Think BIGGER

DEEP DIVE FÜR FACHLEUTE

Apache Cassandra

Verteilte Wide-Column-Datenbank für schreibstarke Workloads

Autor	Erhard Rainer
Veröffentlicht	11.10.2016
Aktualisiert	06.07.2026
Umfang	ca. 35 Min. Lesezeit
Online	www.thinkbigger.at/wissen/apache-cassandra/

Inhalt

0. Überblick (für Eilige) 2
1. Einordnung: Wann Apache Cassandra die richtige Wahl ist 4
2. Architektur & Kernkonzepte 7
3. Datenmodellierung (query-first) in der Praxis 9
4. CQL - systematische Praxisreferenz 13
5. Konsistenz, Replikation, Verfügbarkeit 16
6. Betrieb, Monitoring & Troubleshooting 19
7. Performance, Skalierung & Kostenkontrolle 22
8. Sicherheit, Governance & Compliance 25
9. Weiterführende Links + Übungsaufgaben 28

0. Überblick (für Eilige)

Apache Cassandra ist eine verteilte, spaltenorientierte NoSQL-Datenbank für Workloads mit hoher Schreiblast, großer horizontaler Skalierung und klaren Verfügbarkeitsanforderungen über mehrere Knoten oder Rechenzentren. Diese Seite ist eine praxisorientierte Referenz für Architektur, Datenmodellierung, CQL, Betrieb, Performance und Sicherheit. Ziel ist nicht nur "was ist Cassandra", sondern "wie baue und betreibe ich Cassandra so, dass es in Produktion stabil bleibt".

> Hinweis zum Versionsstand: Diese Referenz ist fachlich auf Stand **4. März 2026**. Auf der offiziellen Apache-Downloadseite waren zu diesem Zeitpunkt u. a. die Linien `5.0.6`, `4.1.10` und `4.0.19` sichtbar. Versionssensitive Aussagen sind als `5.0.x` bzw. `4.1.x` markiert.

0.1. Inhaltsverzeichnis (kurz)

- 0. Überblick (für Eilige)
- 1. Einordnung: Wann Apache Cassandra die richtige Wahl ist
- 2. Architektur & Kernkonzepte
- 3. Datenmodellierung (query-first) in der Praxis
- 4. CQL - systematische Praxisreferenz
- 5. Konsistenz, Replikation, Verfügbarkeit
- 6. Betrieb, Monitoring & Troubleshooting
- 7. Performance, Skalierung & Kostenkontrolle
- 8. Sicherheit, Governance & Compliance
- 9. Weiterführende Links + Übungsaufgaben

0.2. Quick-Glossar

- **Partition Key**: Schlüsselanteil, der bestimmt, auf welchen Knoten eine Partition landet.
- **Clustering Columns**: Sortierlogik innerhalb einer Partition.
- **RF (Replication Factor)**: Anzahl Replikate pro Datenbereich.
- **CL (Consistency Level)**: Anzahl/Art der erforderlichen Antworten pro Lese-/Schreiboperation.
- **SSTable**: Unveränderliche Datei auf Platte, in die Memtables geflusht werden.
- **Memtable**: In-Memory-Struktur für aktuelle Schreibvorgänge.
- **CommitLog**: Append-only-Durability-Log für Schreiboperationen.
- **Tombstone**: Löschmarker, der erst später durch Compaction endgültig entfernt wird.
- **Repair**: Anti-Entropy-Synchronisation zwischen Replikaten.
- **Hinted Handoff**: Zwischenspeicherung verpasster Writes für vorübergehend ausgefallene Replikate.
- **LWT**: Lightweight Transaction auf Basis von Paxos für compare-and-set Semantik.
- **UCS**: Unified Compaction Strategy, in Cassandra 5.0.x die empfohlene Standardstrategie für viele Workloads.

0.3. Lernpfad: Vom Einstieg zur Produktion

1. Verstehe zuerst das Betriebsmodell: Partitionierung, Replikation, Konsistenz, Compaction.
2. Modelliere danach strikt query-first, nicht tabellen-first.
3. Baue CQL-Schemata und Testdaten, miss echte Abfragepfade und P99-Latenzen.

4. Definiere dann Runbooks: Repair-Zyklen, Alerting, Incident-Workflow, Capacity-Planung.
5. Sichere zuletzt das Setup ab: TLS, AuthN/AuthZ, Auditing, Guardrails.

0.4. Mini-Projekt (durchgehend): IoT-Telemetrie mit globalem Betrieb

Dieses Mini-Projekt begleitet alle Kapitel. **Szenario:**

- 50.000 Geräte senden Telemetrieevents.
- Lastspitzen: bis zu 150.000 writes/s.
- Zwei Rechenzentren: `eu-central` und `us-east`.
- Anforderungen:
- Hohe Verfügbarkeit bei Knotenausfall.
- Zeitreihenabfragen pro Gerät für definierte Zeitfenster.
- TTL-basierte Datenhaltung (z. B. 90 Tage Rohdaten).
- Klares Runbook für Repair, Compaction und Incident Response.

Wir verwenden dieses Szenario fortlaufend für:

- Datenmodellierung und Partition-Key-Design.
- CL-Strategien für Read/Write.
- Compaction-Wahl.
- Betriebs- und Troubleshooting-Entscheidungen.

0.5. Profi-Heuristiken

- Cassandra ist stark, wenn du Zugriffspfade **vorher** kennst.
- Vermeide "freie Ad-hoc-Abfragen" als primären Use Case.
- Große Partitionen sind der häufigste Stabilitätskiller.
- Reparaturprozesse sind kein Optional-Feature, sondern Pflicht.
- Compaction-Strategie ist kein Detail, sondern Kernentscheidung.
- LWT nur dort einsetzen, wo echte linearisierbare CAS-Semantik benötigt wird.
- `ALLOW FILTERING` ist ein Warnsignal für Modellierungsfehler.

0.6. So setzt du es um - Checkliste

- Definiere 3 bis 7 harte Query-Patterns vor der Schemaarbeit.
- Setze ein dokumentiertes Ziel für P95/P99-Latenz je Pattern.
- Lege RF und CL pro Use Case fest (inkl. Failure-Modell).
- Plane Repair-Frequenzen inklusive Nachweis gegen `gc_grace_seconds`.
- Definiere mindestens 10 Betriebsmetriken und Alarmgrenzen.

0.7. Mentales Modell für den Alltag

Wenn du Cassandra in Projekten sicher einsetzen willst, hilft ein klares mentales Modell, das technische und organisatorische Ebenen verbindet: Cassandra ist kein "SQL-Ersatz mit anderer Syntax", sondern ein verteiltes Speichersystem mit bewusstem Trade-off zwischen Modellierungsdisziplin, Verfügbarkeit und operativer Steuerung; genau deshalb sind Schema, Konsistenzprofile und Betriebsprozesse untrennbar. Viele Fehlentscheidungen entstehen, weil Teams nur eine dieser Ebenen optimieren: Entwicklung optimiert Query-Comfort, Operations optimiert Knotenstabilitaet, Architektur optimiert Zukunftssicherheit; wenn diese drei Ziele nicht in einem gemeinsamen Vertrag zusammenlaufen, wird das System

inkonsistent im Verhalten, auch wenn jeder Teil für sich logisch wirkt. In der Praxis bewaehrt sich eine einfache Leitfrage pro Entscheidung: "Welches konkrete Endpunktverhalten unter Last oder Teilausfall verbessert sich messbar?" Wenn die Antwort fehlt, ist die Entscheidung meist nur implizite Hoffnung. Für die tägliche Arbeit bedeutet das: jeder neue Query-Pfad braucht einen expliziten Platz im Datenmodell; jede neue Lastannahme braucht einen reproduzierbaren Test; jede neue Tabelle braucht ein Betriebsprofil, das Repair-, Compaction- und Alerting-Folgen sichtbar macht. Cassandra-Teams, die dieses Modell leben, diskutieren weniger über "Best Practice aus Blog X" und mehr über überprüfbare Hypothesen am eigenen Workload, und genau das ist der Kern von produktionsreifer Arbeit mit verteilten Datenbanken. Auch das Thema Geschwindigkeit wird dadurch realistischer: schnell ist nicht der Build, der heute gruen ist, sondern der Betrieb, der in sechs Monaten noch vorhersehbar bleibt, obwohl Traffic, Teamgroesse und Anforderungen steigen. Deshalb lohnt es sich, schon früh ein gemeinsames Vokabular festzulegen: Was ist ein kritischer Endpunkt? Was ist tolerierbare Staleness? Wann ist ein Partition-Budget gerissen? Wann gilt ein Repair als "zu spät"? Wann ist ein Alert nur Rauschen? Sobald diese Begriffe klar sind, sinkt der Interpretationsspielraum in Incidents drastisch. Das gleiche gilt für Verantwortungen: Wenn niemand Owner für Partition-Gesundheit, Repair-Frequenz oder CL-Konsistenz ist, wird jedes Problem "eigentlich Sache der anderen". Ein gutes Cassandra-Setup macht Ownership explizit und messbar. Technisch ist das Ergebnis unspektakulaer, aber wirkungsvoll: weniger Notfallqueries, weniger Ad-hoc-Parameteraenderungen, weniger "mystische" Latenzspruenge. Organisatorisch ist der Effekt noch größer: Teams lernen, Cassandra nicht als Black Box zu behandeln, sondern als System, das auf explizite Verträge reagiert. Dieses mentale Modell ist die Grundlage für alle folgenden Kapitel: Architekturentscheidungen werden als Auswirkungen auf Endpunkt-SLOs gelesen; Datenmodellierung wird als Steuerung von Lastverteilung verstanden; CQL wird als Vertragssprache für Zugriffspfade genutzt; Konsistenz wird fachlich begründet statt technisch dogmatisch gesetzt; Betrieb wird als kontinuierlicher Sicherheitsmechanismus etabliert; und Security/Governance wird nicht als "letzter Schritt", sondern als Bestandteil der Betriebsfaehigkeit betrachtet.

