

MongoDB for PageIndex Retrieval: Preliminary Benchmark, Diagnosis, and Deployment Optimization Plan

Junyao Dong

Abstract. PageIndex stores documents as large hierarchical JSON trees that an agent navigates from the root to relevant nodes. This report first presents a preliminary benchmark of the exact storage calls issued by ConDB’s retrievers on MongoDB, PostgreSQL/JSONB, DuckDB, and SQLite, at scales up to ten million nodes. MongoDB is competitive on navigation reads, content fetch, writes, concurrency, and storage footprint; the only material weakness is `get_subtree`, a materialized-path range scan over tens of thousands of nodes. The initial multi-engine run recorded a 2.5s MongoDB P95 for that operation, but a controlled re-measurement with the collection resident in cache gives a stable P95 of roughly 0.4s, about $3\times$ PostgreSQL rather than the $\sim 20\times$ implied by the first tail. The diagnosis is MongoDB’s document-at-a-time read model: a non-covering subtree scan performs one `FETCH` per matched node. A resident-cache A/B test shows the consequence: splitting `text` out of the node document does not change latency when both schemas already fit in cache, while a lean covering index gives a modest $\sim 1.4\times$ improvement by removing `FETCH`. The deployment plan is therefore ordered by production constraints, not by the resident-cache A/B: split `text` first to protect cache residency, keep sharding and compound indexes led by `tree_id`, add lean covering indexes where useful, and denormalize hot subtree views only if `get_subtree` remains a bottleneck.

1 Introduction

ConDB indexes a document as a PageIndex tree: a hierarchy of nodes carrying a title, a summary, page indices, and (at the leaves) text. Retrieval is a top-down traversal: the retriever starts at the root, lists a node’s children to decide where to descend, pulls a subtree for the language model to read, and fetches the content of the nodes it selects. This report is organized as an experiment-to-plan document. Section 3 presents the preliminary benchmark, the MongoDB bottleneck diagnosis, and the resident-cache optimization A/B. Section 4 converts those measurements into a deployment optimization plan.

We deliberately benchmark only the operations ConDB actually issues. Generic database operations that the retrieval path never performs (ancestor-to-root walks, page-range filters, naive substring scans) are excluded, because their results would not inform the optimization.

2 Workload and methodology

Operations. The retrievers (`beam_retriever.py`, `block_retriever.py`) reach storage through four calls (Table 1). Everything measured below is one of these, plus the storage footprint, the write path (incremental index and reasoning-trace edits), and concurrency.

Table 1: The storage operations ConDB retrieval issues.

Operation	Storage call	Used for
Point lookup	<code>get_node(tree_id, node_id)</code>	Navigate to / read one node
Expand children	<code>get_children(tree_id, node_id)</code>	List children, pick where to descend
Get subtree	<code>get_subtree(tree_id, node_id, depth)</code>	Render the tree view, pull a block
Fetch content	<code>get_entity(tree_id, node_id)</code>	Read selected nodes' content

Data. Synthetic PageIndex trees in the canonical format, validated against ConDB's `DocumentTreeAdapter`, at two scales (Table 2). Content is random words; what the benchmark exercises is the tree shape and the per-node payload sizes.

Table 2: Datasets.

Dataset	Nodes	JSON size	Depth	Fan-out
Medium	70,843	85 MB	5	6–12
Large	10,000,000	14.06 GB	8	6–14

Engines and setup. MongoDB 7.0 (Community Edition) and PostgreSQL 16 (JSONB document column) run as servers in Docker over localhost; DuckDB and SQLite are embedded. All engines run on the same 96-core Linux host with 1 TB of RAM. Every engine stores one record per node with identical logical fields and the indexes each query needs. The tree is flattened to one row per node; `get_subtree` uses a materialized path with a byte-order range predicate, identical across engines (PostgreSQL's `path` column uses `COLLATE "C"` so its range matches the others' binary comparison). The subtree query returns node ids; adding the title and summary the tree view also renders would cost the relational engines more but not MongoDB, whose non-covering `FETCH` already reads the whole document, so adding fields to the projection costs it nothing—making this choice conservative toward MongoDB. Each read is measured over 500 sampled nodes (200 subtree roots, drawn at depth ≤ 3), fixed by seed and identical across engines; every query was checked to return the same row count on all four engines. Each concurrency level runs its client processes for five seconds. The host has ample memory, so every engine's data is cache-resident and reads are not disk-bound.

