Think BIGGER

DEEP DIVE FÜR FACHLEUTE

Neo4j

Graphdatenbank für Beziehungen, Pfade und vernetzte Daten

Autor	Erhard Rainer
Veröffentlicht	31.08.2018
Aktualisiert	06.07.2026
Umfang	ca. 35 Min. Lesezeit
Online	www.thinkbigger.at/wissen/neo4j/

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 Neo4j die richtige Wahl ist 4
2. Architektur & Kernkonzepte 7
3. Datenmodellierung in der Praxis 10
4. Cypher - systematische Praxisreferenz 14
5. Transaktionen, Konsistenz, Cluster, Verfügbarkeit 18
6. Betrieb, Monitoring & Troubleshooting 20
7. Performance, Skalierung & Kostenkontrolle 23
8. Sicherheit, Governance & Compliance 26
9. Weiterführende Links + Uebungsaufgaben 28

Neo4j ist eine Graphdatenbank mit Fokus auf Beziehungen als First-Class-Element: Knoten, Kanten, Eigenschaften und traversale Abfragen stehen im Zentrum, nicht Tabellen und Joins.

Diese Seite ist als praxisorientierte Referenz aufgebaut: Einordnung, Architektur, Datenmodellierung, Cypher, Transaktionen, Cluster-Betrieb, Performance, Security und Uebungsaufgaben.

> Hinweis zum Versionsstand: Diese Referenz ist auf **11. April 2026** ausgerichtet. Die offizielle Neo4j-Dokumentation führt zu diesem Zeitpunkt im Operations Manual die Linien **2026.03 (Current)**, **5.26 (LTS)** und **4.4**. Im Cypher Manual waren zugleich **Version 25**, **Version 5** und **Version 4.4** sichtbar. Versionssensitive Aussagen sind im Text als `2026.03`, `5.26` oder `25` markiert.

0. Überblick (für Eilige)

0.1. Kurzbild

Neo4j ist stark, wenn du:

- Beziehungen selbst abfragen willst, nicht nur Fremdschlüssel aufloesen.
- mehrstufige Traversals, Nachbarschaften, Pfade, Abhängigkeiten oder Netzwerke schnell beantworten musst.
- aus lokalen Graphmustern fachliche Entscheidungen ableitest, etwa Fraud, Empfehlungen, Root-Cause-Analysen oder Impact-Analysen.
- eine deklarative Graphsprache wie Cypher mit gutem Query-Tuning und klaren Datenmodellen nutzen willst.

Neo4j ist schwach, wenn du:

- primär klassische Tabellenberichte mit vielen breiten Aggregationen brauchst.
- einen Graph nur deshalb einfuehren willst, weil "Graph modern klingt", während in Wahrheit einfache relationale Joins reichen.
- ungeplante Massenschreiblast mit quasi beliebiger horizontaler Write-Skalierung auf einem einzigen logischen Graphen erwartest.
- ohne Modellierungsdisziplin einfach JSON, CSV und relationale Tabellen 1:1 in Knoten kippen willst.

0.2. Inhaltsverzeichnis (kurz)

- 0. Überblick (für Eilige)
- 1. Einordnung: Wann Neo4j die richtige Wahl ist
- 2. Architektur & Kernkonzepte
- 3. Datenmodellierung in der Praxis
- 4. Cypher - systematische Praxisreferenz
- 5. Transaktionen, Konsistenz, Cluster, Verfügbarkeit
- 6. Betrieb, Monitoring & Troubleshooting
- 7. Performance, Skalierung & Kostenkontrolle
- 8. Sicherheit, Governance & Compliance
- 9. Weiterführende Links + Uebungsaufgaben