---

1. Einordnung: Wann Apache Cassandra die richtige Wahl ist

1.1. Problemklassen, in denen Cassandra typischerweise stark ist

Cassandra ist besonders sinnvoll, wenn mehrere Bedingungen gleichzeitig erfüllt sind:

- Sehr hohe Schreiblast über viele Knoten.
- Verteilung über mehrere Failure-Domains (Rack, DC, Region).
- Hohe Verfügbarkeitsanforderung auch bei Teilstörungen.
- Vorhersehbare Query-Pfade mit klaren Partition Keys.
- Bereitschaft zur bewussten Datenmodellierung (Denormalisierung statt Joins).

Typische Domänen:

- IoT/Telemetrie.
- Eventing/Aktivitätslogs.
- Session- und Profilzustände.
- Time-Series mit TTL.
- Globale, verteilte operative Stores.

1.2. Wann ein RDBMS oft besser passt

Ein relationales System passt meist besser, wenn du primär brauchst:

- Komplexe Joins über viele Entitäten.
- ACID-Transaktionen über mehrere Partitionen/Entitäten.
- Starke referenzielle Integrität per Foreign Keys.
- Ad-hoc-Analysen mit hoher Flexibilität. Cassandra kann diese Anforderungen teilweise technisch umschiffen, aber oft mit unnötig hoher Komplexität und Betriebskosten.

1.3. Cassandra vs. Document-DB (praktische Perspektive)

Cassandra-Vorteile:

- Sehr robuste horizontale Skalierung im Write-Pfad.
- Replizierungs- und Konsistenzmodell fein steuerbar pro Anfrage.
- Gute Kontrolle über Hot Paths bei sauberem Schlüsselmodell. **Document-DB-Vorteile (je Produkt):**
- Flexiblere Ad-hoc-Abfragen.
- Oft einfacherer Einstieg bei schemaloser Iteration.
- Häufig reichhaltige Aggregationsframeworks.

Entscheidend ist nicht NoSQL vs. SQL, sondern:

- Stabilität der Query-Muster.
- Größe/Verteilung der Daten.
- SLOs für Availability und Latency.

1.4. Harte No-Go-Fälle für Cassandra

- Fachmodell verlangt regelmäßig Multi-Entity-Transaktionen.
- Team erwartet SQL-ähnliche Join-Freiheit als Standard.
- Datenmodell ist noch komplett volatil und Query-Muster unklar.
- Es existiert kein Betriebsteam für Repair/Compaction/Monitoring.

> **Anti-Pattern:** "Wir nehmen Cassandra für alles, weil es skalierbar ist." Ergebnis in der Praxis: aufwendige Workarounds, instabile Query-Latenzen, hohe Betriebslast.

1.5. Entscheidungsmatrix (kompakt)

Kriterium	Cassandra passt	Cassandra passt eher nicht
Write Throughput	Sehr hoch, dauerhaft	Niedrig bis moderat
Verfügbarkeit	Multi-Node/Multi-DC kritisch	Single-Node genügt
Query-Muster	Klar definiert, stabil	Stark ad-hoc
Datenmodell	Denormalisierung akzeptiert	Join-/FK-zentriert
Betriebsteam	SRE/DBA mit NoSQL-Erfahrung	Kein Betriebsfokus

1.6. So setzt du es um - Checkliste

- Führe vor Architekturentscheidung 3 Real-Workload-Tests durch.

- Bewerte mindestens zwei Alternativen mit denselben Lastprofilen.
- Dokumentiere explizit, welche No-Go-Punkte ausgeschlossen sind.
- Definiere ein Exit-Kriterium: Wann wäre Cassandra die falsche Wahl?

1.7. Praxisvergleich: Cassandra vs. "wir optimieren später"

Der häufigste Einfuehrungsfehler ist nicht ein falscher Parameter, sondern ein falscher Projektmodus. Viele Teams starten mit einem generischen Event-Table-Design und sagen:

- Wir nehmen erstmal einen breiten Datensatz.
- Queries und CL definieren wir später.
- Monitoring machen wir nach dem Go-Live.

Bei Cassandra führt diese Reihenfolge fast immer zu technischen Schulden:

- Tabellen sind nicht query-first.
- Partition Keys passen nicht zum echten Traffic.
- CL/Timeout-Probleme werden erst unter Last sichtbar.
- Späteres Refactoring bedeutet oft Re-Modeling plus Re-Backfill.

Die bessere Reihenfolge ist:

1. Endpunkt-Katalog mit Muss-Queries definieren.
2. Lastprofil und SLO pro Query festlegen.
3. Erst dann Schema, CL und Betriebsmodell bauen.
4. Vor Produktion Lasttests mit realistischer Datenverteilung fahren. Das ist der Unterschied zwischen "Cassandra als Datenbank" und "Cassandra als verteiltes System".

1.8. Architekturentscheidungen sauber dokumentieren

Wenn Cassandra langfristig stabil laufen soll, braucht jede Kernentscheidung eine kurze, pruefbare Begründung. Bewaehrte ADR-Punkte (Architecture Decision Record):

- Warum Cassandra und nicht Alternative X?
- Welche Queries sind first-class und welche second-class?
- Welche RF/CL-Kombinationen gelten je Endpunkt?
- Welche Partition-Size- und Latenzbudgets wurden vereinbart?
- Welche Anti-Pattern sind organisatorisch verboten?

Ergaenze jede ADR um "Betriebsfolgen":

- Welche Metriken zeigen früh, dass die Entscheidung kippt?
- Was ist der erste Schritt im Incident?
- Welche Rollback- oder Migrationstrategie existiert? So wird Architektur nicht nur beschlossen, sondern dauerhaft betreibbar gemacht.

---

2. Architektur & Kernkonzepte

2.1. Ring, Token, Partitioner, VNodes

Cassandra verteilt Partitionen über einen Token-Ring. Der Partitioner (in der Praxis meist Murmur3Partitioner) hasht den Partition Key auf einen Tokenbereich. Der jeweilige Tokenbereich wird von Knoten als Replikate gehalten. Mit VNodes (`num_tokens > 1`) besitzt jeder Knoten mehrere Tokenbereiche. VNodes erleichtern Rebalancing und Expansion, erhöhen aber die Zahl der beteiligten Peers bei Streaming/Repair. Die Wahl von `num_tokens` ist ein Trade-off:

- Mehr Flexibilität beim Skalieren.
- Mehr interne Verteilungskomplexität und potenziell höhere Repair-/Streaming-Last.

2.2. Replikatplatzierung, Snitches und Topologie

Replica Placement hängt von Replikationsstrategie und Topologie-Metadaten ab. Für Produktion wird `NetworkTopologyStrategy` empfohlen. Damit definierst du RF je Datacenter. Snitches liefern Cassandra Informationen zu DC/Rack-Zugehörigkeiten. Sie beeinflussen:

- Replica Placement.
- Routing-Entscheidungen.
- Failure Isolation. Fehlerhafte Rack-/DC-Modellierung führt zu vermeidbaren Ausfällen bei partiellen Störungen.

2.3. Write Path (vereinfacht, aber operativ relevant)

Beim Schreiben passiert auf dem Zielknoten typischerweise:

1. Mutation in den CommitLog.
2. Mutation in Memtable.
3. Später Flush in SSTables. Das bedeutet:

- CommitLog sichert Durability.
- Memtable gibt schnellen In-Memory-Zugriff.
- SSTables sind immutable und werden später kompaktifiziert.