Fairness caveats. SQLite and DuckDB run in-process, so a query is a function call; MongoDB and PostgreSQL pay per-call client-server overhead (a localhost round trip plus protocol handling) that the embedded engines never incur. The fair single-call comparison is therefore MongoDB versus PostgreSQL; absolute microseconds across the embedded/server line are not comparable, and the concurrency section (independent processes, no GIL) shows real server throughput. MongoDB's `storageSize` is post-compression (WiredTiger snappy); the others are uncompressed on-disk files, so MongoDB's uncompressed logical size is reported separately, and the random-word content likely compresses better than real prose, flattering the compressed figure. All measurements are single-host. The operations through concurrency use one tree per database; multi-tenancy, batched reads, and deletion are measured separately below on 25 co-resident trees.

3 Preliminary experiments and diagnosis

3.1 Baseline benchmark

3.1.1 Storage and ingest

Table 3: Storage and ingest (large dataset). On-disk totals include the indexes. MongoDB holds 15.21 GB of logical BSON in 5.27 GB of data on disk (WiredTiger snappy).

Engine	On-disk	Index	Uncompr.	Ingest (n/s)	Build
DuckDB	5.02 GB	0	–	348,831	10 s
MongoDB	5.81 GB	536 MB	15.21 GB	41,387	69 s
PostgreSQL (JSONB)	17.71 GB	1.19 GB	–	62,816	52 s
SQLite	19.51 GB	980 MB	–	52,880	38 s

DuckDB and MongoDB are the most compact (Table 3), DuckDB through columnar encoding and MongoDB through compression; even uncompressed, MongoDB’s 15.21 GB of BSON is no larger than the relational engines’ raw tables. PostgreSQL and SQLite are several times larger on disk.

3.1.2 Retrieval operations

The navigation reads (point lookup, children, content fetch) are sub-millisecond on every engine but DuckDB, which is not built for OLTP point reads. MongoDB runs roughly 3× behind PostgreSQL per call on these reads (Table 4), part of it per-call client overhead, but at well under a millisecond everywhere the gap carries no practical cost; the optimization space is not here. Only `get_subtree` costs at scale: a large subtree returns tens of thousands of documents. The initial run (Table 4) put MongoDB’s P95 in the seconds while the others stayed under ~150ms, but that figure did not hold up under scrutiny—a controlled re-measurement (Section 3.2.2) finds a stable ~0.4s. The materialized-path range also beats a recursive `parent_id` traversal, so ConDB should keep it.

Table 4: Retrieval read latency (ms, lower is better). Best per row in bold. The large `get_subtree` P95 (2478) is from the original multi-engine run; a controlled re-measurement finds ~0.4s (Section 3.2.2).

Operation	MongoDB	PostgreSQL	DuckDB	SQLite
<i>Medium, P50</i>				
Point lookup (<code>get_node</code>)	0.238	0.081	2.705	0.008
Expand children (<code>get_children</code>)	0.251	0.090	0.856	0.017
Get subtree (<code>get_subtree</code>)	0.498	0.210	2.700	0.091
Fetch content (<code>get_entity</code>)	0.228	0.071	2.750	0.007
<i>Large, P50 / P95</i>				
Point lookup	0.270 / 0.32	0.087 / 0.09	5.59 / 8.97	0.012 / 0.014
Expand children	0.282 / 0.32	0.099 / 0.14	4.28 / 5.05	0.023 / 0.032
Get subtree (~36k rows)	30.9 / 2478	10.7 / 123	9.9 / 47	11.8 / 135
Fetch content	0.249 / 0.32	0.081 / 0.09	5.21 / 8.35	0.012 / 0.013