0.3. Quick-Glossar

- **Node**: Knoten im Graph, z. B. `Customer`, `Device`, `Merchant`.
- **Relationship**: Gerichtete Kante zwischen zwei Knoten, z. B. `USES_DEVICE`, `PAID_AT`.
- **Property**: Schluessel-Wert-Paar auf Knoten oder Beziehungen.
- **Label**: Typmarkierung für Knoten, z. B. `:Customer`.
- **Relationship Type**: Typmarkierung für Beziehungen, z. B. `:TRANSFERRED_TO`.
- **Path**: Folge von Knoten und Beziehungen entlang eines Musters.
- **Pattern**: Graphmuster in Cypher, das mit `MATCH` oder `MERGE` beschrieben wird.
- **Cypher**: Deklarative Graphabfragesprache von Neo4j.
- **Constraint**: Schema-Regel für Eindeutigkeit, Existenz, Typ oder Schluesse.
- **Range Index**: Standardindex für exakte Vergleiche, Bereiche und Sortierungen.
- **Full-Text Index**: Lucene-basierter Suchindex für textuelle Suche.
- **Vector Index**: Index für Embeddings und semantische Aehnlichkeitssuche.
- **Page Cache**: Speicher für Daten und native Indizes zur Vermeidung teurer Disk-Zugriffe.
- **Primary / Secondary**: Rollen einer Datenbankkopie im Cluster für Write-HA bzw. Read-Scaling.
- **PROFILE / EXPLAIN**: Werkzeuge für Query-Plan und Tuning.

0.4. Lernpfad

1. Erst Problemklassen und Architekturlogik verstehen: Warum Graph, wann nicht, wie Cluster und Speicher funktionieren.
2. Dann das Datenmodell aus fachlichen Fragen rueckwaerts entwerfen: Knoten, Beziehungen, Constraints, Traversal-Budgets.
3. Danach Cypher systematisch ueben: `MATCH`, `MERGE`, `WITH`, `UNWIND`, Subqueries, Indexe, `PROFILE`.
4. Anschliessend Betrieb und Troubleshooting festziehen: Backups, Logs, Metriken, Runbooks, Failure-Drills.
5. Zum Schluss Performance, Security und Governance operationalisieren.

0.5. Mini-Projekt (durchgehend)

Wir nutzen ein durchgehendes Szenario:

- Zahlungsplattform mit Kunden, Karten, Geraeten, IP-Adressen, Haendlern und Transaktionen.
- Zielbilder:
- Betrugsringe über gemeinsam genutzte Karten, Geraete oder IPs erkennen.
- Risiko-Scores für Transaktionen in unter 150 ms P95 ableiten.
- Ermittlern nachvollziehbare Beziehungspfade liefern.
- Historische Daten revisionssicher halten und operative Abfragen trotzdem schnell beantworten.

Das Mini-Projekt begleitet mehrere Kapitel:

- Datenmodellierung für Identitäten, Ereignisse und Beziehungen.
- Cypher-Muster für Nachbarschaften, Pfade und Aggregationen.
- Cluster- und Routing-Entscheidungen.
- Runbooks für Hotspots, Locking und lange Queries.
- Security- und Governance-Regeln für sensible Zahlungsdaten.

0.6. Profi-Heuristiken

- Ein Graph ist kein Kunstprojekt. Starte mit 5 bis 10 harten Fragen, die heute schlecht beantwortbar sind.
- Beziehungen sind nur dann wertvoll, wenn du sie aktiv traversierst oder fachlich absicherst.
- Lege Constraints vor dem ersten produktiven `MERGE` an.
- Supernodes sind meist ein Modellierungs- oder Abfrageproblem, nicht nur ein Performanceproblem.
- `PROFILE` gehört in die Entwicklungsroutine, nicht erst in den Incident.
- Cluster liefern hohe Verfügbarkeit und Read-Scaling, aber keine magische Entlastung für schlechtes Modell.
- Trenne operative OLTP-Graphabfragen sauber von schweren analytischen Graphberechnungen.

0.7. So setzt du es um - Checkliste

- Definiere 5 bis 10 kritische Traversal- und Nachbarschaftsfragen vor dem Modell.
- Lege pro Entitaet kanonische Identitäten und Constraints fest.
- Dokumentiere, welche Pfade synchron im API-Pfad und welche asynchron berechnet werden.
- Führe für alle kritischen Queries `EXPLAIN` und `PROFILE` vor Go-Live durch.
- Plane Backups, Restore-Tests, Query-Logs, Sicherheitsrollen und Cluster-Topologie vor Produktion.

