Think BIGGER

DEEP DIVE FÜR FACHLEUTE

MongoDB

Dokumentenorientierte NoSQL-Datenbank für flexible Datenmodelle

Autor	Erhard Rainer
Veröffentlicht	10.11.2016
Aktualisiert	06.07.2026
Umfang	ca. 23 Min. Lesezeit
Online	www.thinkbigger.at/wissen/mongodb/

thinkbigger.at

Business Intelligence · Machine Learning · KI-Agenten · Schulungen

Erhard Rainer · Schönaugasse 106 · 8020 Graz · info@think-bigger.at

Inhalt

0. Überblick (für Eilige) 2
1. Einordnung: Wann MongoDB die richtige Wahl ist 4
2. Architektur & Kernkonzepte 6
3. Datenmodellierung in der Praxis 8
4. Query-, Aggregation- und Index-Referenz 11
5. Konsistenz, Replikation, Transaktionen, Verfügbarkeit 14
6. Betrieb, Monitoring & Troubleshooting 16
7. Performance, Skalierung & Kostenkontrolle 18
8. Sicherheit, Governance & Compliance 21
9. Weiterführende Links + Uebungsaufgaben 23

MongoDB ist eine dokumentenorientierte NoSQL-Datenbank mit Fokus auf flexible Datenmodelle, hohe Entwicklergeschwindigkeit und horizontale Skalierung über Replica Sets und Sharding.

Diese Seite ist als praxisorientierte Referenz aufgebaut: Architektur, Datenmodellierung, Query- und Index-Design, Konsistenzprofile, Betrieb, Troubleshooting, Performance und Security.

> Hinweis zum Versionsstand: Diese Referenz ist auf **4. März 2026** ausgerichtet. Die offizielle MongoDB-Dokumentation führt zu diesem Zeitpunkt die Linie **8.2** als aktuelle Dokumentationslinie. Versionssensitive Aussagen sind datiert und als solche markiert.

0. Überblick (für Eilige)

0.1. Kurzbild

MongoDB ist stark, wenn du:

- Daten modelnah als Dokumente speichern willst.
- Produktanforderungen hast, die sich häufig weiterentwickeln.
- hohe Schreib- und Leselasten horizontal verteilen musst.
- mit bewusstem Index- und Query-Design arbeitest.

MongoDB ist schwach, wenn du:

- primär komplexe relationale Join-Welten mit starker Fremdschluessel-Logik brauchst.
- unkontrollierte Ad-hoc-Abfragen auf sehr grossen Datenmengen ohne Design-Disziplin zulassen willst.

0.2. Inhaltsverzeichnis (kurz)

- 0. Überblick (für Eilige)
- 1. Einordnung: Wann MongoDB die richtige Wahl ist
- 2. Architektur & Kernkonzepte
- 3. Datenmodellierung in der Praxis
- 4. Query-, Aggregation- und Index-Referenz
- 5. Konsistenz, Replikation, Transaktionen, Verfügbarkeit
- 6. Betrieb, Monitoring & Troubleshooting
- 7. Performance, Skalierung & Kostenkontrolle
- 8. Sicherheit, Governance & Compliance
- 9. Weiterführende Links + Uebungsaufgaben

0.3. Quick-Glossar

- **Document**: BSON-Objekt als zentrale Speichereinheit.
- **Collection**: Gruppe inhaltlich ähnlicher Dokumente.
- **Replica Set**: Mehrere Knoten mit Primary/Secondary-Replikation und automatischem Failover.
- **Oplog**: Replikationsprotokoll für Aenderungsereignisse.
- **Sharding**: Horizontale Verteilung von Daten über mehrere Shards.
- **Shard Key**: Feld oder Feldkombination für Datenverteilung.
- **Write Concern**: Haltbarkeits- und Replikationsgarantie für Schreibvorgaenge.

- **Read Concern**: Konsistenzniveau für Lesevorgaenge.
- **Read Preference**: Routingregel, von welchem Replikatknoten gelesen wird.
- **Aggregation Pipeline**: Stufenbasierte Datenverarbeitung mit `$match`, `$group`, `$project` usw.
- **TTL Index**: Index, der Dokumente nach Zeitkriterium automatisch entfernen kann.
- **Schema Validation**: Serverseitige Validierung, z. B. per JSON Schema.

0.4. Lernpfad

1. Erst Architektur und Konsistenzmodell verstehen (Abschnitte 1-2, 5).
2. Dann query-first Datenmodell und Index-Design aufbauen (Abschnitte 3-4).
3. Danach Betrieb und Monitoring festziehen (Abschnitt 6).
4. Zum Schluss Performance, Kosten und Security systematisch haerten (Abschnitte 7-8).

0.5. Mini-Projekt (durchgehend)

Wir nutzen ein durchgehendes Szenario:

- Plattform für Bestellungen, Zahlungen und Versandstatus.
- Hoher Schreibanteil in `orders`, `payments`, `shipment_events`.
- API-Anforderungen:
- Bestellung laden inkl. Positionen und Status in < 120 ms P95.
- Zahlungsstatus idempotent aktualisieren.
- Event-Historie pro Bestellung 90 Tage aufbewahren.
- Multi-Region-Leselast mit klaren Staleness-Regeln.

Dieses Mini-Projekt wird in mehreren Kapiteln weitergenutzt:

- Datenmodellierung (Embedding vs Referencing).
- Konsistenzprofile (write/read concern).
- Betriebsrunbooks (Replication-Lag, Chunk-Skew, Slow Queries).
- Performance und Kosten (Indexpflege, TTL, Tiering).

0.6. Profi-Heuristiken

- Starte niemals mit "wir speichern einfach JSON und schauen später".
- Jede kritische Query bekommt vor Go-Live einen passenden Index oder wird gestrichen.
- Dokumentgroesse und Array-Wachstum sind Produktionsrisiken, nicht nur Modelldetails.
- Transaktionen sind wertvoll, aber kein Ersatz für gutes Datenmodell.
- Sharding ist kein Schalter, sondern ein Betriebsmodell mit klaren Verantwortungen.