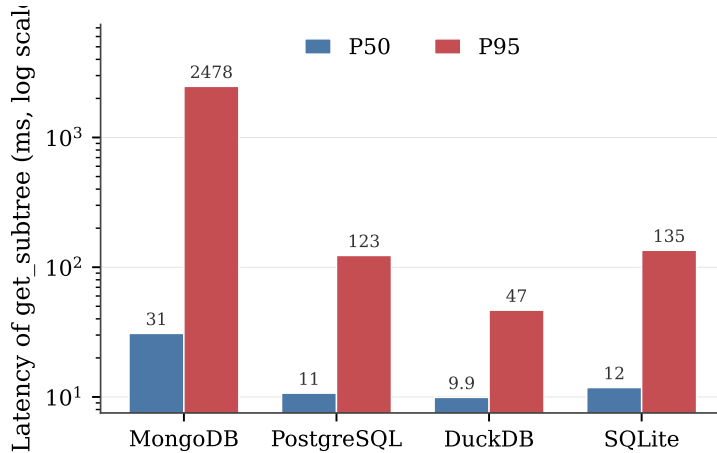


Figure 1: `get_subtree` latency on the large dataset (log scale). The medians are within a small factor. MongoDB’s P95 outlier at 2.5s is from this multi-engine run; in isolation and under concurrent load it is ~ 0.4 s (Section 3.2.2).

3.1.3 Write path

On the write path (Table 5), PostgreSQL posts the lower update latency of the two servers, but each `jsonb_set` versions the row and leaves a dead tuple for a later `VACUUM`. WiredTiger is MVCC too, yet it reclaims obsolete document versions automatically, so an update-heavy workload carries no growing bloat; its measured on-disk delta is small but noisy, since WiredTiger reports storage in checkpoint-sized chunks. DuckDB is unsuitable for OLTP writes.

Table 5: Write path (medium): 5,000 field updates, 2,000 incremental inserts. “Bloat” is on-disk growth from the updates.

Engine	Upd. P50	Upd. ops/s	Insert ops/s	Bloat
MongoDB	0.242	4,050	5,586	0 MB
PostgreSQL (JSONB)	0.123	7,856	8,732	7 MB
DuckDB	2.966	332	450	7 MB
SQLite	0.017	40,442	21,815	0 MB

3.1.4 Concurrency

Single-call latency understates the server engines. Driving point lookups from a growing pool of independent OS processes (no GIL bottleneck), both server engines gain throughput with added clients, and the gap to the embedded engines narrows far below what the single-call numbers suggest (Figure 2). MongoDB rises steadily across the whole range; PostgreSQL peaks near 16 clients; SQLite is fastest in absolute terms but peaks earliest, within the first few clients, then degrades as the processes contend on its single file. Repeated runs move the peak positions and heights somewhat; the shapes hold.

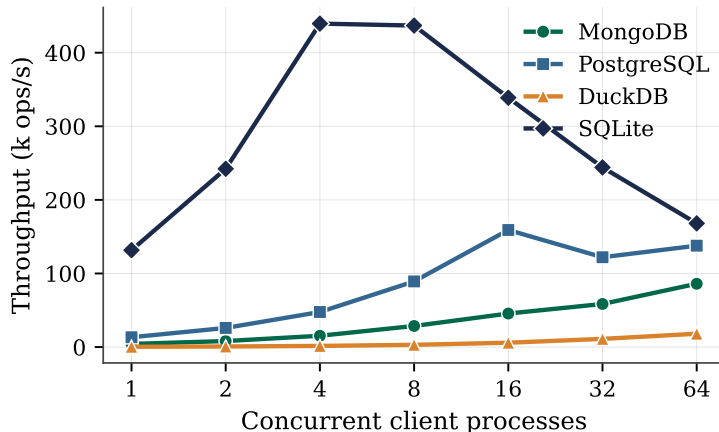


Figure 2: Point-lookup throughput versus concurrent client processes (medium).

