ms at 5 M and 6–11 ms at 50 M (195 k superblocks); the candidate block-bound sweep dominates as γ grows; scoring stays below a third of the total.

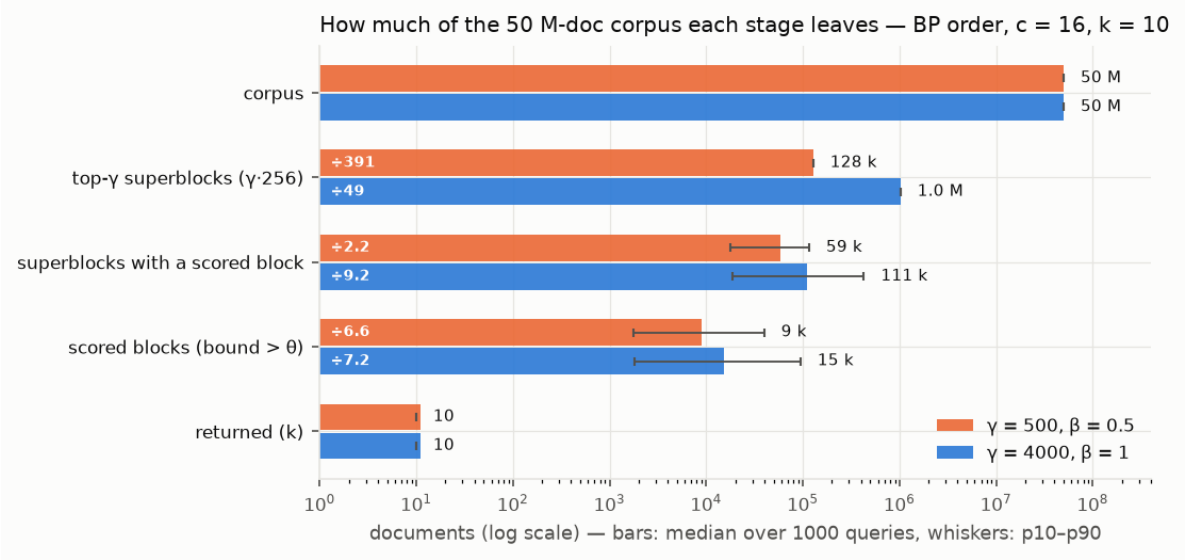

Figure 10. What each stage leaves of the 50 M-doc corpus (BP, c = 16, k = 10, the two Table 10 operating points re-run with per-query counters; medians with p10–p90 whiskers, in documents). Top-γ selection does the bulk of the cut (÷391 at γ = 500, ÷49 at γ = 4000); in the global block order only 230 of the 500 (433 of the 4000) candidate superblocks contribute a block whose bound beats θ, and inside those the block bound removes another 6.6–7.2×, so the median query scores 9 k docs (0.02 % of the corpus) at the fast point and 15 k (0.03 %) at the 99 % point, with a p90 of 40 k / 95 k. Appendix B walks two long queries through the same stages.

7. Reporting against the paper

How §6 compares against the paper, with explicit caveats:

- Paper: 8.8 M docs, similarity-clustered blocks (recursive graph bisection), i7-1260P, k=10 → 0.425 ms at 99.1 % of safe recall, 178× over MaxScore. Ours, same ordering algorithm (BP), Xeon 8375C, 1000 queries (Table 12): 5 M slice → 99.3 % recall@10 at 4.0 ms (44×) and 96.6 % at 2.7 ms (65×); 50 M docs → 99.1 % at 17 ms (123×), 98.3 % at 10.9 ms (193×) and 96.8 % at 7.8 ms (269×), all over our own exact full scan (§6; b = 32/c = 32 and b = 16/c = 64 indexes). At the paper's own γ = 500: 93.0–94.5 % at 1.8–2.5 ms (5 M), 88.7–89.6 % at 5.6–8.4 ms (50 M).
- Quality is matched; the latency gap (8–13× at 5 M) is located by the per-stage counters: the bound passes over 282-term queries (5–10× longer than MS MARCO's) and, at 50 M, the block-bound re-sweep; scoring is a minor share. The paper's SIMD-packed bound computation is the part not reproduced.
- Limitations: no relevance labels (agreement with the exact top-k, a stricter metric than the paper's recall of judged documents), 8-bit quantisation changes scores (≈0.5 % recall at γ = all), SIMD only in the two hot kernels (§6, item 5), single-machine numbers (single-threaded latency; throughput scales 12× to 16 threads, Table 12), and every row in §6 is measured on the same fixed 1000-query sample (seed 42), so rows are directly comparable; the exact-scan baselines use separate 100–300-query samples.