2.4. Read Path (vereinfacht, aber operativ relevant)

Beim Lesen kombiniert Cassandra:

- Memtable-Daten.
- Mehrere SSTables. Hilfsstrukturen wie Bloom Filter reduzieren unnötige SSTable-Zugriffe. Trotzdem steigen Read-Kosten bei vielen überlappenden SSTables. Genau deshalb sind Compaction-Strategie, Tombstone-Management und Partition-Größen operativ kritisch.

2.5. LSM-Modell und Konsequenzen

Cassandra basiert auf einem LSM-Ansatz:

- Schreibvorgänge sind günstig (append-orientiert).
- Lesepfade können komplexer werden (mehrere SSTables + Tombstones).

Kernfolgen für die Praxis:

- Du musst Compaction aktiv steuern.
- Du musst Tombstones diszipliniert behandeln.
- Du musst Partitionen kontrolliert dimensionieren.

2.6. Konsistenzmechanismen im Hintergrund

Eventual Consistency wird unterstützt durch:

- Hinted Handoff (best effort).
- Read Repair.
- (Full/Incremental) Repair. Wichtig: Hints und Read Repair ersetzen kein geplantes Repair.

2.7. Mini-Projekt: Architekturentscheidung

Für die IoT-Telemetrie aus Abschnitt 0:

- Zwei DCs: `eu-central`, `us-east`.
- Keyspace mit `NetworkTopologyStrategy` und RF je DC.
- Device-gebundene Partitionierung mit Zeit-Buckets.
- Repair als fester Betriebsprozess, nicht "nur bei Bedarf".

2.8. So setzt du es um - Checkliste

- Prüfe Snitch- und Rack/DC-Mapping vor Go-Live auf jedem Knoten.
- Verifiziere Token-/Ownership-Verteilung (`nodetool status`).
- Definiere `num_tokens` bewusst statt Default ungeprüft zu übernehmen.
- Dokumentiere Write-/Read-Pfad-Risiken je Tabelle.
- Lege vorab fest, wie du Compaction-Backlog erkennst.

2.9. Topologiewechsel: Node add, replace, decommission

In Produktionsclustern ist die Architektur nicht statisch. Typische Lebenszyklusaktionen:

- Neue Knoten hinzufuegen (Scale-out).
- Defekte Knoten ersetzen (replace).
- Knoten kontrolliert entfernen (decommission). Jede Aktion bewegt Daten im Hintergrund und erzeugt Last auf Netzwerk, Storage und Peers. Praxisregeln:
- Topologieaenderungen nie parallel zu grossen Deployments oder Backfills fahren.
- Vorab klare Schwellwerte für Netzwerk und Disk-I/O definieren.
- Nach jeder Aktion Tokenverteilung und Latenzprofile erneut prüfen.

Fehlerbild aus der Praxis:

- Scale-out wird als "neuer Knoten + fertig" verstanden.
- Danach steigen P99-Latenzen wegen intensivem Streaming plus laufendem Produktionsverkehr. Deshalb: Topologieaenderungen als Change mit Wartungsfenster behandeln.

2.10. Architektur-Risiken früh erkennen

Architekturbedingte Probleme sind oft schleichend:

- Leichte Latenzdrifts über Wochen.

- Mehr Pending Compactions ohne sofortigen Alarm.
- Zunehmende Ungleichverteilung einzelner Tabellen.

Frühe Signale:

- Token Ownership wird ungleichmaessig.
- Tablestats zeigen deutlich unterschiedliche SSTable-Anzahlen pro Node.
- Einzelne API-Endpunkte steigen in P99 ohne offensichtliche Fehler. Wenn diese Signale auftreten, ist die Ursache oft kein einzelner Bug, sondern ein Missverhaeltnis zwischen Architekturannahmen und realem Traffic.

2.11. Architektur-Check vor Produktion

Vor dem ersten produktiven Traffic sollte ein technischer Architektur-Check zwingend durchlaufen werden. Checkpunkte:

- Snitch-Konfiguration auf jedem Node identisch und korrekt.
- RF je DC dokumentiert und gegen Failure-Annahmen getestet.
- Tokenverteilung plausibel, kein extremer Ownership-Drift.
- CommitLog-, Data- und Backup-Pfade auf getrennten Storage-Konzepten geplant.
- JVM- und OS-Parameter nach reproduzierbarem Baseline-Test gesetzt.
- Monitoring und Alerting greifen auf Cluster- und API-Ebene. Ein wichtiger Punkt ist "Systemgrenze". Definiere vorab:
- Maximale Datenrate.
- Maximale Partitiongroesse.
- Maximale tolerierbare Latenz unter Peak. Ohne diese Grenzen bleibt der Betrieb reaktiv, weil niemand sagen kann, wann ein Zustand "normal", "kritisch" oder "außer Spezifikation" ist.

2.12. Architektur-Entscheidungen an SLO koppeln

Architektur wird belastbar, wenn sie direkt an Service Level Objectives gekoppelt ist. Beispiel:

- SLO: P99 für "aktueller Gerätestatus" < 120 ms.
- Architekturauswirkung: dedizierte Read-Tabelle statt Historientabelle.
- Betriebsauswirkung: Compaction-/Repair-Plan für diese Tabelle priorisieren. Wenn SLO und Architektur nicht verbunden sind, werden Entscheidungen zu Meinungen. Wenn sie verbunden sind, werden Entscheidungen messbar und damit verhandelbar.

---

3. Datenmodellierung (query-first) in der Praxis

3.1. Das wichtigste Prinzip: Modelliere vom Query rückwärts

In Cassandra ist die Frage nicht: "Wie normalisiere ich das Fachmodell optimal?" Sondern: "Welche Abfragen müssen garantiert schnell funktionieren, und wie muss dafür der Primärschlüssel aussehen?" Der Primärschlüssel definiert:

- Verteilung (Partition Key).
- Sortierung und Scan-Range innerhalb der Partition (Clustering).

3.2. Primärschlüssel sauber schneiden

Regelwerk:

- Partition Key so wählen, dass Last breit verteilt wird.
- Clustering Columns so wählen, dass gewünschte Sortierung/Range direkt möglich ist.
- Keine unendlichen Partitionen aufbauen.

Beispiele:

- Gut: `(device_id, day_bucket)` als Partition Key bei Telemetrie.
- Schlecht: nur `device_id`, wenn ein Gerät unbegrenzt viele Events produziert.

3.3. Zeitreihenmuster: Bucketing ist Pflicht

Bei Zeitreihen ist Bucketing oft nicht optional, sondern überlebenswichtig. Typische Buckets:

- pro Stunde.
- pro Tag.
- pro Woche (nur bei moderater Eventdichte).

Trade-off:

- Zu kleiner Bucket: mehr Partitionen, mehr Metadaten-/Abfrage-Overhead.
- Zu großer Bucket: Hotspots und große Partitionen.

3.4. Table-per-Query statt Universal-Tabelle

Du modellierst in Cassandra oft mehrere Tabellen für dieselben fachlichen Daten. Grund:

- Jede Tabelle ist auf einen klaren Query-Pfad optimiert.
- Denormalisierung reduziert Laufzeit-Joins. Das ist kein Designfehler, sondern Kernprinzip.

3.5. Tombstones verstehen und aktiv vermeiden

Tombstones entstehen u. a. durch:

- `DELETE`.
- `TTL`-Ablauf.
- Updates auf Collections (abhängig vom Muster).

Probleme bei Tombstone-Last:

- Höhere Read-Latenz.
- Mehr Compaction-Aufwand.
- Risiko von Warnungen/Timeouts bei Scans.

Strategien:

- Löschmuster vereinfachen.
- TTL bewusst einsetzen (nicht blind auf alle Tabellen).
- Query-Pfade so gestalten, dass sie nicht über riesige Tombstone-Bereiche laufen.

3.6. Anti-Pattern-Kasten: Drei Klassiker

> **Anti-Pattern 1: ALLOW FILTERING als Standardlösung**

> Wirkung: Cassandra liest viel mehr Daten als nötig replica-seitig ein.
>
> Symptom: gute Funktion im Test, instabile Latenz in Produktion. **Anti-Pattern 2: Falscher Partition Key**
>
> Wirkung: Hot Partition, ungleichmäßige Last, einzelne Knoten am Limit. **Anti-Pattern 3: Unbegrenzte Partitionen**
>
> Wirkung: steigende Latenz, höhere Memory-/Compaction-Last, schwerer Betrieb.

3.7. Mini-Projekt: Schema für Telemetrieabfrage

Zielabfrage:

- "Liefere alle Events eines Geräts zwischen `t1` und `t2` in Zeitreihenfolge."

Schemaidee:

- Partition Key: `(device_id, day_bucket)`.
- Clustering: `event_ts` (absteigend oder aufsteigend je Query).
- Zusätzliche Tabelle für "letzter Zustand pro Gerät" statt ad-hoc Aggregation.