3.1.5 Multi-tenancy, batched reads, and deletion

Mapping the retriever’s calls to storage surfaces three operations the sections above leave out: a per-query tenant filter when many trees share one store, the batched id-list read the navigation step issues, and whole-tree deletion. We measure them on the medium tree replicated into 25 co-resident trees (1,771,075 rows in one store), with every read filtered by `tree_id` and served by a compound index led by `tree_id`.

Multi-tenant selectivity. With 25 trees in one collection, a `tree_id`-led index holds each read close to its single-tree latency (Table 6): the cost tracks index depth, not the number of co-resident trees. MongoDB rises about $1.5\times$ on point lookup and $2.6\times$ on subtree over the single-tree medium figures (Table 4) for a $25\times$ larger store, with no scan blow-up, and the MongoDB–PostgreSQL ordering is unchanged.

Table 6: Per-query reads with 25 trees co-resident in one store, each filtered by `tree_id` (P50, ms).

Operation	MongoDB	PostgreSQL	DuckDB	SQLite
Point lookup	0.345	0.124	3.772	0.008
Expand children	0.372	0.147	3.073	0.014
Get subtree	1.273	0.312	4.605	0.069

Batched reads. The navigation step returns a list of candidate node ids, so the retriever fetches several nodes at once. One batched read (`$in` on MongoDB, `IN` on the relational engines) beats the point-lookup loop it replaces on every engine, and the margin grows with batch size because each iteration of the loop on a server engine is a round trip. At 200 ids MongoDB is 4.6 ms batched against 273 ms looped, a $60\times$ gap; PostgreSQL is 1.4 ms against 29 ms (Table 7). MongoDB gains the most because batching amortizes the per-call client overhead that costs it on single reads; in-process SQLite, with no round trip, gains about $2\times$. The retriever should resolve a candidate set with one batched read, not a loop.

Table 7: Batched id-list read at 10, 50, and 200 ids against the 200-id point-lookup loop it replaces (P50, ms per batch).

Engine	Batched (n ids)			Loop
	10	50	200	200
MongoDB	0.46	1.56	4.56	273
PostgreSQL (JSONB)	0.27	0.78	1.40	29
DuckDB	19.6	22.2	39.1	790
SQLite	0.03	0.13	0.50	0.97

Deletion. Removing a tree (one tenant, 70,843 of 1,771,075 rows) is a lifecycle operation, not part of retrieval. The delete is fast on every engine, under four seconds, but none returns disk space on the delete alone (Table 8). SQLite (`VACUUM`) and PostgreSQL (`VACUUM FULL`) reclaim the removed tree’s footprint through a full, locking rewrite that takes 10 to 17 s; MongoDB `compact` and DuckDB do not measurably shrink the file for a single-tenant deletion, which is WiredTiger’s documented behavior of not returning space to the operating system on delete. For MongoDB this mirrors the write path: WiredTiger carries no vacuum debt under updates, and in exchange a one-off bulk delete does not auto-shrink. WiredTiger reports `storageSize` in checkpoint-sized chunks, so changes below tens of megabytes are reporting granularity, not reclaimed space.

Table 8: Whole-tree delete and space reclamation (delete one of 25 trees).

Engine	Delete (rows/s)	Reclaim	Reclaim time	On-disk (before → after)
PostgreSQL (JSONB)	157,578	<code>VACUUM FULL</code>	17.3 s	2.77 → 2.66 GB
SQLite	141,842	<code>VACUUM</code>	10.6 s	2.56 → 2.46 GB
MongoDB	60,680	<code>compact</code>	0.1 s	911 MB, no shrink
DuckDB	20,171	<code>checkpoint</code>	–	790 MB, no shrink

None of the three changes the benchmark conclusion: a `tree_id`-led index keeps tenant selectivity stable, batched reads remove the point-lookup loop cost, and deletion behavior affects lifecycle maintenance rather than retrieval.