---

8. Reproduction

All numbers in §6 and Appendix A were measured on the AWS instance from §1.2 with the v2 SPLADE corpus in `data/v2/` (`part-NNNN.jsonl` files plus `queries.jsonl`, the default query file). The 5 M-document slice is a corpus directory holding every 10th document; substitute its path for `data/v2` below.

```
cargo build --release

# Exact baseline: build the index, then bench it (top-100, 100 queries)
./target/release/scibench index data/v2 --index-path splade-index
./target/release/scibench bench --index-path splade-index --num-queries 100
```

```
# LSP index in corpus order (b=16 docs/block, c=16 blocks/superblock)
./target/release/scibench index-lsp data/v2 --index-path lsp-index --b 16 --c 16

# LSP bench (recall is computed against the exact index for the same queries)
./target/release/scibench bench --index-path splade-index --num-queries 100 \
    --mode lsp --lsp-index-path lsp-index --gamma 250 --beta 0.5 --eta 1.0

# Reordered indexes (§3.4): dominant-term bucketing and k-means
./target/release/scibench index-lsp data/v2 --index-path lsp-index-dominant --reorder dominant
./target/release/scibench index-lsp data/v2 --index-path lsp-index-kmeans --reorder kmeans \
    --kmeans-k 4096 --kmeans-sample 1000000 --kmeans-iters 5

# Final configuration from §6: recursive graph bisection order, c=16 (~30 min on 50 M docs)
./target/release/scibench index-lsp data/v2 --index-path lsp-bp-c16 --b 16 --c 16 --reorder bp
# Same 1000-query sample for every row; exact top-k cached on first use per (corpus, k)
./target/release/scibench bench --index-path splade-index --num-queries 1000 --seed 42 \
    --exact-cache cache/exact-50m-k10.jsonl --limit 10 \
    --mode lsp --lsp-index-path lsp-bp-c16 --gamma 4000 --beta 1.0 --eta 1.0   # 98.4 %, 28 ms
./target/release/scibench bench --index-path splade-index --num-queries 1000 --seed 42 \
    --exact-cache cache/exact-50m-k10.jsonl --limit 10 \
    --mode lsp --lsp-index-path lsp-bp-c16 --gamma 500 --beta 0.5 --eta 1.0    # 87.7 %, 12 ms

# Query interface: one query through the LSP index. Queries are pre-computed SPLADE vectors,
# so --query names one from queries.jsonl (omit it for a random one); the command prints the
# query's terms and weights, the top-k with scores, and the work counters
./target/release/scibench search --mode lsp --lsp-index-path lsp-bp-c16 --gamma 2000 --beta 0.5 \
    --limit 10 --query "poodle coat color genetics inheritance fading"
./target/release/scibench search --limit 10 --query "poodle coat color genetics inheritance
        fading"   # exact baseline

# b = c = 32 (Table 11): reuse the BP ordering of the c = 16 index (100 s instead of 30 min)
./target/release/scibench index-lsp data/v2 --index-path lsp-bp-b32c32 --b 32 --c 32 \
    --reorder from-index --order-from lsp-bp-c16
./target/release/scibench bench --index-path splade-index --num-queries 1000 --seed 42 \
    --exact-cache cache/exact-50m-k10.jsonl --limit 10 \
    --mode lsp --lsp-index-path lsp-bp-b32c32 --gamma 8000 --beta 0.5 --eta 1.0
        # 99.1 %, 17 ms (26 ms with --sort-blocks --prefetch 0)

# Follow-up geometry (Table 12): b = 16, c = 64 from the same ordering
./target/release/scibench index-lsp data/v2 --index-path lsp-bp-b16c64 --b 16 --c 64 \
    --reorder from-index --order-from lsp-bp-c16
./target/release/scibench bench --index-path splade-index --num-queries 1000 --seed 42 \
    --exact-cache cache/exact-50m-k10.jsonl --limit 10 \
    --mode lsp --lsp-index-path lsp-bp-b16c64 --gamma 500 --beta 0.5              # 89.6 %, 5.6 ms
# Throughput: replay the sample on 16 threads after the timed single-thread pass
./target/release/scibench bench --index-path splade-index --num-queries 1000 --seed 42 \
```

```
    --exact-cache cache/exact-50m-k10.jsonl --limit 10 \
    --mode lsp --lsp-index-path lsp-bp-c16 --gamma 500 --beta 0.5 --threads 16    # 888 QPS
# The follow-up's rejected ideas are behind flags (--progressive, --gamma-short, --tau on an
# --impact-order index, --beta-blocks, --max-sb-frac); scripts/xp.sh LABEL 5m|50m INDEX FLAGS
# runs one logged configuration and scripts/remote.sh syncs/builds on the instance.
```