3.8. "So setzt du es um" - Modellierungsworkflow

1. Sammle alle Muss-Abfragen inkl. erwarteter Last.
2. Definiere pro Abfrage einen geplanten Primärschlüssel.
3. Simuliere Datenvolumen pro Partition für 30/90/365 Tage.
4. Verifiziere Top-Partition-Größen mit Lasttests.
5. Leite daraus Compaction-Strategie und TTL-Politik ab.

3.9. So setzt du es um - Checkliste

- Jede Produktionstabelle hat genau dokumentierte Zielabfragen.
- Jede Tabelle hat ein dokumentiertes Partition-Size-Budget.
- Für jede Tabelle ist ein Tombstone-Risiko bewertet.
- Für jede Query ist klar, ob `ALLOW FILTERING` verboten ist.
- Modellierung ist mit realistischen Datenmengen getestet.

3.10. Modellierungsreview für Teams

Vor jedem produktiven Rollout sollte ein strukturierter Modellierungsreview stattfinden. Minimalfragen je Tabelle:

- Welche Abfrage ist der Primaarpfad?
- Welche CL ist pro Pfad geplant?
- Wie gross wird eine Partition nach 1, 7, 30, 90 Tagen?
- Welche Writes erzeugen Tombstones?
- Wie sieht ein Rollback aus, falls das Modell kippt?

Sinnvoll ist ein "Red Team"-Check:

- Ein anderer Engineer versucht bewusst, das Modell durch Lastspitzen und Randfaelle zu brechen.
- Ziel ist nicht "Schema ist schoen", sondern "Schema bleibt unter Fehlern stabil".

3.11. Query-Vertraege statt impliziter Annahmen

Eine starke Praxis ist der Query-Vertrag pro API:

- Erlaubte Filterfelder.
- Garantierte Sortierung.
- Erlaubtes Zeitfenster.
- Erwartete Antwortgroesse.
- Definierte Fallback-Strategie bei Last. Wenn diese Punkte fehlen, entstehen später "Notfall-Queries", die das Datenmodell umgehen. Typische Symptome:
- neue "kurzfristige" Sekundaerabfragen im Code.
- unkontrolliertes `ALLOW FILTERING`.
- steigende Last auf Tabellen, die dafür nie gedacht waren. Mit Query-Vertraegen bleibt Cassandra vorhersehbar, selbst wenn mehrere Teams parallel entwickeln.

3.12. Migration aus relationalen Modellen

Beim Wechsel von RDBMS zu Cassandra scheitern Teams oft an der Denkweise, nicht an Syntax. Typische Stolperfallen:

- Tabellen 1:1 aus dem relationalen Modell übernehmen.
- Join-Pfade durch `ALLOW FILTERING` oder breite Batches ersetzen.
- Denormalisierung als "Datenredundanzfehler" behandeln.

Pragmatischer Migrationspfad:

1. Fachobjekte in Query-Gruppen aufteilen.
2. Pro Query-Gruppe ein Cassandra-optimiertes Tabellenpaar modellieren.
3. Datenfluss für dual-write oder Backfill definieren.
4. API-seitig schrittweise auf neue Query-Vertraege umstellen.
5. Alte relationale Zugriffspfade erst abschalten, wenn Messwerte stabil sind. Mit diesem Vorgehen wird Migration ein kontrollierter Umbau statt Big-Bang-Risiko.

3.13. Datenqualität in einem denormalisierten Modell

Denormalisierung braucht klare Qualitaetsregeln. Empfehlungen:

- Definiere eine Source-of-Truth je fachlichem Attribut.
- Definiere Sync-Regeln für projektionierte Tabellen.
- Definiere Prüfjobs für Abweichungen zwischen Projektionen.

Wichtiger Punkt:

- "Doppelt gespeichert" ist nicht automatisch "inkonsistent".
- Inkonsistenz entsteht erst, wenn Synchronisationsregeln fehlen. Wenn du diese Regeln explizit machst, wird Denormalisierung betrieblich robust und auditierbar.

---

4. CQL - systematische Praxisreferenz

4.1. Keyspace und Replikation korrekt definieren

Für Produktion gilt in der Regel:

- `NetworkTopologyStrategy` verwenden.
- RF je DC explizit setzen.

Beispiel:

```
CREATE KEYSPACE iot
WITH replication = {
  'class': 'NetworkTopologyStrategy',
  'eu-central': 3,
  'us-east': 3
};
```

4.2. Tabellen-DDL (Primärschlüssel präzise)

```
CREATE TABLE iot.telemetry_by_device_day (
  device_id text,
  day_bucket date,
  event_ts timestamp,
  event_type text,
  temperature double,
  humidity double,
  payload text,
  PRIMARY KEY ((device_id, day_bucket), event_ts)
) WITH CLUSTERING ORDER BY (event_ts DESC);
```

Hinweise:

- Partition Key ist `(device_id, day_bucket)`.
- Clustering Key ist `event_ts`.
- Sortierung deckt den Hauptquery direkt ab.

4.3. DML-Praxis: Insert/Update/Delete/TTL

Insert:

```
INSERT INTO iot.telemetry_by_device_day (
  device_id, day_bucket, event_ts, event_type, temperature, humidity, payload
) VALUES (
  'dev-42', '2026-03-04', '2026-03-04T10:00:00Z', 'measurement', 21.7, 41.2, '{}'
) USING TTL 7776000;
```

`TTL` muss fachlich begründet sein. Ein globales "TTL für alles" erzeugt oft unnötige Tombstone-Last.

4.4. Batches: korrekt einsetzen

Batches in Cassandra sind **keine** 1:1-SQL-Transaktionen. Wichtig:

- Batches nur für zusammengehörige Mutationen nutzen.
- Große Multi-Partition-Logged-Batches können teuer sein.
- `UNLOGGED BATCH` nur gezielt und mit Verständnis einsetzen.

> **Anti-Pattern:** "Batch = Performance-Booster".

> In Cassandra ist Batch primär ein Korrektheits-/Atomicity-Werkzeug für bestimmte Fälle, kein pauschaler Throughput-Turbo.

4.5. LWT / Paxos: nur wo wirklich nötig

CQL-Conditional Statements (`IF`, `IF NOT EXISTS`) verwenden Paxos-Phasen. Das liefert linearisierbare CAS-Semantik auf Single-Partition-Ebene. Dafür zahlst du mit höherer Latenz und geringerem Durchsatz. Beispiel:

```
INSERT INTO iot.device_registry (device_id, owner, created_at)
VALUES ('dev-42', 'team-a', toTimestamp(now()))
IF NOT EXISTS;
```

`SERIAL` und `LOCAL_SERIAL` betreffen die Serial-Phase.

4.6. Indizes: 2i, SAI, Grenzen

Für Cassandra 5.0.x ist SAI ein zentraler Indexpfad. Wichtige Punkte:

- SAI unterstützt zusätzliche Filtermöglichkeiten.
- SAI ist nicht Freifahrtschein für schlechte Primärschlüssel.
- Für Cassandra laut SAI-FAQ ist SAI an 5.0 gekoppelt.

Empfehlung:

- Primärmodell zuerst sauber.
- Index nur dort, wo Query-Pfad sonst nicht sinnvoll modellierbar ist.

4.7. Materialized Views: bewusst konservativ

Materialized Views sind in der Architektur-Übersicht als experimentell markiert. Wenn du sie einsetzt:

- Nur bei klar begründetem Mehrwert.
- Einschränkungen und Limitierungen explizit kennen.
- Betriebs- und Rebuild-Strategie vorab definieren.

4.8. `ALLOW FILTERING`: im Zweifel Modell korrigieren

`ALLOW FILTERING` kann im Einzelfall legitim sein. Als Standardmuster ist es meist ein Symptom, dass Partitionierung/Primärschlüssel nicht zum Query passen. In produktiven Clustern sollte der Einsatz klar begrenzt oder per Guardrails kontrolliert sein.

4.9. Mini-Projekt: CQL-Bausteine

Im Mini-Projekt setzen wir typischerweise ein:

- Eine Write-optimierte Telemetrie-Tabelle pro Gerät + Bucket.
- Eine Query-Tabelle für "letzter bekannter Zustand".
- Optional SAI für eng umrissene Zusatzfilter in 5.0.x.
- LWT nur für Registry-/Idempotenzfälle.

4.10. So setzt du es um - Checkliste

- Jede CQL-Tabelle hat eine dokumentierte Primärabfrage.

- Kein Batch ohne dokumentierten Zweck.
- LWT nur auf klar benannten CAS-Pfaden.
- Indexeinsatz nur nach Lasttest + Selektivitätsprüfung.
- `ALLOW FILTERING` nur mit expliziter Ausnahmefreigabe.