0.8. Mentales Modell für den Alltag

Wenn du Neo4j sicher einsetzen willst, hilft ein einfaches, aber konsequentes mentales Modell: Ein Graph ist kein "schickeres Tabellenlayout", sondern ein Zugriffssystem für Beziehungen. Das ändert die Kernfrage im Alltag. In relationalen Projekten ist oft zentral, wie viele Tabellen beteiligt sind und welche Join-Reihenfolge sinnvoll ist. In Neo4j ist die wichtigere Frage meist: "Von welchem Anker starte ich, wie gross ist der lokale Suchraum und welche Beziehungen darf ich expandieren?" Genau dort entstehen später die grossen Unterschiede in Geschwindigkeit, Kosten und Wartbarkeit.

Viele Teams scheitern nicht an Cypher-Syntax, sondern an einem unklaren Betriebsvertrag zwischen Fachbereich, Entwicklung und Plattform: Das Fachmodell beschreibt alles, was irgendwie zusammenhaengt; die Anwendung erwartet freie Mehrhop-Abfragen; der Betrieb will planbare Last. Diese drei Perspektiven müssen früh zusammengefuehrt werden. Ein produktionsreifer Graph ist deshalb nicht der maximal vernetzte Graph, sondern der Graph, dessen Traversals, Identitäten, Constraints, Indizes und Betriebsgrenzen explizit sind. Gute Neo4j-Teams diskutieren weniger über "wie graphig etwas aussieht" und mehr über reproduzierbare Fragen: Welche Startpunkte sind garantiert selektiv? Welche Kanten dürfen in API-Pfaden expandiert werden? Wo endet ein Traversal-Budget? Wo braucht es Voraggregation oder Materialisierung? Wo ist eine SQL- oder Suchmaschine die bessere Ergänzung?

---

1. Einordnung: Wann Neo4j die richtige Wahl ist

1.1. Problemklassen, in denen Neo4j typischerweise stark ist

Neo4j passt besonders gut, wenn mehrere der folgenden Punkte zusammenkommen:

- Das eigentliche Fachwissen steckt in Beziehungen, nicht in isolierten Datensätzen.
- Mehrhop-Abfragen gehören zum Normalfall, nicht zur seltenen Sonderauswertung.
- Pfade, Nachbarschaften, Komponenten, Communities oder Abhängigkeiten sind fachlich relevant.
- Die Fachdomaene lässt sich als Property Graph klar modellieren.

- Das Team ist bereit, in Modellierung, Constraints, Query-Tuning und Betrieb zu investieren.

Typische Einsatzfelder:

- Fraud- und Risikoanalyse.
- Identity Resolution und Customer 360.
- Netzwerk- und Infrastruktur-Abhaengigkeiten.
- Supply-Chain- und Impact-Analysen.
- Wissensgraphen und semantische Verknüpfungen.
- Empfehlungen auf Basis gemeinsamer Beziehungen.
- Zugriffs-, Berechtigungs- und Rollenbeziehungen.

1.2. Wann ein RDBMS oft besser passt

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

- starke tabellarische Konsistenz mit klaren Fremdschlüssel- und Join-Strukturen.
- Standard-Reporting, breite Aggregationen und klassische BI-Abfragen.
- viele einfache CRUD-Endpunkte mit wenig Traversal-Tiefe.
- streng relationale Invarianten ohne graphische Fragestellungen.
- SQL-Standardnaehe und grosses Oekosystem an ETL-, BI- und Reporting-Werkzeugen als Hauptprioritaet.

Neo4j ersetzt ein RDBMS nicht automatisch. In vielen produktiven Landschaften ist es eine Spezialisierung für Beziehungspfad-Workloads, nicht das universelle System für alles.

1.3. Neo4j vs. relationale Datenbank - praktische Perspektive

Neo4j-Vorteile:

- Beziehungen sind direkt modelliert statt indirekt über Join-Tabellen.
- Mehrhop-Abfragen bleiben fachlich lesbar und operativ schnell.
- Pfad- und Nachbarschaftslogik ist in Cypher meist deutlich nativer ausdrueckbar.
- Das Datenmodell passt gut zu Netzwerken, Abhängigkeiten und Verknüpfungen.

RDBMS-Vorteile:

- Hohe Reife für klassische OLTP/OLAP-Muster.
- Sehr starke Integritaets- und Standardisierungsmechanismen.
- Breites Tooling für Reporting, ETL, CDC, Data Warehousing und Governance.
- Oft einfacher für Teams, die rein tabellarisch denken und arbeiten.