0.7. So setzt du es um - Checkliste

- Definiere 5 bis 10 harte API-Queries vor dem Schema-Design.
- Lege pro Query P95/P99-Ziele und erwartete Last fest.
- Dokumentiere pro Endpunkt `readConcern`, `writeConcern`, `readPreference`.
- Erstelle ein erstes Incident-Runbook für Replication-Lag und Slow Queries.
- Plane TTL-/Archivierungsstrategie vor Produktionsstart.

1. Einordnung: Wann MongoDB die richtige Wahl ist

1.1. Problemklassen, in denen MongoDB typischerweise stark ist

MongoDB passt sehr gut, wenn folgende Punkte zusammenkommen:

- dokumentzentrierte Domaene mit natuerlichen Aggregaten.
- hohe Aenderungsgeschwindigkeit der Anforderungen.
- Bedarf an horizontaler Skalierung für Lese- und Schreibpfade.
- Kombination aus operativem Store und flexiblen Auswertungen via Aggregation.

Typische Beispiele:

- E-Commerce Kataloge, Warenkoerbe, Bestellstatus.
- Event- und Aktivitaetsdaten.
- Content-Plattformen mit variablen Metadaten.
- SaaS-Anwendungen mit tenant-spezifischen Datenvarianten.

1.2. Wann ein relationales System oft besser passt

Ein RDBMS ist oft die robustere Standardwahl, wenn du brauchst:

- komplexe, dynamische Joins über viele Entitaeten.
- starke referenzielle Integrität über viele Tabellen.
- etablierte SQL-Reporting-Welten als Primaranforderung.
- sehr strikte, transaktionslastige Mehrzeilen-Workflows als Kern.

MongoDB kann Teile davon leisten, aber häufig mit hoeherem Modellierungs- und Betriebsaufwand.

1.3. MongoDB vs. andere NoSQL-Klassen (praxisnah)

- Gegen Key-Value Stores:
- MongoDB ist flexibler für reichhaltige Abfragen und Aggregationen.
- Gegen Wide-Column Stores:
- MongoDB ist oft einfacher für dokumentzentrierte Entwicklungsmodelle.
- Gegen Search-Engines:
- MongoDB ist operativer Primaerstore, keine reine Volltextplattform.

Entscheidend ist die Hauptfrage:

- Ist dein dominantes Problem Dokument-Lifecycle plus flexible Query/Index-Kombination?

1.4. Harte No-Go-Faelle

- Kein Team kann oder will Query-/Index-Disziplin durchsetzen.
- Datendesign basiert auf ungebremstem Dokumentwachstum.
- Join-zentrierte Analytics werden als Hauptlast erwartet.
- Es existiert kein Betriebsmodell für Replica Sets, Backups und Security-Hardening.

> **Anti-Pattern:** "MongoDB ist schemalos, also brauchen wir kein Modell." Ergebnis: inkonsistente Daten, indexfeindliche Queries, steigende Incident-Rate.

1.5. Entscheidungsmatrix (kompakt)

Kriterium	MongoDB passt	MongoDB passt eher nicht
Datenstruktur	Dokument-/Aggregate-zentriert	Stark normalisierte Join-Welt
Aenderungsrate	Hoch, iterativ	Stabil, seltene Schemaaenderungen
Skalierung	Horizontaler Ausbau geplant	Single-Node reicht langfristig
Query-Muster	Definierbar und indexierbar	Stark unvorhersehbare Ad-hoc-Last
Teamreife	Betrieb + Query-Disziplin vorhanden	"Schema egal, Index später"

1.6. Einfuehrungsstrategie statt Big Bang

Bewaehrte Reihenfolge:

1. Ein klar abgegrenzter Service als Pilot.
2. API-Query-Katalog und Index-Review als Pflicht.
3. Betriebsfaehigkeit mit Backups, Monitoring, Rollenmodell vor Feature-Ausweitung.
4. Erst danach breite Ausrollung.

1.7. So setzt du es um - Checkliste

- Schreibe ein 1-seitiges ADR: warum MongoDB, warum nicht Alternative X.
- Definiere explizite No-Go-Abfragen für Release-Reviews.
- Verankere einen Index-Review im Pull-Request-Prozess.
- Setze klare Exit-Kriterien (wann ein Service nicht in MongoDB bleiben soll).

1.8. Typische Fehlentscheidungen in der Auswahlphase

1. "Wir wählen MongoDB, weil wir kein Schema brauchen."
2. "Wir wählen MongoDB, weil joins unbequem sind."
3. "Wir wählen MongoDB, weil Sharding später alles löst."

Diese Begründungen sind schwach, weil sie kein messbares Ziel enthalten.

Besser ist:

1. Belastbare Query- und Lastannahmen.
2. Dokumentierte Konsistenzanforderungen je Endpunkt.
3. Betriebsfaehigkeit mit klaren SLOs vor Produktionsgo-live.

1.9. Readiness-Check vor der Produktiventscheidung

- Gibt es einen API-Query-Katalog mit Volumenannahmen?
- Gibt es einen ersten Indexentwurf inklusive Explain-Nachweis?
- Gibt es ein Security-Baseline-Konzept (RBAC, TLS, Audit)?
- Gibt es Runbooks für Failover und Slow-Query-Incidents?
- Gibt es einen datenbasierten Vergleich mit mindestens einer Alternative?

2. Architektur & Kernkonzepte

2.1. Kernbausteine

MongoDB basiert operativ auf:

- `mongod` als Datenbankprozess.
- **Replica Sets** für Redundanz und Failover.
- **Sharding** über `mongos` Router plus Config-Server-Replica-Set.
- **WiredTiger** als zentrale Storage-Engine in Self-Managed-Deployments.

2.2. Dokumentmodell und BSON-Grenzen

Dokumente sind BSON-basiert und haben u. a. diese wichtige Grenze:

- Maximale Dokumentgroesse: 16 MiB.

Praxisfolge:

- Dokumentdesign muss Wachstumspfad mitdenken.
- Unbegrenzte Arrays in einem Dokument sind ein klares Risiko.

2.3. WiredTiger, Journal und durable Writes

WiredTiger kombiniert:

- Speicherpuffer.
- Kompression.
- Journal für Haltbarkeit.

Write Concern bestimmt, wann ein Write für den Client als bestätigt gilt. Ohne bewusstes Write-Concern-Profil sind Haltbarkeitsaussagen unsauber.