Appendix A. Corpus and exact-index statistics (AWS r6i.8xlarge)

Measured over the full 50 M-document SPLADE v2 corpus from `splade-index/terms.bin` + `meta.json`; per-document figures come from a 1 M-doc sample (`part-0000.jsonl`). The complete distributions (posting-list lengths, weights, query sizes) are in `stats.md`; this appendix explains the headline table (Table 15).

Table 15. Corpus and exact-index statistics (50 M docs).

Metric	Value
Documents	50,000,000
Postings	2,579,154,624
Vocabulary slots	49,152
Terms with ≥1 posting	48,243 (98.2%)
Mean postings / doc	51.6 (incl. padding)
Exact index size	postings.bin 19.22 GiB + docids.bin 2.86 GiB

Documents. Each document is a sparse SPLADE vector stored as a list of `(term_id, weight)` pairs, not text.

Postings. One posting is a single `(term, weight)` entry for one document; the count is the number of stored entries in the 50 M × 49 K sparse matrix and is the quantity that fixes both the index size and the work an exact scan does.

Vocabulary slots. `vocab_size` is `max term id + 1` (lsp_build.rs:98), so ids range over 0…49,151. 49,152 = 48 × 1024 is the model's output dimension padded to a round size, not the number of terms in use; it fits in a `u16`, which is what the LSP block layout (§3.1) relies on.

Terms with ≥1 posting. 48,243 of the 49,152 slots have a non-empty posting list; the remaining 909 are never used by any document.

Mean postings / doc. 2,579,154,624 / 50,000,000 = 51.6. The qualifier matters: term id 0 never occurs in any query but holds 420,209,545 postings (16.3% of the index), every one a `[0, 0.0]` pair. Documents were padded with repeated zero-weight entries (in the sample, 8–11% of docs are

padded, ≈60 copies each, max 418). Removing them leaves 2,158,945,079 real postings, i.e. 43.2 per document — the figure used in §1.1 and throughout the analysis.

Exact index size. `postings.bin` stores each posting as `u32` doc index + `f32` weight = 8 B, and 2,579,154,624 × 8 B = 19.22 GiB exactly; the padding postings alone account for 3.13 GiB of it. `docids.bin` holds the string doc ids (`u32` count, `(n+1) × u32` offsets, concatenated strings — the layout in lsp.rs:12), about 61 B per document.

Consequences for LSP.

- The 420 M zero-weight postings contribute nothing to any score, so the LSP builder drops them at parse time (lsp_build.rs); the exact baseline keeps them so its numbers stay comparable with the original index.
- With 2.16 B real postings and queries averaging 282 terms, an exact scan touches ≈39 M postings per query (p90 99 M, max 220 M) — 78% of the corpus in postings — which is the root of the seconds-long baseline latency in §1.3.

Appendix B. Two long queries, stage by stage

The two slowest queries of the fixed 1000-query sample at the 99 % operating point (50 M docs, BP order, c = 16, k = 10, SIMD build; solo runs of the two Table 10 configurations with `--per-query-csv`, which now also records the stage counters: superblocks touched, superblock entries read, and the four stage timers). Each table gives the query at both operating points next to the median over all 1000 queries of the same run, so the reader can see which stage grows and by how much. Both runs use the default global block order, so "candidate superblocks with a scored block" counts candidates that contributed at least one block whose bound beat θ; it is below γ even when the superblock bound never drops under θ.

