CHECKLIST

NoSQL Indexing Strategies for High-Performance Queries Checklist

A practical checklist for planning, validating, and maintaining NoSQL indexes that support fast reads without creating unnecessary write overhead.

NoSQL Indexing Strategies for High-Performance Queries

Downloadable Checklist

Use this checklist to evaluate whether a NoSQL index is actually improving the query path you care about?and not just adding maintenance cost.

---

1) Start with real query patterns

- [ ] List the queries that are on the latency-critical path.
- [ ] Record the exact filters, sorts, and pagination method for each query.
- [ ] Identify which queries are tenant-scoped, user-scoped, or asset-scoped.
- [ ] Note which queries return full documents versus narrow projections.
- [ ] Remove assumptions based on schema shape alone; focus on actual access patterns.

2) Rank fields by selectivity and reuse

- [ ] Identify fields with high cardinality and strong filtering power.
- [ ] Identify fields with low cardinality, such as status or severity labels.
- [ ] Prefer indexing fields that are both selective and frequently reused.
- [ ] Avoid leading a compound index with a low-selectivity field unless the workload demands it.
- [ ] Confirm that the field order matches the way the database can use the index.

3) Match the index to the dominant query shape

- [ ] Choose the smallest index that supports the primary query path.
- [ ] Align equality filters, sort fields, and pagination fields in the right order.
- [ ] Put leading fields in the order the engine can exploit efficiently.
- [ ] Verify that the index supports filtering and sorting in one access path when possible.
- [ ] Avoid creating indexes that only partially match the query shape.

4) Evaluate the main index type

Single-field index

- [ ] Use when one highly selective field drives a dominant lookup.
- [ ] Confirm the query rarely depends on additional predicates or sort order.
- [ ] Avoid relying on it for low-cardinality fields.

Compound index

- [ ] Use when queries consistently combine filters, ordering, and pagination.
- [ ] Confirm the field order mirrors the query?s access pattern.

- [ ] Remove redundant compound indexes that differ only in trailing fields.

Covering index

- [ ] Use when the query can be answered from index entries alone.
- [ ] Limit coverage to the fields needed for common narrow projections.
- [ ] Confirm the added index size is worth the read-latency benefit.

Array or multikey-aware index

- [ ] Validate that array size and variability are acceptable.
- [ ] Estimate worst-case index growth per document.
- [ ] Confirm the read benefit exceeds the write and storage cost.

Partial or filtered index

- [ ] Use when only a subset of documents is queried frequently.
- [ ] Match the filter predicate to the application query semantics.
- [ ] Confirm the index remains useful if the data lifecycle changes.

5) Check write amplification and storage cost

- [ ] Estimate the write overhead added by each proposed index.
- [ ] Consider insert-heavy, update-heavy, and delete-heavy workloads separately.
- [ ] Account for memory or storage consumption by the index structure.
- [ ] Avoid adding indexes that duplicate coverage already provided elsewhere.
- [ ] Confirm the operational cost is acceptable before production rollout.

6) Validate under representative workload

- [ ] Test with production-like data volume and data distribution.
- [ ] Measure query latency, not just query success.
- [ ] Inspect the execution plan or equivalent index usage diagnostics.
- [ ] Confirm the query uses the intended index for the dominant path.
- [ ] Test both the happy path and the most common edge cases.
- [ ] Recheck performance after load increases, not only during small-scale testing.

7) Verify pagination and sorting behavior

- [ ] Confirm the index supports stable ordering for paginated results.
- [ ] Prefer cursor-based pagination for large datasets when supported.
- [ ] Ensure sort fields are included in the index in the correct position.
- [ ] Check that pagination does not cause broad scans or repeated work.
- [ ] Validate the index works for the same query across multiple tenants or partitions.

8) Review security and scoping implications

- [ ] Confirm tenant filters are included in the access path when required.
- [ ] Verify that authorization- or scope-related fields are not left to expensive scans.
- [ ] Check that index design does not encourage unsafe broad queries.
- [ ] Review any overlap between access control boundaries and query filters.
- [ ] Ensure sensitive data exposure is not widened by convenience-driven indexing.

9) Eliminate redundancy before production

- [ ] Identify indexes that are unused in real query traffic.
- [ ] Remove indexes that are superseded by a better compound index.
- [ ] Check whether two indexes differ only by trailing fields.
- [ ] Keep the index set small enough to understand and maintain.
- [ ] Revisit the index list after workload changes or new product features.

10) Operationalize ongoing maintenance

- [ ] Monitor index usage after deployment.
- [ ] Track write latency and ingestion throughput alongside read latency.
- [ ] Reassess indexes after major data growth or cardinality changes.
- [ ] Document why each index exists and which query it supports.
- [ ] Schedule periodic index review to remove stale or redundant entries.

Quick decision test

Before creating a new index, answer these questions:

- [ ] Does this index support a real, high-frequency query path?
- [ ] Does the field order match the query?s filters and sort order?
- [ ] Is the leading field selective enough to justify the index?
- [ ] Will the index reduce latency enough to offset write overhead?
- [ ] Have I tested it with representative data and load?
- [ ] Is there already an index that covers this path well enough?
- [ ] Can I explain exactly which query this index improves?

If you cannot answer yes to most of these, the index is probably not ready.

Example planning note

For a multi-tenant event system, a strong candidate index might be built around the most common path first: tenant-scoped recent events, optionally filtered by severity and sorted by event time. A second index can support a narrower lookup such as asset-specific investigation over a short time window. Avoid building many overlapping indexes unless each one clearly supports a distinct, frequent query.

Final pre-production checklist

- [ ] Query shape documented
- [ ] Field order validated
- [ ] Selectivity reviewed
- [ ] Write cost estimated
- [ ] Storage impact estimated
- [ ] Execution plan checked
- [ ] Representative load tested
- [ ] Redundant indexes removed
- [ ] Security boundaries reviewed
- [ ] Maintenance owner assigned

Notes

Keep the index set as small as possible while still supporting the queries that matter most. In NoSQL systems, the best index strategy is usually the one that matches real access patterns, remains selective, and does not create unnecessary maintenance overhead.