2.4. Replica Set intern: Primary, Secondary, Elections

- Ein Knoten ist Primary (nimmt regulaere Writes an).
- Secondaries replizieren asynchron über Oplog-Eintraege.
- Bei Ausfall des Primary erfolgt eine Wahl.

Wichtige Betriebsfolgen:

- Failover-Zeiten beeinflussen API-Fehlerbilder.
- Replication-Lag beeinflusst Lese-Staleness bei Secondary-Reads.

2.5. Oplog und Replikationsfenster

Der Oplog ist ein begrenztes Aenderungsfenster. Wenn ein Secondary zu lange hinterherhaengt, kann Re-Sync nötig werden.

Deshalb:

- Oplog-Dimensionierung ist Teil von Capacity-Planung.
- Nicht nur Datenvolumen, auch Recovery-Zeitfenster beachten.

2.6. Sharding-Architektur

Sharded Cluster bestehen aus:

- `mongos` Routern (Client-Einstieg).
- Config-Server-Replikatset (Metadaten, Chunk-Verteilung).
- mehreren Shards (jeweils meist Replica Sets).

Sharding löst Lastverteilung, erhöht aber Betriebskomplexitaet.

2.7. Shard Key Design als Schluesselentscheidung

Ein guter Shard Key:

- verteilt Last breit.
- vermeidet Hotspots.
- passt zu den wichtigsten Query-Pradikaten.

Typische Muster:

- Hashed Key für gleichmaessige Verteilung.
- Ranged Key für Bereichsabfragen.
- Compound Key für Balance aus Routing und Query-Mustern.

> **Anti-Pattern:** Monoton steigender Shard Key ohne Gegenmassnahmen. Ergebnis: Schreib-Hotspot auf wenigen Chunks.

2.8. Read- und Write-Pfad im Alltag

Write-Pfad (vereinfacht):

1. Write trifft Primary.
2. Primary schreibt lokal.
3. Replikation auf Secondaries.
4. Bestätigung gemaess Write Concern.

Read-Pfad (vereinfacht):

1. Routing gemaess Read Preference.
2. Rückgabe gemaess Read Concern.
3. Ergebnis kann je nach Profil stale oder majority-bestaetigt sein.

2.9. Mini-Projekt: Zielarchitektur

Für die Bestellplattform:

- Start mit Replica Set (3 Nodes, ungerade Stimmenzahl).
- kritische Writes mit `w: "majority"`.
- Leseprofile:
- Checkout: Primary.
- Historie/Reporting-nahe APIs: Primary Preferred oder Secondary nach Risikoanalyse.
- Sharding erst bei messbarem Need, nicht praeventiv.

2.10. So setzt du es um - Checkliste

- Dokumentiere Topologie inkl. Wahlquorum und Failure-Domains.
- Definiere Oplog-Fenster anhand geplanter Ausfall- und Recovery-Szenarien.
- Setze Read/Write-Profile pro API-Endpunkt statt globaler Defaults.
- Plane Shard-Key-Review vor jeder grossen Datenmengensteigerung.

2.11. Failure-Domains bewusst planen

Ein stabiles Clusterdesign trennt Fehlerzonen:

- unterschiedliche Hosts/Availability-Zonen.
- keine gemeinsamen Single Points für Storage und Netzwerk.
- klare Regeln für Wahlquorum und Prioritaeten.

Ziel:

- Ausfall einer Zone darf kein unkontrolliertes Datenrisiko erzeugen.

2.12. Architektur-Review vor jedem größeren Wachstumsschritt

Wenn Last oder Datenvolumen deutlich steigen, sollte ein kurzer Architekturreview verpflichtend sein.

Reviewfragen:

- Ist das Oplog-Fenster noch ausreichend?
- Stimmen die aktuellen Read-/Write-Concern-Profile noch mit den API-Anforderungen?
- Führt das Datenwachstum zu hot collections oder hot shards?
- Müssen Archivierung oder neue Zugriffspfade vor dem Wachstum eingefuehrt werden?

3. Datenmodellierung in der Praxis

3.1. Query-first statt Entity-first

In MongoDB startest du nicht mit ER-Modell-Denken, sondern mit Zugriffspfaden:

- Welche API braucht welche Felder?
- Welche Filter und Sortierungen sind kritisch?
- Welche Dokumentgrenzen stabilisieren Latenz und Kosten?

3.2. Embedding vs Referencing

Embedding ist stark bei:

- one-to-few Beziehungen.
- Daten, die fast immer zusammen gelesen werden.
- klar begrenzten Unterstrukturen.

Referencing ist stark bei:

- one-to-many mit grossem Wachstum.
- getrennten Lebenszyklen.
- getrennter Schreiblast.

3.3. Dokumentgroesse und Wachstumsbudget

Setze pro Collection ein Budget:

- Zielgroesse im Regelfall.
- Warnschwelle.
- harte Grenze.

Damit vermeidest du:

- unerwartete Dokument-Migrationseffekte.
- ueberdimensionierte Updates.
- Performance-Drift durch unkontrolliertes Wachstum.

3.4. Array-Design: kontrolliert statt offen

Unbegrenzte Arrays sind in operativen Collections häufig problematisch.

Besser:

- begrenzte In-Dokument-Historie.
- alte Eintraege in separate Event-Collection auslagern.
- für Verlaufsdaten oft Bucket- oder Time-Series-Muster nutzen.

3.5. Schema Validation trotz Flexibilität

Schemaflexibilitaet ist kein Argument gegen Mindestregeln. Nutze JSON-Schema-Validierung für:

- Pflichtfelder.
- Datentypen.
- Grenzen (z. B. Stringlaengen, Enum-Werte).

So reduzierst du Datenqualitätsprobleme ohne Entwicklungsagilitaet zu verlieren.

3.6. Schema-Versionierung

Praktisches Muster:

- Feld `schemaVersion` in Dokumenten.
- Leser und Schreiber verstehen mindestens zwei Versionen im Übergang.
- Backfills und Migrationen als explizite Jobs planen.

3.7. Time Series Collections

Für Messdaten sind Time-Series-Collections oft besser als freihaendig modellierte Event-Collections:

- effizientere Speicherung.
- optimierte Zeitfensterabfragen.
- integrierte Zeitreihenlogik.