Table 16. Query A — "bracketology seeding criteria RPI strength schedule 2026" (1715 non-zero terms; rank 1 by latency at γ = 4000, β = 1).

stage	γ = 500, β = 0.5	median query	γ = 4000, β = 1	median query
query terms (non-zero)	1,715	148	1,715	148
terms used for bounds ⌈β·n⌉	858	74	1,715	148
1 · superblock bound pass				
superblock entries read	26,049,465	2,969,100	50,258,619	5,493,806
superblocks touched (bound > 0)	195,235	193,703	195,299	194,582
share of all 195 k superblocks	100.0 %	99.2 %	100.0 %	99.6 %

stage	γ = 500, β = 0.5	median query	γ = 4000, β = 1	median query
time (ms)	32.5	3.96	62.9	7.12
2 · top-γ selection				
candidates kept (γ)	500	500	4,000	4,000
time (ms)	0.50	0.48	0.57	0.60
3 · block bounds of the candidates				
candidate blocks (γ·c)	8,000	8,000	64,000	64,000
time (ms)	24.4	3.01	95.1	11.7
4 · visit and score				
candidate superblocks with a scored block	497	230	2,967	433
blocks scored (block bound > θ)	3,975	561	13,786	958
docs scored	63,600	8,976	220,576	15,336
share of the corpus scored	0.13 %	0.02 %	0.44 %	0.03 %
time (ms)	5.56	1.02	22.6	3.65
result				
θ (10th score, quantised units)	36.1	14.0	39.4	14.2
recall@10 (positions 1–3 / 4–10)	0.60 (1.00 / 0.43)	0.90 (median)	0.90 (1.00 / 0.86)	1.00 (median)
total latency (ms)	63.2	8.83	181.4	23.7

Table 17. Query B — "nested loops break multiple levels C alternatives" (1895 non-zero terms; rank 2 by latency at γ = 4000, β = 1).

stage	γ = 500, β = 0.5	median query	γ = 4000, β = 1	median query
query terms (non-zero)	1,895	148	1,895	148
terms used for bounds ⌈β·n⌉	948	74	1,895	148
1 · superblock bound pass				

stage	γ = 500, β = 0.5	median query	γ = 4000, β = 1	median query
superblock entries read	24,601,569	2,969,100	46,311,926	5,493,806
superblocks touched (bound > 0)	194,993	193,703	195,253	194,582
share of all 195 k superblocks	99.8 %	99.2 %	100.0 %	99.6 %
time (ms)	30.9	3.96	58.1	7.12
2 · top-γ selection				
candidates kept (γ)	500	500	4,000	4,000
time (ms)	0.57	0.48	0.64	0.60
3 · block bounds of the candidates				
candidate blocks (γ·c)	8,000	8,000	64,000	64,000
time (ms)	28.3	3.01	97.7	11.7
4 · visit and score				
candidate superblocks with a scored block	480	230	2,528	433
blocks scored (block bound > θ)	3,231	561	10,271	958
docs scored	51,696	8,976	164,336	15,336
share of the corpus scored	0.10 %	0.02 %	0.33 %	0.03 %
time (ms)	4.50	1.02	17.0	3.65
result				
θ (10th score, quantised units)	37.1	14.0	39.8	14.2
recall@10 (positions 1–3 / 4–10)	0.70 (0.67 / 0.71)	0.90 (median)	1.00 (1.00 / 1.00)	1.00 (median)
total latency (ms)	64.6	8.83	173.8	23.7

What the two queries show.

- Both are in the top 1 % by length (1715 and 1895 non-zero terms, 12× the median 148), and every stage scales with that: the bound pass reads 26–50 M superblock entries (9× the median) and takes 31–63 ms (8–9×); the candidate block-bound sweep, which re-reads the same terms' lists for the γ candidates, is the largest stage at γ = 4000 (95–98 ms, 8× the

median); scoring is 5–23 ms. The stage shares are the same as for the median query — nothing new appears, everything is 8–9× bigger — which is the latency ∝ nnz^0.75 law of Figure 3 seen from inside.

- Long queries touch every superblock (100 %), but so does the median query (99 %): with ~150 terms almost every 256-doc superblock contains at least one of them, so the bound pass never prunes by itself and top-γ selection is the first real cut. The bounds of long queries are also looser relative to θ (more terms whose maxima can be summed), so they keep more blocks: 0.10–0.44 % of the corpus is scored against 0.02–0.03 % for the median query, and 2528–2967 of the 4000 candidates contribute a scored block at γ = 4000 against 433.
- Quality follows the same pattern as the slices: at the fast point (β = 0.5 keeps the heaviest 858–948 terms for bounds) Query A loses four of its top-10 — all from positions 4–10, the head is intact — and Query B three. The bound-ranks from the Figure 7 diagnostic say why: under β = 1 Query A's answers sit at superblock ranks 1, 5, 10, 10, 85, 175, 1177, 1395, 1762 and 2334, Query B's at 49–1443, so at γ = 500 four and three of them are out of reach whatever the bounds do. With β = 1 and γ = 4000 every answer is inside the candidate set: Query B is recovered fully, and Query A's remaining miss (0.90) is not a pruning loss but the 8-bit quantisation moving a near-tie at the top-10 boundary (§7, ≈0.5 % at γ = all). For queries like these the exact scan is not a fair alternative either: it reads ∝ terms × posting lists, and the exact p90 on this sample is 12.3 s.