4.11. CQL-Pattern: Upsert und Idempotenz

Viele Event- und Telemetriepfade brauchen idempotentes Schreiben. Bewaehrte Muster:

- Eindeutiger Event-Key in der Primary-Key-Struktur.
- Write-Pfade so bauen, dass wiederholte Zustellung denselben Endzustand erzeugt.
- LWT nur für echte Konfliktfaelle, nicht für den gesamten Ingest.

Pragmatische Regel:

- Erst idempotent modellieren.
- Erst danach prüfen, ob LWT überhaupt erforderlich ist.

4.12. CQL-Pattern: Query-by-state ohne Vollscan

Anforderung:

- "Zeig alle Geräte im Status `offline`."

Fehlerhafter Reflex:

- Primärtable scannen und filtern.

Besser:

- Dedizierte Tabelle für den Zugriffspfad.
- Zustandsaenderungen in beide Tabellen schreiben (denormalisiert).
- Klare Verantwortlichkeit für Konsistenz der Projektion. Das kostet Schreibaufwand, reduziert aber drastisch Laufzeitrisiko.

4.13. CQL-Pattern: Historie und aktueller Zustand trennen

Kombiniere in produktiven Modellen oft zwei Tabellen:

- Historische Event-Tabelle (zeitlich geordnet, TTL möglich).
- Aktueller Zustand pro Entity (klein, read-optimiert).

Vorteile:

- Historienabfragen belasten nicht den aktuellen Zustands-Endpunkt.
- API für "aktueller Zustand" bleibt stabil und schnell.
- Betriebsmetriken werden klarer interpretierbar.

4.14. CQL-Review: Release-Checkliste für Teams

Vor jedem Schema-Release:

- Sind neue Tabellen mit Lasttest belegt?
- Ist jede neue Query ohne `ALLOW FILTERING` dokumentiert?
- Gibt es Grenzwerte für Antwortgroessen?

- Sind CL-Annahmen für den Endpunkt explizit?
- Gibt es ein Degradationsverhalten für Lastspitzen? Ein CQL-Release ohne diese Antworten ist in der Regel nur "funktional", aber nicht "betriebsreif".

---

5. Konsistenz, Replikation, Verfügbarkeit

5.1. Tunable Consistency praktisch nutzen

Cassandra erlaubt CL pro Operation. Gängige CLs:

- `ONE`, `LOCAL_ONE`.
- `QUORUM`, `LOCAL_QUORUM`.
- `ALL`.
- `ANY` (nur Write).

Faustregel für starke Sichtbarkeit:

- Wähle Read-/Write-CL so, dass Quoren überlappen (`W + R > RF`).

5.2. Multi-DC: LOCAL_QUORUM als Standardmuster

In Multi-DC-Setups ist `LOCAL_QUORUM` oft ein guter Default:

- Gute Balance aus Konsistenz und Latenz.
- Lokale Antwortpfade ohne Fern-DC als Pflichtteil. `EACH_QUORUM` ist strenger, aber latenz- und ausfallsensitiver.

5.3. Replikationsfaktor je Datacenter

RF muss pro DC bewusst festgelegt werden. Muster:

- RF=3 je produktivem DC ist häufig ein Ausgangspunkt.
- Abweichungen müssen fachlich und betrieblich begründet sein.

Wichtig:

- RF, CL und Failure-Model zusammen betrachten.
- Kein isoliertes Tuning einzelner Parameter.

5.4. Hinted Handoff realistisch einordnen

Hints reduzieren Inkonsistenzfenster. Aber:

- Hints sind best effort.
- Hints sind kein Ersatz für Repair.
- Parameter wie `max_hint_window` begrenzen das Verhalten.

5.5. Repair-Strategie: Pflichtprozess

Repair gleicht verpasste Writes und Replikadifferenzen aus. Ohne Repair steigt Risiko für:

- persistente Divergenzen.
- Datenverlustszenarien bei Folgefehlern.

- Reappearing data nach Tombstone-Ablauf.

Bewährte Praxis:

- Regelmäßige inkrementelle Repairs.
- Zusätzlich periodische Full Repairs.
- Frequenz so wählen, dass `gc_grace_seconds` sicher eingehalten wird.

5.6. Read Repair und Konsistenzwirkung

Read Repair hilft, Divergenzen bei Lesevorgängen zu korrigieren. Es ersetzt aber keinen systematischen Repair-Lauf.

5.7. CL-Profile für typische Pfade

Beispielprofil Telemetrie:

- Write: `LOCAL_QUORUM` für produktive Messdaten.
- Read (aktueller Zustand): `LOCAL_QUORUM`.
- Read (weniger kritisch, hohe Last): `LOCAL_ONE` für sekundäre Views. Du solltest CL pro Endpunkt dokumentieren, nicht ad hoc im Code verteilen.

5.8. Mini-Projekt: Ausfall-Szenario

Szenario:

- Ein Node in `eu-central` fällt 90 Minuten aus.

Erwartung bei sauberer Konfiguration:

- Writes mit `LOCAL_QUORUM` bleiben verfügbar.
- Hints puffern einen Teil der verpassten Updates.
- Repair synchronisiert verbleibende Lücken.

5.9. So setzt du es um - Checkliste

- Definiere RF je DC in einem Architekturvertrag.
- Definiere CL je API-Endpunkt (Read/Write getrennt).
- Simuliere Node- und Rack-Ausfall vor Go-Live.
- Lege Repair-Kalender inkl. Verantwortlichen fest.
- Prüfe regelmäßig, ob Repair-Frequenz zu `gc_grace_seconds` passt.

5.10. Konsistenz-Entscheidungen als API-Vertrag

CL darf kein verstecktes Treiberdetail sein. Praktische Regel:

- Jede API beschreibt ihre Konsistenz explizit.
- Jede API beschreibt ihr Verhalten bei Teilstoerungen.
- Jede API beschreibt, welche Daten eventuell stale sein dürfen.

Beispiel:

- "Dashboard-Kachel: LOCAL_ONE erlaubt, stale bis 10 Sekunden."

- "Abrechnungsstatus: LOCAL_QUORUM Pflicht, keine stale Daten toleriert." Damit werden Konsistenzkosten transparent und fachlich gesteuert.

5.11. Failure-Drills für Konsistenz und Verfügbarkeit

Plane regelmäßige Drills:

1. Ein Node fällt aus.
2. Ein Rack ist teilweise nicht erreichbar.
3. Inter-DC-Latenz steigt deutlich. Messe dabei:

- Erfolgsquote pro Endpunkt.
- Latenzveraenderung pro CL-Profil.
- Zeit bis zur Normalisierung nach Wiederherstellung. Nur so siehst du, ob theoretische RF/CL-Entscheidungen im echten Betrieb tragen.

5.12. Repair-Governance im Team

Repair scheitert selten technisch, aber oft organisatorisch. Ursachen:

- Keine klare Verantwortung.
- Keine feste Frequenz.
- Kein Reporting über "wann zuletzt repariert".

Minimum für Governance:

- Repair-Kalender mit Owner.
- Automatische Nachweise je Keyspace/Tabelle.
- Eskalation, wenn Schwellwerte ueberschritten werden. Repair ist ein kontinuierlicher Prozess, kein Ticket für den Notfall.

5.13. CL-Entscheidungshilfe (praxisnah)

Use Case	Schreib-CL	Lese-CL	Typisches Ziel
Kritischer Status	`LOCAL_QUORUM`	`LOCAL_QUORUM`	geringe stale-Wahrscheinlichkeit
Operative Suche	`LOCAL_QUORUM`	`LOCAL_ONE`	niedrige Latenz, stale tolerierbar
Telemetrie-Rohdaten	`LOCAL_ONE` oder `LOCAL_QUORUM`	`LOCAL_ONE`	hoher Durchsatz
Konfliktsensitive Registrierung	LWT + `SERIAL` / `LOCAL_SERIAL`	`LOCAL_QUORUM`	eindeutiger Zustand
Diese Tabelle ist keine universelle Wahrheit, sondern ein Startpunkt.			
Entscheidend bleibt:			

- Welche Daten darf kurzfristig stale sein?
- Welche API darf bei Teilstoerung verlangsamen statt inkonsistent zu werden?
- Welcher Endpunkt ist business-kritisch?

5.14. Konsistenzkosten transparent machen

Jeder Konsistenzgewinn hat Kosten. Kostenarten:

- Höhere Latenz.
- Geringerer Durchsatz.
- Höhere Abhängigkeit von mehreren Replikaten.

Die richtige Frage lautet nicht "welche CL ist technisch möglich", sondern:

- "Welche CL ist fachlich notwendig und wirtschaftlich vertretbar?" Wenn diese Frage pro Endpunkt dokumentiert ist, sinkt die Zahl ungeplanter Konsistenzdiskussionen im Incident drastisch.

---

6. Betrieb, Monitoring & Troubleshooting

6.1. Betriebsgrundsatz: Cassandra braucht aktives Operieren

Cassandra ist kein "install and forget"-System. Du brauchst:

- reproduzierbares Monitoring.
- definierte Eingriffswege.
- klare Runbooks für häufige Incidents.