3.8. Bucket Pattern für Event-Daten

Wenn klassische Collection plus grosses Eventvolumen:

- Events pro Entitaet und Zeitfenster bucketn.
- Query-Fenster und Schreiblast gegeneinander ausbalancieren.

Ziel:

- weniger Dokumentfragmentierung.
- kontrollierbare Indexgroesse.
- gute Balance aus Write- und Read-Kosten.

3.9. Mini-Projekt: Modellierung der Bestellplattform

Beispielhafte Kerncollections:

- `orders`:
- eingebettete Positionen und Summen.
- aktueller Status, Kundensegment, Zeitstempel.
- `payment_events`:
- referenziert `orderId`.
- idempotenter `providerEventId`.
- TTL auf technische, kurzlebige Rohereignisse.
- `shipment_events`:
- Eventverlauf separat, da potenziell gross.

Beispielstruktur:

```
{
  "_id": "ord_2026_000981",
  "customerId": "cust_1042",
  "status": "paid",
  "items": [
    { "sku": "A-100", "qty": 2, "price": 49.9 },
    { "sku": "B-210", "qty": 1, "price": 12.5 }
  ],
  "totals": { "gross": 112.3, "currency": "EUR" },
  "createdAt": { "$date": "2026-03-01T10:13:00Z" },
  "updatedAt": { "$date": "2026-03-01T10:15:22Z" },
  "schemaVersion": 3
}
```

3.10. Anti-Pattern-Kasten

> **Anti-Pattern:** "Wir legen alle Lebenszyklusdaten in ein einziges Order-Dokument." Folge: Dokumentwachstum, schwierige Teilupdates, Konflikte unter Last.

3.11. So setzt du es um - Checkliste

- Schreibe pro Collection die Top-Queries als Vertrag nieder.
- Definiere Embedding/Referencing bewusst mit Wachstumsannahmen.
- Setze Schema Validation für Pflicht- und Risiko-Felder.
- Dokumentiere Migrationsstrategie inkl. `schemaVersion`.
- Lege für jede Eventdaten-Collection eine TTL- oder Archivierungsregel fest.

3.12. Pattern-Katalog für den Alltag

- **Outbox-Muster**: fachliche Änderung plus Integrationsereignis robust koppeln.
- **Polymorphe Dokumente mit Guardrails**: flexibel bleiben, aber Pflichtfelder erzwingen.

- **Status-Transition-Muster**: nur erlaubte Zustandswechsel serverseitig absichern.
- **Read-optimierte Projektionen**: für häufige API-Pfade gezielte, schlanke Dokumentformen.

3.13. Migrationsstrategie ohne Downtime

Bewaehrte Reihenfolge:

1. Leser auf alte und neue Form vorbereiten.
2. Schreiber auf neue Form umstellen.
3. Backfill kontrolliert in Batches.
4. Altfelder nach Beobachtungsfenster abbauen.

Wichtig:

- Migrationen immer mit Performance-Monitoring begleiten.
- Backfills nie ohne Lastbegrenzung in Peak-Zeiten starten.

4. Query-, Aggregation- und Index-Referenz

4.1. CRUD-Grundmuster

Typische Kernoperationen:

```
db.orders.insertOne({
  _id: "ord_2026_000982",
  customerId: "cust_2044",
  status: "created",
  createdAt: new Date(),
  updatedAt: new Date(),
  schemaVersion: 3
});

db.orders.find(
  { customerId: "cust_2044", status: { $in: ["created", "paid"] } },
  { _id: 1, status: 1, updatedAt: 1 }
).sort({ updatedAt: -1 }).limit(20);

db.orders.updateOne(
  { _id: "ord_2026_000982", status: "created" },
  {
    $set: { status: "paid", updatedAt: new Date() },
    $inc: { paymentRetryCount: 1 }
  }
);
```

4.2. Indexarten, die du im Alltag brauchst

- Single Field Index.
- Compound Index.
- Multikey Index (Arrays).
- Partial Index.
- TTL Index.
- Wildcard Index (gezielt, nicht inflationaer).
- Text Index (begrenzter Suchfall).

4.3. Compound Index Design (ESR-Regel)

Pragmatische Leitlinie:

- **E**quality-Felder zuerst.
- **S**ort-Felder danach.
- **R**ange-Felder zuletzt.

Das ersetzt kein Explain-Review, ist aber ein gutes Startmuster.

4.4. Aggregation Pipeline: operativer Standard

Aggregation ist nicht nur Analytics, sondern oft API-Rueckgrat.

Beispiel:

```
db.orders.aggregate([
  { $match: { createdAt: { $gte: ISODate("2026-03-01T00:00:00Z") } } },
  { $group: { _id: "$status", totalGross: { $sum: "$totals.gross" }, cnt: { $sum: 1 } } },
  { $sort: { totalGross: -1 } }
]);
```

Wichtig:

- fruehes `$match` zur Datenreduktion.
- Projektionen bewusst klein halten.
- speicherintensive Stufen messen und begrenzen.

4.5. Explain als Pflicht, nicht als Ausnahme

Prüfe für kritische Queries:

- `winningPlan`.
- `keysExamined` vs `docsExamined`.
- ob ein Sort ohne passenden Index ausgeführt wird.
- ob Stage-Ketten zu viele Dokumente durchschleusen.

4.6. Query- und Index-Lifecycle

Query-Patterns ändern sich mit Features. Deshalb:

- regelmäßiger Abgleich "aktive Queries vs bestehende Indizes".
- ungenutzte Indizes kontrolliert entfernen.
- neue Indizes vorab gegen Write-Overhead bewerten.

4.7. Transaktionen in Query-Pfaden

MongoDB unterstützt Multi-Document-Transaktionen, auch in sharded Umgebungen. Nutze sie gezielt für echte atomare Mehrdokument-Notwendigkeiten.

Trade-off:

- höhere Latenz.
- mehr Ressourcenbindung.
- komplexere Fehlerbehandlung.

4.8. Typische Query-Anti-Pattern

- Sortierung ohne passenden Index.
- breite Regex-Filter ohne selektives Prefix.
- "alles über Aggregation lösen" ohne Data-Shape-Disziplin.
- unlimitierte Endpunktabfragen ohne Pagination.