Appendix C. Superblocks vs superblock entries

Two counts that are easy to confuse, both used throughout §6 and Appendix B.

A superblock is a unit of the *document partition*: `b · c` consecutive documents in the index's document order (256 with b = c = 16). Its count depends only on the corpus size and the geometry, `⌈docs / (b·c)⌉`, and a query's "superblocks touched / visited" counts these units.

Table 18. Number of superblocks (and blocks) by geometry.

corpus	blocks (16 docs)	superblocks, c = 16	c = 4	c = 1	b = c = 32 (32-doc blocks)
50 M docs	3,125,000	195,313	781,250	3,125,000	48,829 (1,562,500 blocks)
5 M slice	312,500	19,532	78,125	312,500	4,883 (156,250 blocks)

A superblock entry is a unit of the *per-term inverted structure* (§3.1): for every term the index stores a sorted list of `(superblock id, max quantised weight of the term in that superblock)`,

one entry per superblock in which the term occurs at least once. `total_sb_entries` is the sum of those list lengths over the vocabulary; a query's "superblock entries read" (Appendix B) is the sum over its β-pruned terms of their list lengths, and the bound pass is linear in it. Each superblock entry in turn owns a run of block entries, `(block index within the superblock, block max)`, one per block of that superblock where the term occurs; the candidate block-bound sweep reads those.

Table 18. Entry counts of the four orderings at each scale (b = c = 16) and of the b = c = 32 index of Table 11. "Per superblock" / "per block" is the number of distinct terms a superblock / block contains on average; the last column is the size of the pruning structure alone (13 B per superblock entry: u32 id + u8 max + u64 block offset; 2 B per block entry), which is where the index-size differences of §6 come from — the forward index is the same in every ordering.

ordering	superblock entries	per superblock	block entries	per block	sb + blk lists
50 M docs					
corpus order	1,600,748,804	8,196	2,111,552,818	676	23.3 GB
dominant term	1,134,128,478	5,807	1,717,690,920	550	16.9 GB
k-means	996,493,771	5,102	1,578,092,564	505	15.0 GB
BP	705,185,889	3,611	1,301,469,408	416	11.0 GB
BP, b = c = 32	424,505,320	8,694	1,147,119,361	734	7.3 GB
5 M slice					
corpus order	160,093,232	8,196	211,167,557	676	2.3 GB
dominant term	124,516,357	6,375	180,945,988	579	1.8 GB
k-means	106,782,602	5,467	167,969,708	538	1.6 GB
BP	81,240,132	4,159	143,456,186	459	1.3 GB
BP, b = c = 32	49,169,729	10,070	128,287,780	821	0.8 GB

The number of superblocks is fixed by the geometry; what the ordering changes is how many *distinct terms* each superblock contains. In corpus order a 256-doc superblock holds ~8,200 distinct terms (from 43 postings per doc with little overlap); BP packs documents that share terms, so a superblock holds ~3,600 and every per-term list is 2.3× shorter at 50 M (1.60 B → 0.71 B entries). That is the second effect of reordering, on top of tighter bounds: at equal γ the bound pass reads fewer entries, which is why latency falls along with recall rising in Figure 6, and why the BP index is the smallest (20.9 GB vs 33.5 GB, §6). For scale, Appendix B's Query A reads 26 M entries for its 858 bound terms — ~30 k per term, i.e. each of its terms occurs in about 15 % of all

superblocks — where the median query reads 3.0 M for 74 terms. Going to b = c = 32 (Table 11) cuts the superblock entries a further 1.7× (0.71 B → 0.42 B: a 1024-doc superblock holds 8,700 distinct terms, but there are 4× fewer of them) — that is the halved bound pass — while the block entries fall only 12 %, since a 32-doc block holds nearly twice the terms of a 16-doc one.