6.2. Nodetool-Runbook (Baseline)

Wichtige Kommandos im Tagesbetrieb:

- `nodetool status` (Clusterzustand, Up/Down, Token-Ownership).
- `nodetool tablestats` (Table-Metriken inkl. SSTables, Read/Write-Latenz).
- `nodetool tpstats` (Threadpool-Druckpunkte).
- `nodetool compactionstats` (Compaction-Backlog).
- `nodetool repair -pr` (geplanter Repair pro Node/Range).
- `nodetool toppartitions` (aktive/heiße Partitionen erkennen).

6.3. Kernmetriken und Warnsignale

Mindestmetriken pro Cluster:

- Read/Write Latency (P95/P99).
- Timeout- und Unavailable-Raten.
- Pending Compactions.
- SSTable Count je Tabelle.
- Tombstone-Pressure-Indikatoren.
- Repair-Drift (wann zuletzt je Node/Tabelle repariert).
- Disk-Auslastung und Wachstum.
- JVM Heap Pressure und GC-Pausen.

6.4. Typische Fehlerbilder

A) P99-Latenz steigt schleichend Häufige Ursachen:

- Compaction-Backlog.
- Zu viele SSTables je Read.
- Tombstone-lastige Query-Pfade.
- Übergroße Partitionen. **B) Lokale Timeouts bei Spitzenlast** Häufige Ursachen:
- Hot Partitions.
- ungeeignete CL für Last-/Failureprofil.
- Threadpool-Engpässe. **C) Dateninkonsistenzen nach längeren Ausfällen** Häufige Ursachen:
- Repair-Intervalle zu lang.
- falsches Vertrauen auf Hints als alleinigen Mechanismus.

6.5. Reproduzierbarer Debugging-Workflow

1. Incident-Typ klassifizieren (Latency, Fehlerquote, Konsistenz, Kapazität).
2. Clusterzustand prüfen (`status` , `tpstats` , `compactionstats`).
3. Betroffene Tabellen identifizieren (`tablestats` , Query-Logs).
4. Partition-/Modellierungsprobleme gegenprüfen (`toppartitions` , Query-Pattern).
5. Kurzfristige Stabilisierung (Load reduzieren, CL/Traffic steuern, Compaction/Repair planen).
6. Nachhaltige Korrektur im Modell/Betrieb umsetzen.

6.6. Backups, Snapshots, Restore

Für Produktionssicherheit:

- Regelmäßige Snapshots.
- Test-Wiederherstellungen in isolierter Umgebung.
- Dokumentiertes RTO/RPO je Datenklasse. Ein Backup ohne validierten Restore-Prozess ist operativ unzureichend.

6.7. Mini-Projekt: Incident "Hot Device"

Szenario:

- Ein Gerät produziert extrem viele Events.
- Eine Partition wächst außer Plan.
- P99-Latenz steigt nur bei bestimmten API-Endpunkten.

Gegenmaßnahmen:

- Bucketing feiner schneiden.
- Query-Pfade und Clustering prüfen.
- ggf. Denylisting für akute Stabilisierung erwägen.
- langfristig Modellkorrektur ausrollen.

6.8. So setzt du es um - Checkliste

- Erstelle ein 1-Seiten-Runbook je Kerntabelle.
- Definiere Alarmgrenzen mit On-Call-Aktion (nicht nur Schwellenwert).
- Teste vierteljährlich Restore + Repair-Runbook.
- Simuliere mindestens 2 realistische Failure-Szenarien pro Halbjahr.
- Miss nach jedem Eingriff denselben KPI-Satz für Vergleichbarkeit.

6.9. Alerting-Design: Signal statt Alarmflut

Ein gutes Cassandra-Alerting trennt:

- Fruehwarnung (Trend kippt).
- akute Stoerung (SLO verletzt).
- kritische Eskalation (Daten- oder Verfuegbarkeitsrisiko).

Schlechtes Alerting:

- Hunderte technische Einzelalarme ohne Priorisierung.
- keine Handlungsempfehlung im Alarmtext.
- keine Verknüpfung mit API-SLOs.

Gutes Alerting:

- wenige, klare Alarme.
- pro Alarm: Ursache, Dringlichkeit, nächster Schritt.
- Verknüpfung zu Dashboards und Runbook.

6.10. Incident-Kommunikation für verteilte Datenbanken

Bei Cassandra-Incidents geht es nicht nur um Technik, sondern um Erwartungsmanagement. Sinnvolle Kommunikationsstruktur:

1. Welche Endpunkte sind betroffen?
2. Ist es ein Verfuegbarkeits- oder Konsistenzproblem?
3. Welches Degradationsverhalten greift?
4. Wann kommt das nächste Update? So vermeidest du, dass Fachteams "Timeout" und "falsche Daten" durcheinander interpretieren.

6.11. Betriebsmetriken als Release-Gate

Ein unterschätzter Ansatz:

- Neue Releases werden nur freigegeben, wenn ein definierter Cassandra-Metriksatz im gruenen Bereich bleibt.

Typische Gate-Metriken:

- P99 je Endpunkt.
- Timeout-Rate.
- Pending Compactions.
- maximale Partitiongroesse.
- Repair-Status. Damit wird Datenbankstabilitaet Teil des Software-Lebenszyklus statt nachgelagerter Ops-Aufgabe.

6.12. Runbook-Vorlage pro Kerntabelle

Eine einfache, aber wirksame Vorlage:

- Tabellenname und Business-Zweck.
- Primärqueries und erwartete Last.
- CL-Profil je Endpunkt.

- Normalbereich für P95/P99 und Timeout-Rate.
- Typische Fehlerbilder.
- Erste 5 Diagnosebefehle.
- Eskalationskette. Diese Vorlage reduziert On-Call-Risiko deutlich, weil auch neue Teammitglieder reproduzierbar reagieren können.

6.13. Post-Incident-Lernen in Cassandra-Teams

Nach jedem Incident sollte es eine kurze, technische Nachbereitung geben:

1. Was war der erste messbare Indikator?
2. Welche Diagnose war hilfreich, welche Zeitverlust?
3. Welche Architekturannahme war falsch oder unvollständig?
4. Welche Guardrail oder Metrik hätte früher geholfen?
5. Welche Runbook-Aenderung folgt konkret? Wichtig:

- Fokus auf Systemverbesserung, nicht auf Schuldzuweisung. So wird jeder Incident zu einer Verbesserung der Datenplattform statt zu einem isolierten Feuerwehreinsatz.

---

7. Performance, Skalierung & Kostenkontrolle

7.1. Compaction-Strategien: STCS, LCS, TWCS, UCS

Compaction ist ein zentraler Performancehebel. Klassische Strategien:

- **STCS**: robustes Default-/Fallback-Muster.
- **LCS**: oft gut bei read-heavy oder vielen Updates/Deletes.
- **TWCS**: typischerweise für zeitbasierte, TTL-lastige Workloads.

Cassandra 5.0.x:

- **UCS** ist als empfohlene Strategie für viele Workloads hervorgehoben. Die Wahl darf nie pauschal erfolgen. Sie muss zum Datenalterungsprofil, Query-Muster und Write/Read-Verhältnis passen.

7.2. Tombstone- und TTL-Kosten beherrschen

TTL und Deletes sind fachlich oft notwendig. Operativ entstehen dadurch Kosten:

- zusätzliche Read-Arbeit.
- spätere Compaction-Last.

Wenn TTL-Workloads dominant sind:

- Datenfenster konsequent modellieren.
- passende Compaction wählen (häufig TWCS/UCS-Variante).
- Repair-Frequenz mit `gc_grace_seconds` abstimmen.

7.3. Caches und Read-Path-Tuning

Wichtige Stellhebel:

- Bloom Filter (`bloom_filter_fp_chance`).

- Index Summary / Key Cache / Row Cache je Workload.
- SSTable-Anzahl und Überlappung.

Merksatz:

- Häufig ist gutes Datenmodell + passende Compaction wirksamer als aggressives Cache-Tuning.

7.4. JVM/GC und Speicher

JVM ist Teil der Datenbank-Performance. Betriebsregeln:

- Heap-Größen konservativ und messbasiert einstellen.
- GC-Pausen und Throughput gemeinsam betrachten.
- Off-Heap-Nutzung und OS-Cache nicht ignorieren. Bei 5.0.x-Features wie SAI kann Direct-Memory-Planung relevant werden.

7.5. Kapazitätsheuristiken (pragmatisch)

Plane nicht nur Datenvolumen, sondern auch Betriebsverkehr:

- Writes/s pro Tabelle.
- Reads/s pro Query-Pfad.
- Replikations-/Repair-/Compaction-Traffic.
- Spitzenprofile (Tageszeiten, Release-Fenster, Backfills).