> **Anti-Pattern:** Ein API-Endpunkt hat keine feste Sortierung und kein Limit. Folge: unvorhersehbare Lastspitzen und Timeout-Risiko.

4.9. Mini-Projekt: Indexset für Kern-APIs

Beispiel:

```
db.orders.createIndex({ customerId: 1, updatedAt: -1 });
db.orders.createIndex({ status: 1, createdAt: -1 });
db.payment_events.createIndex({ providerEventId: 1 }, { unique: true });
db.payment_events.createIndex({ createdAt: 1 }, { expireAfterSeconds: 60 * 60 * 24 * 90 });
```

4.10. So setzt du es um - Checkliste

- Für jede kritische Query liegt ein Explain-Review vor.
- Kein Produktionsendpunkt ohne Pagination-Limit.
- Compound-Index-Reihenfolge ist dokumentiert begründet.
- TTL-Indizes sind fachlich abgestimmt und getestet.
- Query- und Index-Lifecycle ist Bestandteil der Release-Routine.

4.11. Query-Qualitaetsgates für Releases

Setze für neue Endpunkte feste Gates:

- Explain ohne kritische Scan-Auffaelligkeiten.
- Nachweisbarer Index-Fit für Filter + Sortierung.
- Lasttest gegen erwartete QPS-Spitze.
- klare Pagination-Grenzen und API-Limits.

Ohne diese Gates wird Query-Schuld oft erst im Incident sichtbar.

4.12. Aggregation-Troubleshooting kompakt

Wenn Aggregation langsam wird:

1. frühe `$match`-Stufen nach vorne ziehen.
2. unnötige Felder per `$project` früh ausschneiden.
3. Group-Cardinality und Speicherbedarf prüfen.
4. schwere Auswertung ggf. in vorberechnete Read-Modelle verlagern.

5. Konsistenz, Replikation, Transaktionen, Verfügbarkeit

5.1. Drei Hebel, die zusammen gedacht werden müssen

- `writeConcern`: Wann gilt ein Write als bestätigt?
- `readConcern`: Welcher Konsistenzgrad wird gelesen?
- `readPreference`: Von welchem Knotentyp wird gelesen?

Nur die Kombination definiert das reale Verhalten.

5.2. Write Concern in der Praxis

Typische Profile:

- `w: 1`: schnell, aber mit geringerem Replikationsschutz.
- `w: "majority"`: höhere Sicherheit gegen Datenverlust bei Ausfall.

Empfehlung:

- Kernprozesse standardmaessig `w: "majority"`.
- bewusstes Abweichen nur mit dokumentiertem Risiko.

5.3. Read Concern in der Praxis

Wichtige Abstufungen:

- `local`: schnell, ggf. weniger streng.
- `majority`: liest bestätigte Daten auf Mehrheitsbasis.
- `snapshot` in Transaktionen für konsistente Sicht auf mehrere Reads.

5.4. Read Preference und Staleness-Risiko

Optionen wie `primary`, `primaryPreferred`, `secondary`, `secondaryPreferred`, `nearest` sind kein Performance-Trick allein, sondern fachliche Entscheidung.

Frage pro API:

- Darf diese Antwort zeitlich hinterherhaengen?
- Wenn ja, wie viel?

5.5. Causal Consistency und Sessions

Bei Workflows mit "read-your-own-write"-Erwartung können causally consistent sessions wichtig sein. Das verhindert ungewollte Ruecksprueenge in Nutzerfluesse nach unmittelbar vorangegangenem Write.

5.6. Retryable Writes und Idempotenz

Retryable Writes helfen bei transienten Fehlern. Aber:

- fachlich sichere Wiederholung braucht idempotente Schluessel/Operationen.

Beispiel:

- `providerEventId` als eindeutiger technisch-fachlicher Idempotenzanker.

5.7. Transaktionen: wann ja, wann nein

Transaktionen sind sinnvoll bei:

- atomaren Mehrdokument-Updates mit echter Fachnotwendigkeit.

Transaktionen sind ueberdimensioniert bei:

- "nur zur Sicherheit" statt gezielter Datenmodellierung.

Regel:

- erst Modell verbessern, dann Transaktion dort einsetzen, wo sie unvermeidbar ist.

5.8. Ausfallszenarien sauber modellieren

Mindestszenarien für Tests:

- Primary fällt während Write-Spitze aus.
- Secondary-Lag steigt in Peak-Zeiten.
- Netzwerkpartition mit temporaerem Quorumverlust.

Nur so werden Konsistenzentscheidungen belastbar.

5.9. Mini-Projekt: Konsistenzprofil je Endpunkt

- `POST /payments/webhook`:
- `writeConcern: "majority"`, idempotente Upserts.
- `GET /orders/{id}` im Checkout:
- `readPreference: primary`, `readConcern: majority`.
- `GET /orders/{id}/events` für Kundenhistorie:
- `readPreference: secondaryPreferred`, falls geringe Staleness akzeptabel.

5.10. Anti-Pattern-Kasten

> **Anti-Pattern:** "Wir stellen readPreference auf secondary, dann ist alles schneller." Folge: inkonsistente Nutzererfahrung bei frischen Writes.

5.11. So setzt du es um - Checkliste

- Definiere je API-Endpunkt ein explizites Konsistenzprofil.
- Teste Failover und Retry-Verhalten automatisiert.
- Idempotenzschluessel für externe Events als Pflicht.
- Ueberwache Replication-Lag und setze harte Alarmgrenzen.
- Dokumentiere fachlich erlaubte Staleness je Lesepfad.

5.12. Konsistenzprofile als API-Vertrag

Pragmatisches Template pro Endpunkt:

- Datenkritikalitaet (hoch, mittel, niedrig).
- erlaubte Staleness (z. B. 0 s, 2 s, 30 s).
- `writeConcern`, `readConcern`, `readPreference`.
- Retry-Regeln und Timeout-Obergrenzen.

- erwartetes Verhalten bei Primary-Failover.

Damit wird Konsistenz nicht implizit, sondern versionierbar und testbar.