Appendix D. The pipeline as one flowchart

Figure 11 draws the system end to end: the two inputs, the index build with its reordering step (§3.2, §3.4), the exact baseline that supplies the ground truth, the four-stage query path (§3.3) and the evaluation harness (§4) that turns runs into the tables and figures of §6.

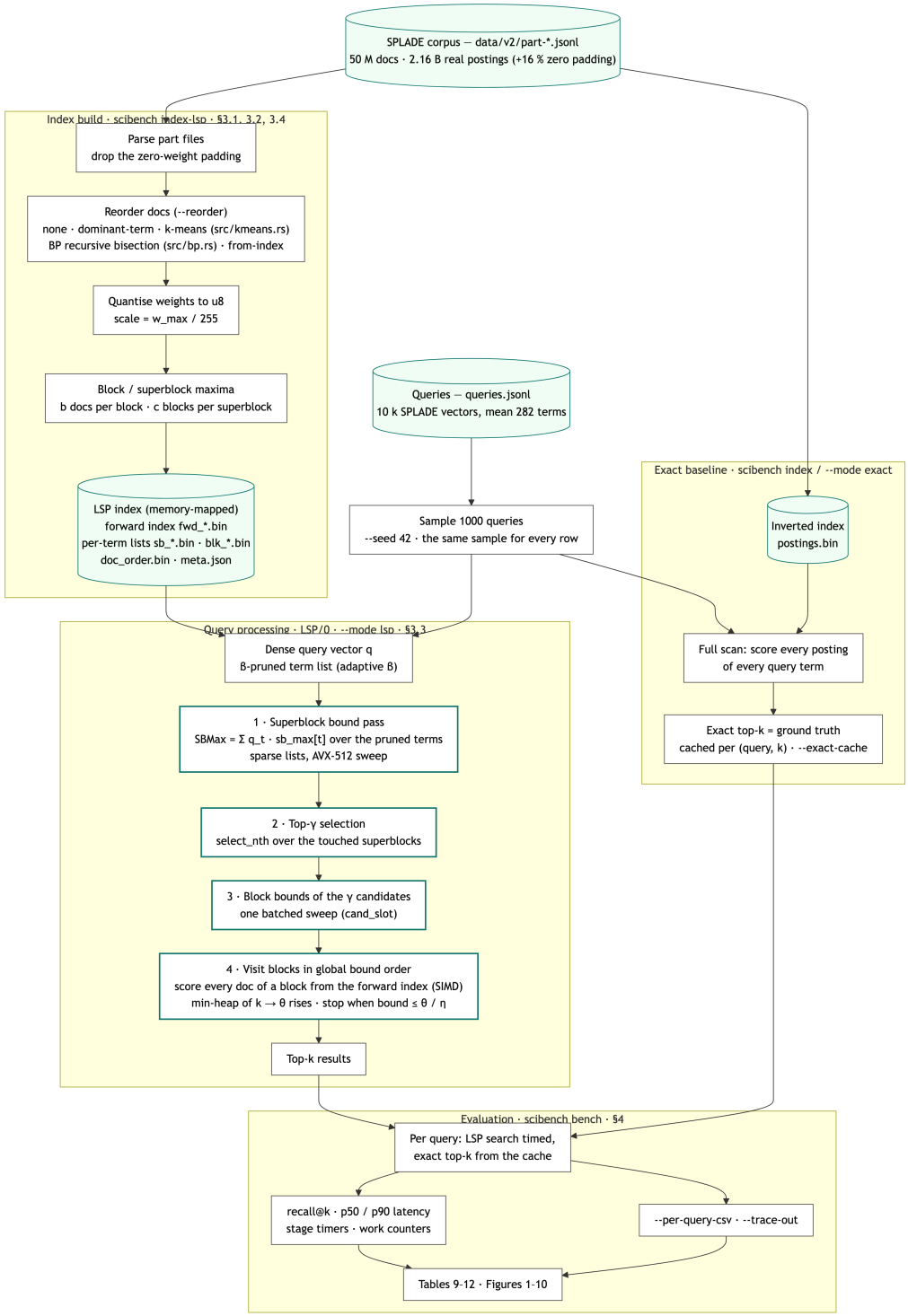

Figure 11. The pipeline end to end. Cylinders are on-disk artefacts; the four numbered boxes are the LSP/0 query stages whose timers and counters appear throughout §6. The evaluation loop runs the timed LSP search and reads the cached exact top-k for the same query, so every row of Tables 9–12 is measured on the same 1000 queries.