Sinnvolle Anfangsheuristik:

- Kapazität für Peak + Sicherheitsreserve auslegen.
- Danach unter Realverkehr nachjustieren.

7.6. Skalierungsstrategie

Scale-out ist in Cassandra der Normalfall. Aber:

- Jeder neue Node erzeugt Betriebsarbeit (Streaming, Rebalancing, Monitoringanpassung).
- Große, seltene Skalierungssprünge sind riskanter als planvolle inkrementelle Erweiterung.

7.7. Kostenkontrolle ohne Qualitätsverlust

- Tabellen mit geringer Business-Relevanz auf günstigere CL profile setzen.
- TTL-Politik differenziert je Datenklasse definieren.
- Unnötige Replikation oder überzogene RF vermeiden.
- Compaction/Repair-Zeitfenster auf Infrastrukturkosten abstimmen.

7.8. Mini-Projekt: Performance-Tuning-Loop

1. Ausgangsmessung mit Realprofil (Write-heavy + Read-spikes).
2. Partition-Größen und Top-Hotspots prüfen.
3. Compaction-Strategie validieren.
4. CL-Profile je Endpunkt feintunen.
5. Erneut messen (P95/P99 + Timeout + Ressourcendruck).

7.9. So setzt du es um - Checkliste

- Pro Tabelle ist eine Compaction-Strategie dokumentiert und begründet.
- Tombstone- und TTL-Profile sind messbar gemacht.
- P95/P99-Ziele sind pro Endpunkt festgelegt.
- Skalierungs-Playbook für Node-Add/Replace ist getestet.
- Monatlicher Capacity-Review ist institutionalisiert.

7.10. Benchmarking, das wirklich Aussagen liefert

Viele Cassandra-Benchmarks sind zu synthetisch. Ein belastbarer Benchmark bildet mindestens ab:

- realistische Partitionsverteilung.
- realistische Mischung aus Write, Read, TTL, Delete.
- reale CL-Konfiguration.
- Lastspitzen statt nur konstantem Throughput. Wichtig ist nicht nur der Mittelwert. Entscheidend sind:
- P95/P99-Latenz.
- Timeout-Verhalten unter Peak.
- Erholung nach Lastwechsel.

7.11. Compaction als Produktentscheidung

Compaction beeinflusst direkt:

- API-Latenz.
- Kosten.
- Stabilität in Lastspitzen. Deshalb sollte Compaction nicht nur ein DBA-Detail sein, sondern mit Produktanforderungen abgeglichen werden:
- Wie alt dürfen Daten sein?
- Welche Endpunkte sind geschäftskritisch?
- Wo ist Latenz wichtiger als Storage-Effizienz? Wenn diese Fragen geklärt sind, wird die Wahl von STCS/LCS/TWCS/UCS nachvollziehbar statt dogmatisch.

7.12. Kostenkontrolle durch Datenklassifizierung

Ein praxisnahes Modell:

- **Tier A**: geschäftskritische Daten, strenge CL, lange Aufbewahrung.
- **Tier B**: operative Daten, mittlere CL, mittlere Aufbewahrung.
- **Tier C**: kurzlebige Telemetrie, guenstigere CL, kurze TTL. Diese Klassifizierung reduziert Kosten oft deutlich, ohne kritische Pfade zu gefaehrden.

7.13. Kapazitaetsrechnung für das Mini-Projekt

Ein grober, aber nützlicher Ansatz:

- 150.000 writes/s Peak.
- durchschnittliche Eventgroesse inklusive Overhead annehmen.
- RF je DC berücksichtigen.
- Compaction- und Repair-Traffic als Reserve einplanen.

Dann nicht nur "Netto-Datenmenge" rechnen, sondern:

- commit log throughput.
- Disk-Write-Pressure inklusive Compaction.
- Headroom für Rebalancing/Streaming. Damit wird Capacity-Planung realistisch statt zu optimistisch.

7.14. Performance-Regressionen früh stoppen

Führe einen festen Performance-Regressionstest pro Release ein:

- gleiche Lastprofile.
- gleiche CL-Profile.
- gleiche Kernmetriken.

Abbruchkriterien:

- P99 verschlechtert sich > definierter Schwellwert.
- Timeout-Rate steigt signifikant.
- Pending Compactions verlassen den tolerierten Bereich. So werden Performanceprobleme im Releaseprozess erkannt und nicht erst im On-Call.

---

8. Sicherheit, Governance & Compliance

8.1. Sicherheitsgrundsatz

Eine Default-Installation ist funktional, aber sicherheitstechnisch nicht produktionsreif. Du brauchst mindestens:

- Verschlüsselung in Transit.
- Client-Authentifizierung.
- Autorisierung über Rollen/Rechte.

8.2. AuthN/AuthZ mit Rollenmodell

Cassandra-Rollen können Benutzer und Gruppen repräsentieren. Operative Mindestregeln:

- Keine Shared-Admin-Accounts für Anwendungen.
- Technische Rollen je Service.
- Least Privilege auf Keyspace-/Tabellenebene.

Beispielmuster:

- `iot_writer_role` mit `MODIFY` auf Write-Tabellen.
- `iot_reader_role` mit `SELECT` auf Read-Tabellen.
- `iot_ops_role` für Betriebsrechte (restriktiv).

8.3. TLS: Client-zu-Node und Internode

Cassandra trennt TLS-Konfiguration für:

- `client_encryption_options`.
- `server_encryption_options`.

Praxisregeln:

- Internode-Verschlüsselung in produktiven Netzen aktivieren.
- mTLS prüfen, wenn Sicherheitsanforderung es verlangt.
- Zertifikatsrotation inklusive Reload-Prozess operationalisieren.

8.4. Auditing

Audit Logging unterstützt Compliance und Forensik. Es erfasst u. a.:

- CQL-Aktivität.
- Authentifizierungs-/Autorisierungsereignisse.

Wichtig:

- Umfang (Kategorien, Keyspaces, Nutzer) gezielt konfigurieren.
- Log-Aufbewahrung und Zugriff rechtlich sauber regeln.

8.5. Dynamic Data Masking (5.0.x)

Für sensible Daten bietet 5.0.x dynamische Maskierung. Nutzen:

- Reduzierte Exposition im Query-Ergebnis.
- Rollentrennung zwischen Masked/Unmasked Zugriff.

Grenze:

- Maskierung ersetzt keine Verschlüsselung und kein Rechtekonzept.

8.6. Guardrails als Governance-Werkzeug

Guardrails helfen, riskante Nutzungsmuster zu begrenzen. Beispiele:

- `allow_filtering_enabled`.
- Partition-/Tombstone-bezogene Schwellen.
- Einschränkungen bestimmter Konsistenzlevel. Damit lassen sich bekannte Anti-Pattern technisch bremsen, bevor sie den Cluster destabilisieren.

8.7. Compliance-Basis

Für regulierte Umgebungen brauchst du:

- Rechte- und Rollennachweise.
- Auditierbare Zugriffspfade.
- dokumentierte Key-/Zertifikatsprozesse.
- regelmäßige Rezertifizierung von Rollen.

8.8. So setzt du es um - Checkliste

- TLS für Client- und Internode-Verkehr ist aktiv und getestet.
- Rollenmodell folgt Least Privilege und ist dokumentiert.
- Audit Logging ist für relevante Kategorien aktiv.
- Guardrails verhindern definierte Anti-Pattern.
- Security-Runbook deckt Zertifikatsrotation und Incident Response ab.

8.9. Rollenrezertifizierung und Drift-Kontrolle

In vielen Umgebungen ist nicht die Erstkonfiguration das Problem, sondern Rollen-Drift über Monate. Bewaehrte Praxis:

- Quartalsweise Rezertifizierung aller Rollen.
- Technische Nutzer gegen aktuelle Service-Landschaft prüfen.
- Ueberprivilegierte Rollen systematisch abbauen.

Ziel:

- Rechte nur dort, wo sie heute noch benötigt werden.

8.10. Datenmaskierung und Beobachtbarkeit ausbalancieren

Maskierung reduziert Risiko, kann aber Observability erschweren. Pragmatischer Mittelweg:

- Betriebsrelevante Felder unmaskiert, fachlich sensible Felder maskiert.
- Klarer Prozess, wer im Incident temporär erweiterten Zugriff bekommt.
- Zugriffserweiterungen immer auditiert und zeitlich begrenzt. So bleiben sowohl Compliance als auch Incident-Faehigkeit erhalten.

8.11. Governance-Metriken für Security-Reviews

Sicherheit wird steuerbar, wenn du sie misst. Sinnvolle Kennzahlen:

- Anteil technischer Konten mit Least Privilege.
- Anzahl ausstehender Rollenrezertifizierungen.
- Audit-Logging-Abdeckung pro Keyspace.
- Anzahl geblockter Guardrail-Verletzungen pro Monat. Diese Metriken machen Security-Qualitaet sichtbar und führen weg von rein reaktiven Audits.