5.13. Failure-Drills als Pflicht

Mindestens quartalsweise:

1. geplanter Primary-Switch unter Lasttestbedingungen.
2. simulierte Lag-Spitze eines Secondary.
3. Verifikation der API-Auswirkungen gegen definierte SLOs.

Ergebnis jedes Drills:

- Messwerte.
- erkannte Luecken.
- konkrete Verbesserungsmassnahmen mit Termin.

6. Betrieb, Monitoring & Troubleshooting

6.1. Betriebsgrundsatz

MongoDB ist kein "set and forget"-System. Stabiler Betrieb braucht:

- klare SLOs.
- reproduzierbare Runbooks.
- trendbasierte Metrikbeobachtung.

6.2. Kernmetriken für den Alltag

- P95/P99-Latenz pro API und Query-Gruppe.
- Replication-Lag pro Secondary.
- Oplog-Fenster und Replizierungszustand.
- Anzahl und Dauer langsamer Operationen.
- Cache-/Memory-Druck und I/O-Latenz.
- Queueing/Locks im Systemstatus.

6.3. Diagnose-Werkzeuge

- `db.serverStatus()`
- `db.currentOp()`
- Profiler (`system.profile`) für gezielte Tiefenanalyse.
- Explain-Plans für konkrete Query-Muster.
- Replikationsstatus über `rs.status()`.
- in Sharding-Umgebungen zusätzlich Balancer-/Chunk-Sicht.

6.4. Runbook: Slow Queries

1. Betroffene API und Query-Shape identifizieren.
2. Explain ausführen und `docsExamined` vs `nReturned` prüfen.
3. Sort- und Filterfelder gegen Index abgleichen.

4. Gegebenenfalls Index hinzufuegen oder Query umformen.
5. Ergebnis unter Lastprofil gegen P95/P99 erneut messen.

6.5. Runbook: Replication-Lag steigt

1. Lag-Knoten und Zeitraum eingrenzen.
2. I/O- und Ressourcenengpaesse prüfen.
3. Oplog-Fenster gegen Lag-Dauer abgleichen.
4. Hot Queries und Write-Spitzen korrelieren.
5. Sofortmassnahmen und mittelfristige Ursachenbehebung trennen.

6.6. Runbook: Sharding-Ungleichgewicht

Typische Signale:

- einzelne Shards deutlich heisser.
- unausgeglichene Chunk-Verteilung.
- punktuelle Latenzspitzen auf bestimmten Routerpfaden.

Diagnose:

1. Query-Muster und Shard-Key-Selektivitaet prüfen.
2. Chunk-Verteilung und Balancer-Zustand analysieren.
3. Hotspot-Ursache dokumentieren: Datenmodell, Key-Wahl oder Lastmuster.

6.7. Backup, Restore, Recovery

Produktionsreif bedeutet:

- regulaere Backups mit dokumentierten RPO/RTO-Zielen.
- Restore-Tests als Pflicht, nicht nur Backup-Erfolgsmeldung.
- für point-in-time Anforderungen Oplog-/Journal-Strategie abgestimmt.

6.8. Mini-Projekt: Incident "Checkout zeigt alten Zahlungsstatus"

Symptom:

- Zahlung eingegangen, API zeigt noch `pending`.

Prüfung:

- Read Preference für Checkout-Endpunkt auf Secondary?
- Replication-Lag zur Incident-Zeit?
- Write Concern im Payment-Pfad zu schwach?

Korrektur:

- Checkout auf `primary + majority`.
- Historienendpunkte getrennt auf staleness-tolerantes Profil.
- Lag-Alarm schaerfen.

6.9. So setzt du es um - Checkliste

- Pro Kern-Collection existiert ein kurzes Runbook.

- Top-10 Queries werden regelmäßig mit Explain geprüft.
- Lag-, SlowQuery- und Error-Alarme enthalten konkrete nächste Aktion.
- Backup und Restore werden mindestens quartalsweise geprobt.
- Postmortems führen zu messbaren Folgeaenderungen.

6.10. Alert-Design ohne Alarmmuedigkeit

Ein Alarm ist nur sinnvoll, wenn er eine klare Aktion auslöst.

Gute Regeln:

- Schwellwerte mit Zeitfenster kombinieren (nicht nur einzelne Peaks).
- technische und fachliche Metrik koppeln.
- Warnung und Eskalation sauber trennen.

Beispiel:

- "Replication-Lag > 5 s für 10 Minuten" ist oft besser als "Lag einmalig > 5 s".

6.11. Troubleshooting-Matrix (Kurzform)

- **Symptom:** Hohe API-Latenz.
- **Erste Prüfung:** Slow Queries, Explain, Index-Fit.
- **Sofortmassnahme:** Query begrenzen oder Last drosseln.
- **Symptom:** Stale Daten in UI.
- **Erste Prüfung:** Read Preference, Lag, Concern-Profil.
- **Sofortmassnahme:** kritischen Endpunkt auf Primary/majority.
- **Symptom:** Unerwartete Write-Fehler.
- **Erste Prüfung:** Quorumzustand, Primary-Status, Retry-Logik.
- **Sofortmassnahme:** Failover-Stabilisierung und Wiederholungsregeln aktivieren.

6.12. Runbook-Qualitaet messen

Ein gutes Runbook ist:

- in < 5 Minuten auffindbar.
- in < 15 Minuten anwendbar.
- von On-Call ohne Spezialwissen ausführbar.
- nach jedem Incident verbessert.

7. Performance, Skalierung & Kostenkontrolle

7.1. Performance beginnt im Modell

Die größten Hebel kommen meist nicht aus "Tuning-Schaltern", sondern aus:

- sauberem Dokumentzuschnitt.
- passenden Indizes.
- klaren Query-Grenzen.

7.2. Indexkosten aktiv managen

Jeder Index kostet:

- Speicher.
- Write-Performance.
- Build-/Wartungszeit.

Daher:

- Index nur für echte Query-Pfade.
- ungenutzte Indizes entfernen.
- grosse Indexaenderungen in Lastfenstern planen.

7.3. Working Set und Speicherrealitaet

Wenn Working Set deutlich größer als verfügbarer effektiver Cache ist:

- steigen I/O-Wartezeiten.
- Latenz variiert stärker.

