CHECKLIST

MongoDB Query Performance Tuning Checklist

A practical checklist for diagnosing and improving slow queries on large MongoDB collections using indexes, query shape, projection, and explain-plan validation.

MongoDB Query Performance Tuning Checklist

Use this checklist to diagnose slow queries on large MongoDB collections and confirm that any change improves performance without creating new operational risk.

1) Capture the exact query pattern

- [ ] Record the query exactly as the application sends it.
- [ ] Include the full filter, sort, projection, skip/limit, and any aggregation stages.
- [ ] Note the collection name, expected result size, and typical request frequency.
- [ ] Identify whether the query is read-critical, dashboard-driven, API-facing, or batch-oriented.
- [ ] Confirm whether the issue appears only at scale or also on smaller datasets.

2) Confirm the real performance symptom

- [ ] Measure latency under production-like data volume.
- [ ] Compare average, p95, and p99 response times.
- [ ] Check CPU usage, memory pressure, and disk activity during the slow query.
- [ ] Look for lock contention, retry storms, or downstream timeouts.

- [ ] Determine whether the problem is read latency, sort cost, pagination cost, or over-fetching.

3) Validate the access path with explain()

- [ ] Run explain() on the exact query shape.
- [ ] Use realistic data, not a tiny development sample.
- [ ] Check whether the plan uses an index or falls back to a collection scan.
- [ ] Compare documents examined vs. documents returned.
- [ ] Review whether the query performs an in-memory sort.
- [ ] Confirm whether the scan order matches the requested sort order.
- [ ] Watch for an index being used technically, but inefficiently.

4) Check for common large-collection failure modes

- [ ] Missing index for the filter pattern.
- [ ] Index exists, but order does not match filter plus sort.
- [ ] Low-selectivity predicate scans too many documents.
- [ ] Large skip() offset causes pagination slowdown.
- [ ] Query returns full documents when only a few fields are needed.
- [ ] Aggregation or client-side processing forces unnecessary work.
- [ ] Sort happens after a large intermediate result is built.
- [ ] Joins, unwinds, or repeated lookups amplify the cost.

5) Review filter selectivity

- [ ] Identify the most selective predicates in the query.
- [ ] Move equality predicates ahead of range predicates when designing compound indexes.

- [ ] Verify that tenant, status, and similar high-value filters are placed first when appropriate.
- [ ] Check whether the query still returns too many candidates even with an index.
- [ ] Confirm that low-selectivity fields are not doing most of the filtering work.

6) Review sort behavior

- [ ] Determine whether the sort is required by the application.
- [ ] Check if the sort can be satisfied by the index order.
- [ ] Avoid in-memory sort for large result sets when possible.
- [ ] Confirm that descending or ascending order matches the index definition.
- [ ] Validate any ?latest records? or dashboard queries carefully.

7) Review projection and payload size

- [ ] Return only the fields the application actually uses.
- [ ] Avoid fetching large documents when a summary view is sufficient.
- [ ] Use projection to reduce network and deserialization overhead.
- [ ] Confirm that omitted fields do not break application logic.
- [ ] Check whether trimming the payload materially improves latency.

8) Review pagination strategy

- [ ] Avoid large-offset pagination with skip() when possible.
- [ ] Prefer cursor-based or keyset pagination for large result sets.
- [ ] Validate that pagination remains stable as the dataset grows.
- [ ] Ensure the pagination field is indexed and unique enough for consistent ordering.
- [ ] Test deeper pages, not just the first page.

9) Evaluate index design options

- [ ] Create or revise compound indexes around the real query pattern.
- [ ] Align index order with common filters and requested sort.
- [ ] Confirm the index supports the most expensive and frequent access path.
- [ ] Consider partial indexes when only a subset of documents is queried often.
- [ ] Consider sparse indexes only when missing values should be excluded.
- [ ] Avoid adding indexes that are not tied to a known workload.
- [ ] Check whether a new index increases write cost or storage too much.

10) Check whether schema shape is part of the problem

- [ ] Decide whether the query is slow because the model requires repeated lookups.
- [ ] Identify whether denormalization would reduce joins or extra reads.
- [ ] Review whether arrays, nested documents, or post-processing create overhead.
- [ ] Assess whether a schema change would be safer than piling on indexes.
- [ ] Make sure any schema change still supports the dominant read pattern.

11) Test candidate fixes safely

- [ ] Compare the current plan with the proposed plan.
- [ ] Validate the change against realistic data distribution.
- [ ] Test under representative concurrency.
- [ ] Measure both read improvement and write impact.
- [ ] Verify memory use and storage growth after the change.
- [ ] Check for plan regressions on similar queries.
- [ ] Roll out in a controlled environment before production.

12) Confirm operational safety before release

- [ ] Ensure the index is actually used in production-like conditions.
- [ ] Confirm the query no longer scans excessive documents.
- [ ] Verify that blocking sorts have been eliminated or reduced.
- [ ] Watch for slower inserts, updates, or deletes caused by new indexes.
- [ ] Recheck API latency after deployment.
- [ ] Monitor for regressions in dashboards, batch jobs, and background tasks.

Quick decision guide

- [ ] If the plan shows a collection scan, focus on index design first.
- [ ] If the plan uses an index but examines too many documents, review selectivity and index order.
- [ ] If the query sorts large result sets, make the sort indexable or reduce the result window.
- [ ] If the application fetches too much data, reduce the projection.
- [ ] If deep pages are slow, replace offset pagination with a cursor-based approach.
- [ ] If the data model forces repeated extra work, consider a schema redesign.

Final pre-production checklist

- [ ] Query shape captured and documented.
- [ ] Explain plan reviewed on realistic data.
- [ ] Index choice matched to filter and sort pattern.
- [ ] Projection minimized.
- [ ] Pagination strategy validated.

- [ ] Write cost and index size reviewed.
- [ ] Concurrency tested.
- [ ] Monitoring in place after deployment.

One-minute review

A MongoDB query is usually slow on large collections because it is doing too much work to find, sort, or return documents. The safest fixes are the ones that reduce work early: use the right index, align the sort with the access path, fetch fewer fields, and avoid pagination patterns that get slower as the dataset grows. Always verify the change with explain() and realistic production-like data before rollout.