3.2 MongoDB `get_subtree` diagnosis

3.2.1 Root cause

The cause is structural, not a configuration mistake. MongoDB’s unit of read is the whole document. A secondary index (here on `path`) is a separate B-tree holding only the indexed field and a record id; to return any other field, the engine must *fetch* the full BSON document and apply the projection to it. A point lookup is one such fetch; a large subtree is tens of thousands, the biggest in the workload far more. The data is cache-resident (and pages in the WiredTiger cache are stored uncompressed), so the cost is neither disk I/O nor decompression: the time goes to per-document work. Because the `path` index is non-covering, each matched record id triggers a `FETCH` of the whole BSON document—`text` payload and all, even though the query needs only a node id—and tens of thousands of those fetches batch through the cursor. Multiplied by subtree size, that per-document overhead is the cost of the operation. The other operations touch few documents, so they stay fast.

The cost therefore scales with the number of nodes in the subtree: the largest subtrees, near the root, are the slowest. How large that tail actually is—and whether the original run’s multi-second figure is a steady-state property or an artifact of that run’s contention—is the subject of the next section.

The row and columnar engines do less work per node. A row store visits each matching row but materializes only the requested column; the wide `text` sits unread in the row (PostgreSQL does not even decompress the JSONB datum, since `node_id` is a separate column). A column store reads only the requested column in vectorized batches. Both could go further still with a covering index that answers the range from the index alone (Section 4). All of these do far less per-row work than a per-document fetch.

This is the trade-off of the document model. It is unfavorable for a projection-heavy range scan over many nodes, but favorable for operations that read or update a whole node: point reads, children reads, content fetches, field updates, tenant-local sharding, and compressed storage. The PageIndex workload matches that model on every measured operation except the large-subtree scan.

3.2.2 Controlled tail re-measurement

The original `get_subtree` figure (P95 2478 ms) was recorded while the other three engines, the write benchmark, and the concurrency sweep ran on the same host. To separate the query’s own cost from that contention, we re-measured it on the ten-million-node tree under controlled conditions: one measuring client, the collection freshly loaded and fully resident, the same 200 seeded subtree paths.

Re-measured in isolation, P95 is $\sim 0.30\text{--}0.42$ s (P50 $\sim 30\text{--}40$ ms), stable across repeated runs—an order of magnitude below the first run. Driving the same query while 16, 32, then 64 concurrent processes hammer point lookups at the same `mongod` barely moves it (Table 9): P95 rises from 331 to 376 ms across the sweep, $1.1\times$, never approaching seconds. Throughout, the WiredTiger cache holds the whole collection (~ 17 GB resident in a 540 GB cache) and reports zero application-thread evictions—so the tail is neither eviction nor, as it turns out, ordinary read contention.

Table 9: `get_subtree` re-measured on the 10M tree, one measuring client, while N background processes drive concurrent point lookups (id-only projection, ms).

Concurrent load N	P50	P95	P99	WT evictions
0	31.4	331	476	0
16	39.4	357	555	0
32	44.1	415	534	0
64	39.0	376	627	0

The corrected picture is milder than the first run implied. Steady-state, `get_subtree` on the large tree is ~ 0.4 s at P95 and $\sim 30\text{--}40$ ms at the median—roughly $3\times$ PostgreSQL at *both* (its 123 ms tail and 11 ms median), not the $\sim 20\times$ the original P95 alone suggested. The 2478 ms was a property of the loaded machine during the first run—most likely whole-system resource contention, or a cold-cache first touch before the collection was warm—not of the query in steady state. The mechanism from the previous section still holds (each matched node is a separate `FETCH`, so cost grows with subtree size), but its steady-state magnitude is sub-second.

3.3 Resident-cache optimization A/B

The diagnosis above separates two regimes. The first is the resident-cache experiment: which schema/index variants change `get_subtree` when the whole collection already fits in cache? The second is production deployment: which schema choices are required when the working set no longer fits RAM? This section answers only the resident-cache question. Each variant below stays on MongoDB Community Edition and was measured on the ten-million-node tree (Table 10).