Massnahmen:

- heisse Daten isolieren.
- Dokument-/Indexfussabdruck verkleinern.
- kalte Daten per TTL/Archivierung entlasten.

7.4. Skalierungspfad

Skalierungsreihenfolge in vielen Teams:

1. Query- und Indexdisziplin.
2. vertikale Kapazitaet sauber ausnutzen.
3. erst dann horizontal über Sharding erweitern.

Sharding ohne vorherige Query-Hygiene skaliert Probleme mit.

7.5. Shard-Key-Drift und Re-Sharding-Risiko

Ein heute guter Shard Key kann morgen kippen. Frühe Signale:

- zunehmend ungleiche Lastverteilung.
- steigende Hot-Chunks.
- mehr scatter-gather für Kernqueries.

Dann:

- nicht nur Infrastruktur hochdrehen.
- zuerst Query-/Key-Ursache technisch belegen.

7.6. Kostenkontrolle durch Datenklassen

Pragmatisches Tiering:

- **Tier A**: kritische operative Daten, volle Replikationsgarantien.
- **Tier B**: wichtige, aber weniger latenzkritische Daten.

- **Tier C**: kurzlebige Roh-Events mit TTL und guenstigerem Profil.

7.7. Mini-Projekt: Kostenhebel

In der Bestellplattform:

- `payment_events_raw` mit 90-Tage-TTL.
- verdichtete Aggregatdaten in separater Collection ohne Rohpayload.
- teure Long-Tail-Queries als asynchrones Reporting statt Online-API.

7.8. Benchmarking mit Aussagekraft

Ein brauchbarer Lasttest bildet ab:

- reale Query-Mischung.
- reale Dokumentgroessen.
- reale Konsistenzprofile.
- Lastspitzen und Recovery-Phasen.

Messe mindestens:

- P95/P99.
- Timeout-Raten.
- Replikationsverhalten unter Peak.

7.9. Anti-Pattern-Kasten

> **Anti-Pattern:** "Wir fuegen für jedes Feature sofort einen Index hinzu." Folge: Write-Pfad wird langsam, Betriebskosten steigen schleichend.

7.10. So setzt du es um - Checkliste

- Monatliches Index-Audit mit Nutzungsnachweis.
- Working-Set-Schaetzung und Memory-Headroom dokumentiert.
- Shard-Key-Qualitaet regelmäßig anhand realer Queries bewertet.
- Datenklassen mit TTL/Archivierungsregeln verbindlich eingefuehrt.
- Performance-Regressionstest als Release-Gate aktiv.

7.11. Kapazitaetsplanung in der Praxis

Plane nicht nur Datenvolumen, sondern den gesamten Systemverkehr:

- Write-Rate plus erwartete Spitzenfaktoren.
- Read-Rate je Endpunktklasse.
- Indexwachstum pro Monat.
- Replikations- und Backup-Overhead.

Sinnvolle Reserve:

- nicht nur für Normalbetrieb, sondern auch für Recovery-Phasen und Backfills.

7.12. Performance-Gate für CI/CD

Vor jedem größeren Release:

1. Vergleich P95/P99 gegen letzte stabile Version.
2. Vergleich Write- und Read-Fehlerquote.
3. Vergleich ressourcenkritischer Metriken (I/O, Cache-Druck, Lag).

Abbruchkriterien müssen vorab festgelegt sein, nicht erst im Incident.

8. Sicherheit, Governance & Compliance

8.1. Sicherheitsbaseline

Für Produktion mindestens:

- starke Authentifizierung.
- rollenbasierte Autorisierung.
- TLS in Transit.
- Verschluesselung at rest (je nach Plattform/Anforderung).
- Auditierbarkeit für sensible Operationen.

8.2. Authentifizierung und Identität

MongoDB unterstützt u. a. SCRAM und x.509-basierte Varianten. Wichtig ist nicht nur der Mechanismus, sondern der Prozess:

- keine Shared Accounts für Anwendungen.
- Serviceidentitaeten je Dienst.
- Schlüssel- und Passwortrotation planbar und geuebt.

8.3. RBAC und Least Privilege

Definiere Rollen pro Anwendungspfad:

- reine Leserollen.
- schreibende Rollen für definierte Collections.
- stark eingeschränkte Admin-Rollen.

Vermeide:

- globale Rollen als schnelle Abkuerzung.

8.4. TLS und Netzwerksegmentierung

- TLS für Client- und internen Verkehr aktivieren.
- Zugang auf trusted Netzsegmente begrenzen.
- administrative Schnittstellen zusätzlich absichern.

8.5. Auditing und Nachvollziehbarkeit

Audit Logging ist zentral für Compliance und Incident-Forensik. Aufzeichnen solltest du mindestens:

- Authentifizierungsereignisse.

- Rollen-/Rechteaenderungen.
- kritische Datenoperationen auf sensiblen Collections.

8.6. Schema Governance

Governance bedeutet auch Datendisziplin:

- Schema Validation aktivieren.
- Feldstandards dokumentieren.
- Breaking-Changes über Versionierung und Migrationsfenster steuern.

8.7. Datenschutz und Datenlebenszyklus

- sensible Felder klassifizieren.
- Aufbewahrungsfristen technisch erzwingen (TTL/Archivierungsprozesse).
- Loeschanforderungen reproduzierbar umsetzen.

8.8. Mini-Projekt: Rollenmodell

Beispiel:

- `order_reader_role`: `find` auf `orders`, `shipment_events`.
- `payment_writer_role`: `insert/update/find` auf `payment_events`.
- `ops_auditor_role`: eingeschränkte Diagnose- und Audit-Sicht.

8.9. So setzt du es um - Checkliste

- Alle produktiven Zugriffe laufen über individuelle Serviceidentitaeten.
- RBAC folgt Least Privilege und wird quartalsweise rezertifiziert.
- TLS ist aktiv und Zertifikatsrotation getestet.
- Audit Logging ist für kritische Pfade eingeschaltet.
- Datenklassifikation ist mit TTL/Archivierung technisch gekoppelt.