8.12. Security-Incident-Drill

Wie bei Verfuegbarkeitsdrills sollte es auch Security-Drills geben. Beispielszenarien:

- kompromittiertes technisches Konto.
- abgelaufenes Zertifikat.
- unautorisierter Query-Versuch auf sensible Tabelle.

Ziel im Drill:

- Erkennungsgeschwindigkeit messen.
- Reaktionskette testen.
- Auditierbarkeit und Nachvollziehbarkeit prüfen. So wird Security handlungsfaehig statt nur dokumentiert.

8.13. Least Privilege in der Praxis durchsetzen

Least Privilege scheitert oft an Bequemlichkeit. Wirksame Mechanismen:

- Rollen nur über IaC/Code-Review aenderbar machen.
- Ablaufdaten für Ausnahmerechte.
- Regelmäßige automatische Reports über ueberprivilegierte Konten. Damit wird Security nicht vom individuellen Disziplinniveau abhängig, sondern technisch erzwungen.

9. Weiterführende Links + Übungsaufgaben

9.1. Offizielle Kernquellen (Apache)

Versions-/Release-Stand:

- Downloads
- Cassandra 5.0.6 Directory

Dokumentations-Einstieg:

- Doku-Start (latest)
- What's new in 5.0

Architektur/Konsistenz:

- Dynamo/Konsistenzlevel
- Architektur-Überblick
- Guarantees

Datenmodell/CQL:

- Data Modeling Intro
- CQL Single File

Compaction/Storage:

- Storage Engine
- Compaction Overview
- STCS
- TWCS
- UCS

Konsistenz/Betrieb:

- Hints
- Repair
- Read Repair
- Snitch

Tools:

- Nodetool Overview
- nodetool status
- nodetool tablestats
- nodetool toppartitions

Sicherheit/Governance:

- Security
- Audit Logging
- Dynamic Data Masking

- cassandra.yaml (Guardrails/Config)

Indexing/SAI:

- SAI FAQ
- SAI Concepts

9.2. Übungsaufgaben (steigende Schwierigkeit)

1. **Schema-Basis (leicht)**

- Implementiere `telemetry_by_device_day` mit sauberem Partition-Key-Bucketing.
- Erzeuge synthetische Last für 10.000 Geräte.
- Miss Partitiongrößen und prüfe Verteilung.

2. **Konsistenzprofil (leicht bis mittel)**

- Definiere drei API-Pfade mit unterschiedlichen CL-Anforderungen.
- Verifiziere Verhalten bei simuliertem Knotenausfall.

3. **Repair-Drill (mittel)**

- Lege einen Repair-Plan für 14 Tage fest.
- Simuliere ausgefallenen Node > 3h und validiere Nachsynchronisation.

4. **Compaction-Vergleich (mittel)**

- Vergleiche STCS, TWCS und UCS anhand derselben Zeitreihenlast.
- Dokumentiere Auswirkungen auf P99, SSTable-Count, Diskverbrauch.

5. **Tombstone-Diagnose (mittel bis schwer)**

- Erzeuge bewusst tombstone-lastigen Workload.
- Führe systematische Diagnose durch und behebe das Modell.

6. **Security Hardening (schwer)**

- Aktiviere TLS client/internode.
- Setze Rollenmodell mit Least Privilege um.
- Aktiviere Audit Logging für relevante Kategorien.

7. **Governance-Drill (schwer)**

- Aktiviere Guardrails gegen definierte Anti-Pattern.
- Prüfe, ob riskante Queries zuverlässig blockiert oder gewarnt werden.

9.3. Abnahmekriterien für "produktionsreif"

- P95/P99-Ziele sind unter Peaklast nachgewiesen.
- Repair-Prozess ist terminiert, getestet und überwacht.
- Hot-Partition-Risiken sind messbar unter Kontrolle.
- Compaction-Strategie ist workload-spezifisch belegt.
- Security-Basis (TLS, AuthZ, Auditing) ist implementiert.

9.4. Abschluss: Was "Pro-Level Cassandra" in der Praxis bedeutet

Pro-Level heißt nicht, jede Funktion zu kennen. Pro-Level heißt:

- Du modellierst bewusst query-first.
- Du triffst RF/CL-Entscheidungen anhand von Ausfallmodellen.
- Du betreibst Repair/Compaction proaktiv statt reaktiv.
- Du kannst Latenzprobleme reproduzierbar diagnostizieren.
- Du hältst Governance und Sicherheit technisch durchsetzbar. Wenn diese fünf Punkte erfüllt sind, ist Cassandra kein Risiko, sondern ein sehr belastbares Fundament für verteilte, schreibintensive Systeme.

9.5. So setzt du es um - Checkliste

- Alle offiziellen Quellen im Team-Wiki als Primärreferenz hinterlegt.
- Übungsaufgaben 1 bis 7 mit messbaren Ergebnissen dokumentiert.
- Für jede Übung sind Lernpunkte und offene Risiken festgehalten.
- Architektur-, Betriebs- und Security-Review gemeinsam abgeschlossen.
- Ein wiederholbarer 90-Tage-Verbesserungsplan ist vereinbart.

9.6. 90-Tage-Umsetzungsfahrplan

Ein realistischer Plan für Teams, die von "Cassandra kennen" zu "Cassandra sicher betreiben" wollen: **Tage 1-30**

- Query-Vertraege finalisieren.
- Tabellenmodell und CL-Profile festziehen.
- Lasttest-Baseline aufbauen. **Tage 31-60**
- Repair-/Compaction-Runbooks automatisieren.
- Alerting auf SLO- und API-Sicht ausrichten.
- Security-Basis (TLS, Rollen, Auditing) nachhaerten. **Tage 61-90**
- Failure-Drills und Restore-Tests fahren.
- Capacity-Plan auf reale Trends anpassen.
- Governance-Metriken etablieren und reporten. Nach 90 Tagen sollte das Team nicht nur Features deployen können, sondern Clusterverhalten voraussagen und Incidents reproduzierbar behandeln können.

9.7. Self-Assessment für Teams

Bewerte euer Team ehrlich mit 0 bis 2 Punkten je Aussage:

- Wir haben für alle Kernendpunkte dokumentierte Query-Vertraege.
- Wir kennen die größten Partitionen und deren Wachstumstrend.
- Wir haben CL-Profile je Endpunkt fachlich abgestimmt.
- Repair ist terminiert, automatisiert und wird nachgewiesen.
- Wir können einen Knotenausfall reproduzierbar behandeln.
- Security-Rollen folgen nachweisbar Least Privilege.
- Unsere Alerts enthalten konkrete nächste Aktionen.

Auswertung:

- 0-5 Punkte: Pionierphase, hohe Betriebsrisiken.
- 6-10 Punkte: solide Basis, aber inkonsistent.
- 11-14 Punkte: gutes Produktionsniveau.

9.8. Nächste Ausbaustufen

Wenn die Basis stabil steht, sind sinnvolle nächste Ausbaustufen:

- Lasttests als Pflichtbestandteil jeder größeren Datenmodell-Aenderung.
- Automatisierte Guardrail-Reports als Governance-Mechanik.
- Regelmäßige Kostenreviews je Datenklasse und Endpunkt.
- Stufenweise Optimierung von Compaction/CL pro Workload. So bleibt das System über Jahre wartbar, auch wenn Traffic, Teams und Anforderungen wachsen. Ergaenze dazu einen quartalsweisen "Cassandra Health Review", der nicht nur technische KPI-Werte sammelt, sondern bewusst die Verbindung zwischen Fachwirkung und Plattformzustand herstellt: Welche API-Pfade haben in den letzten 90 Tagen die meiste Lastveraenderung gesehen, welche Tabellen zeigen den staerksten Partitionswachstumstrend, wo mussten CL-Profile aufgrund neuer Produktanforderungen angepasst werden, und welche Security- oder Governance-Massnahmen wurden tatsächlich wirksam statt nur beschlossen? Dieser Review sollte immer mit konkreten Folgeentscheidungen enden, z. B. neues Partition-Budget für einen Endpunkt, Anpassung der Compaction-Strategie für eine stark gealterte Tabelle, strengere Guardrails für riskante Filterpfade oder gezielte Entlastung nichtkritischer Abfragen durch differenzierte CL-Wahl. Auf diese Weise bleibt Cassandra keine statische "einmal richtig konfigurierte" Datenbank, sondern ein aktiv gesteuertes Produktionssystem, dessen Stabilität, Kosten und Sicherheitsniveau laufend an reale Produktentwicklung angepasst werden. Genau so bleibt Produktionsqualitaet langfristig messbar und steuerbar.

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/apache-cassandra/