Table 10: `get_subtree` variants measured on the 10M tree (one client; view returns `node_id+title+summary`, `id` returns `node_id` only; ms).

Variant	P50	P95	Result
Baseline, view (<code>text</code> inline)	39.6	419	reference
Subset Pattern (split <code>text</code>), view	39.2	419	no change
Lean covering <code>{path,node_id}</code> , id	19.3	191	~1.4×, covered
Fat covering <code>+title,summary</code> , view	33.5	350	~1.2×; 4.66 GB index
Lean covering + batched <code>\$in</code> , view	137	1314	~3× slower

Table 10 is therefore a *resident-cache* optimization ranking, not a full deployment ranking. In a large deployment with many trees or tenants, the constraint can reverse: the working set may exceed RAM, and then the dominant constraint is which bytes remain cache-resident. Under that condition, splitting `text` out of the hot node collection is the first schema requirement, followed by `tree_id`-led indexes/sharding so each query stays tenant-local, and then by lean covering indexes or denormalized subtree views. The experiment below does not quantify that cache-constrained benefit; it only shows that the benefit cannot appear when both schemas already fit in cache.

3.3.1 Measured resident-cache variants

Subset Pattern: no latency change while resident. Moving the large `text` field into a separate collection shrinks the scanned documents, but in this resident-cache experiment it does not reduce latency. It shrinks the collection by ~75% (4.97 → 1.25 GB) yet leaves `get_subtree` P95 unchanged (419 → 419 ms). The reason is the cache: at 540 GB it holds the whole collection many times over (Section 3.2.2), so document *size* never gates the scan—document *count* does. Each of the ~36k matched nodes is a separate `FETCH`, and shrinking each one does not reduce their number. The Subset Pattern’s premise—a working set too large for RAM—does not hold on this host. For production scale, this is exactly the boundary condition to watch: if inline `text` pushes the `structure/title/summary` working set out of cache, the split becomes a cache-residency requirement even though it did not move this resident-cache latency number.

Covering index: modest improvement by removing `FETCH`. Covered execution is the only measured resident-cache variant that reduces latency. A lean `{path,node_id}` index serves the id-only subtree query index-only (`explain` confirms `PROJECTION_COVERED`, `totalDocsExamined: 0`) for ~1.4×. But the tree view also needs `title` and `summary`: extending the index to cover them works (~1.2×) yet inflates it to 4.66 GB—as large as the collection’s own data—because `summary` is a multi-kilobyte key. Resolving the view with a covered id scan plus a batched `$in` is worse,

not better ($\sim 3\times$ slower): a $\sim 36\text{k-id}$ `$in` overflows the 16 MB command limit (forcing ~ 36 chunked round trips) and re-fetches the same documents anyway. Keep the anchored materialized-path range as the predicate—an unanchored regex cannot form a range and must scan the whole index—and treat cache sizing and `zstd` as irrelevant here: the data is already resident, so neither touches the per-fetch cost.

Structural denormalization: larger but unmeasured. Nested Sets or a tree-order `_id` on a clustered collection turns the subtree into one range query or a sequential read; a pre-computed subtree document turns it into a point read. All are viable precisely because the tree is write-once (their cost is paid once at ingest), and all remove fetches wholesale rather than shrinking them. They were not measured in this pass. With the baseline already at $\sim 0.4\text{s}$, the remaining decision is whether the added schema complexity is justified for a sub-second, infrequent operation. (`$graphLookup` is not a candidate—it re-traverses `parent_id`, still materializes documents, and in 7.0 is capped at 100 MB and ignores `allowDiskUse`.)