8.10. Governance als Operating Model

Sicherheit wird stabil, wenn sie als zyklischer Prozess organisiert ist:

- monatlicher Security-Review der Betriebsmetriken.
- quartalsweise Rollenrezertifizierung.
- definierte Freigabeprozesse für Schema- und Rechteaenderungen.
- Nachweisfaehigkeit für Audits ohne Ad-hoc-Aufwand.

8.11. Security-Drills für reale Handlungsfaehigkeit

Uebungsszenarien:

1. kompromittiertes technisches Konto.
2. abgelaufenes Zertifikat.
3. unautorisierter Zugriff auf sensitive Collection.

Ziel je Drill:

- Erkennungszeit messen.

- Reaktionskette testen.
- Dokumentation und Rollenmodell nachschaerfen.

9. Weiterführende Links + Uebungsaufgaben

9.1. Offizielle Kernquellen (primär)

Versionen und Release-Hinweise:

- MongoDB Documentation Hub
- MongoDB Server Release Notes

Grundlagen und Architektur:

- MongoDB Manual (Current)
- Replica Set Concepts
- Sharding
- Shard Keys

Datenmodellierung:

- Data Modeling
- Embedded vs Referenced Data
- Schema Validation
- Time Series Collections

Query und Aggregation:

- CRUD Operations
- Aggregation Pipeline
- Indexes
- Explain Results

Konsistenz und Transaktionen:

- Write Concern
- Read Concern
- Read Preference
- Transactions
- Retryable Writes

Betrieb und Security:

- Backup Methods
- Database Profiler
- Security Checklist
- Auditing
- Encryption at Rest

9.2. Uebungsaufgaben (steigende Schwierigkeit)

1. **Modellierungsbasis**

- Entwirf `orders`, `payment_events`, `shipment_events` query-first.
- Begruende Embedding vs Referencing je Collection.

2. **Indexdesign**

- Leite für 6 API-Queries ein initiales Indexset ab.
- Prüfe jede Query mit Explain und dokumentiere die Ergebnisse.

3. **Konsistenzprofile**

- Definiere pro Endpunkt `writeConcern`, `readConcern`, `readPreference`.
- Simuliere einen Primary-Failover und validiere API-Verhalten.

4. **Transaktionsfall**

- Implementiere einen echten Multi-Document-Use-Case mit Transaktion.
- Vergleiche Latenz und Fehlerbehandlung gegen nicht-transaktionalen Entwurf.

5. **Operations-Drill**

- Erzeuge gezielt Slow Query und Replication-Lag.
- Führe Runbook-Schritte durch und dokumentiere Root Cause plus Fix.

6. **Sharding-Szenario**

- Entwerfe zwei alternative Shard Keys für Event-Daten.
- Simuliere Last und bewerte Hotspot- und Scatter-Gather-Risiko.

7. **Security-Hardening**

- Setze Rollenmodell mit Least Privilege um.
- Aktiviere TLS und Audit Logging für kritische Pfade.

9.3. 90-Tage-Fahrplan für Teams

Tage 1-30

- Query-Vertragskatalog aufsetzen.
- Datenmodell und Schema Validation definieren.
- Basis-Monitoring inklusive Slow Query und Replication-Lag aufbauen.

Tage 31-60

- Konsistenzprofile pro Endpunkt verbindlich machen.
- Index-Lifecycle-Prozess etablieren.
- Backup/Restore-Test mit dokumentiertem Ergebnis ausführen.

Tage 61-90

- Performance-Gates in Release-Prozess integrieren.
- Security-Rezertifizierung (Rollen, TLS, Audit) durchfuehren.
- Sharding-Readiness oder Optimierungsplan datenbasiert entscheiden.

9.4. Self-Assessment

Bewerte mit 0 bis 2 Punkten je Aussage:

- Wir haben einen dokumentierten Query-Katalog als Architekturvertrag.
- Jede kritische Query ist über Explain und Indexdesign abgesichert.
- Konsistenzprofile sind pro Endpunkt festgelegt und getestet.
- Wir können Replication-Lag-Incidents reproduzierbar behandeln.
- Backup und Restore sind praktisch geuebt.
- Rollenmodell folgt nachweisbar Least Privilege.
- Datenlebenszyklus (TTL/Archiv) ist technisch durchgesetzt.

Auswertung:

- 0-5: hohes Risiko, wenig Produktionsreife.
- 6-10: solide Basis, aber inkonsistent.
- 11-14: belastbares Produktionsniveau.

9.5. Abschluss

MongoDB wird dann produktionsreif, wenn drei Dinge gleichzeitig stimmen:

- Datenmodell passt zu den echten Query-Pfaden.
- Konsistenz- und Replikationsverhalten ist bewusst festgelegt.
- Betrieb, Security und Kostenkontrolle sind als dauerhafter Prozess etabliert.

Dann ist MongoDB nicht nur "flexibel", sondern ein stabiles Fundament für skalierbare Anwendungssysteme.

9.6. 120-Tage-Verbesserungsplan

Phase 1 (0-30 Tage)

- Query- und Konsistenzvertraege finalisieren.
- Top-Runbooks erstellen und Team schulen.

Phase 2 (31-60 Tage)

- Performance-Gates und Explain-Reviews in CI/CD verankern.
- Index-Lifecycle-Prozess produktiv machen.

Phase 3 (61-90 Tage)

- Failure- und Security-Drills durchfuehren.
- Backup/Restore und Incident-Kommunikation verifizieren.

Phase 4 (91-120 Tage)

- Governance-Metriken auf Management- und Teamdashboard integrieren.
- technische Schulden aus Postmortems systematisch abbauen.

Über den Autor

Erhard Rainer berät kleine und mittlere Unternehmen in Österreich zu Business Intelligence, Machine Learning und KI-Agenten. Er hat 15 Jahre Erfahrung in Data Analytics, davon 10 Jahre in der Leitung von Teams und Departments in Konzernen, und ist Dozent für betriebswirtschaftliche und Data-Analytics-Fächer.

Sprechen Sie mit mir über Ihre Daten: Das Erstgespräch ist kostenlos.

https://www.thinkbigger.at/contact.html#anfrage

© 2026 Think BIGGER – Erhard Rainer. Dieses PDF gibt den Stand des Online-Artikels vom 06.07.2026 wieder; maßgeblich ist die Online-Fassung unter https://www.thinkbigger.at/wissen/mongodb/