3.3.2 Out-of-scope levers for this bottleneck

MongoDB version upgrade. Upgrading the engine does not fix this without a schema change. Against this operation—a projection-heavy range scan over ordinary documents—the 8.0–8.3 line offers nothing: the 8.0 *Express* path is a point/equality fast path on a single index (`_id`-style, no range, sort, or skip), 8.0 *block processing* is scoped to time-series collections, and the `$graphLookup` disk-spill fix (SERVER-23980) lands only in 8.2.0 (reverted from 8.1) and only helps the recursive alternative anyway. The document-at-a-time `FETCH` is unchanged across the line.

Atlas/Enterprise-only features. The most direct engine feature for this access pattern is a column-store index, so a projection-heavy scan reads only the projected column, as the relational engines do—is Atlas-only. But `PageIndex` never scans the whole corpus, so a column store solves a problem this workload does not have; the schema changes above address the one it does. Self-managed Search and Vector Search run on Community since 8.2 (a separate `mongot` process, public preview, replica set required) but serve relevance retrieval, not a range-projection scan. No Atlas- or Enterprise-only feature addresses the document-at-a-time read model itself, so the only levers are schema, index, and storage configuration—all on Community.

4 Deployment optimization plan

The measured A/B in Section 3.3 ranks variants only for a cache-resident host. A production deployment has the opposite constraint: many trees or tenants can make the working set larger than RAM, so the first goal is to keep the hot navigation data in cache. The deployment plan is therefore:

1. **Split text from the hot node collection.** Store `tree_id`, `node_id`, `structure`, `title`, and `summary` in the navigation collection, and store large body text in a separate collection keyed by `(tree_id,node_id)`. The resident-cache experiment shows this is not a latency improvement when both layouts fit in RAM. At production scale it is still the first schema requirement, because `get_subtree` and navigation never need body text; keeping text inline only makes cold payload bytes compete with hot structure bytes for cache.

2. **Lead every index and shard key with `tree_id`.** Multi-tenant reads must remain tenant-local: point lookup, children lookup, subtree range scan, and content fetch should all be served by compound indexes beginning with `tree_id`. This preserves selectivity as the number of trees grows and is the natural shard key for horizontal scale.
3. **Add lean covering indexes on the navigation collection.** A `{tree_id,path,node_id}` index can serve id-only subtree scans without document fetch. Avoid putting multi-kilobyte `summary` into the default covering key unless the measured workload requires it: the resident-cache A/B shows the fat index works, but the 4.66 GB index buys only a small additional gain.
4. **Denormalize hot subtree views only after measuring demand.** If `get_subtree` remains a top latency contributor, use write-once structure to precompute bounded structure+summary subtree views, or add Nested Sets / tree-order `_id` for range-locality. These are the only options that remove the many-fetch shape entirely, but they add storage amplification and more complicated ingest.

The rollout test should reproduce the missing production condition: run the Subset Pattern A/B with the WiredTiger cache below the inline-text working set and above the split navigation working set. That experiment prices the cache-residency benefit directly. The current preliminary benchmark cannot quantify it because both layouts were already cache-resident.

5 Conclusion

The preliminary benchmark supports MongoDB for the PageIndex retrieval workload with one qualification. Point reads, children reads, content fetches, writes, multi-tenant selectivity, batched reads, concurrency, and footprint are all acceptable or favorable. The one weak operation is `get_subtree`, whose steady-state P95 is about 0.4 s on the large tree after re-measurement, roughly $3\times$ PostgreSQL rather than the 2.5 s first observed under whole-system contention.

The resident-cache optimization A/B shows that the measured bottleneck is the number of non-covering document fetches. Splitting `text` does not improve latency when both layouts fit in cache; a lean covering index gives the only cheap measured improvement, about $\sim 1.4\times$. That A/B does not set the production priority order. For deployment, the first schema step is still to split `text` from the navigation collection so the hot structure/title/summary working set can remain cache-resident. After that, keep indexes and shard keys led by `tree_id`, add lean covering indexes where they serve real hot paths, and reserve subtree denormalization for measured `get_subtree` bottlenecks. The remaining validation is a cache-constrained Subset Pattern A/B that forces the inline-text layout out of cache while the split navigation layout fits.