Die Kernfrage ist nicht "Graph oder SQL?", sondern:

- Sind Beziehungen fachlich der Primaergegenstand?
- Sind Traversals eine Kernoperation oder nur eine seltene Spezialabfrage?
- Lassen sich kritische Fragen mit einem Graphmodell deutlich nativer und robuster beantworten?

1.4. Neo4j vs. Suchmaschine und Vector Store

Neo4j konkurriert nicht nur mit SQL, sondern oft auch mit Such- und KI-Systemen:

- Eine Suchmaschine ist besser für Volltextsuche, Ranking und Dokumentretrieval.

- Ein reiner Vector Store ist oft gut für Embedding-Retrieval, aber schwach bei strukturierter Beziehungslogik.
- Neo4j ist stark, wenn semantische Ähnlichkeit mit expliziter Beziehungslogik kombiniert werden soll.

Praxisregel:

- Volltext = Suchproblem.
- Embedding-Aehnlichkeit = Vector-Problem.
- "Wer ist über welche Kette mit wem verbunden, und warum ist das relevant?" = Graphproblem.

Neo4j kann Full-Text- und Vector-Indizes integrieren, aber der Mehrwert entsteht erst, wenn diese Suche mit dem Graphkontext zusammenarbeitet.

1.5. Harte No-Go-Faelle für Neo4j

- Das Team hat keine konkreten Graphfragen und hofft auf spätere Einsatzzwecke.
- Der Graph soll nur als Spiegel eines relationalen Schemas existieren.
- Der Hauptnutzen wäre tabellarisches Reporting ohne tiefe Traversals.
- Es gibt keine Bereitschaft für Constraints, Idempotenz und Query-Tuning.
- Man erwartet horizontale Write-Skalierung für einen einzigen logischen Graphen wie bei verteilten Key-Value-Systemen.

> **Anti-Pattern:** "Wir importieren erstmal alles als Knoten und schauen dann, welche Beziehungen sich vielleicht ergeben." Ergebnis in der Praxis: unklare Identitäten, Duplikate, schlechte `MERGE` -Semantik, schwache Performance und geringe fachliche Aussagekraft.

1.6. Entscheidungsmatrix (kompakt)

Kriterium	Neo4j passt	Neo4j passt eher nicht
Kernfrage	Beziehungen, Pfade, Nachbarschaften	Tabellen, Summen, einfache CRUD
Hauptoperation	Traversals über mehrere Hops	Standard-Join oder reine Volltextsuche
Datenmodell	Netzwerkartig, verknüpft	Strikt tabellarisch
Performanceziel	Schnelle lokale Graphfragen	Massenaggregation und Reporting
Teammodus	Modellierungs- und Query-Disziplin	"Schema später" / Ad-hoc ohne Grenzen

1.7. So setzt du es um - Checkliste

- Dokumentiere mindestens 5 konkrete Fragen, die ohne Graph schlecht oder teuer sind.
- Vergleiche ein Neo4j-Modell mit mindestens einer SQL- und ggf. Suchmaschinen-Alternative.
- Definiere harte Erfolgskriterien: P95, Query-Lesbarkeit, Incident-Faehigkeit, Betriebsaufwand.
- Halte fest, welche Teile der Landschaft bewusst nicht in Neo4j gehören.

1.8. Praxisvergleich: Graph-first vs. "wir mappen Tabellen später um"

Der häufigste Einfuehrungsfehler ist, relationale Entitaeten 1:1 in Knoten zu ueberfuehren und Join-Tabellen mechanisch in Beziehungen zu verwandeln. Das wirkt am Anfang produktiv, führt aber später oft zu zwei Problemen:

- Das Modell beantwortet die eigentlichen Fragen nicht sauber.

- Die Traversals starten nicht an selektiven Punkten, sondern laufen zu breit.

Die bessere Reihenfolge ist:

1. Fachliche Kernfragen als Muster formulieren.
2. Kanonische Identitäten und Eindeutigkeitsregeln festlegen.
3. Erst dann Labels, Beziehungstypen, Properties und Constraints modellieren.
4. Schliesslich Queries, Indizes, Pfadbudgets und Runbooks aufbauen.

1.9. Architekturentscheidungen sauber dokumentieren

Bewaehrte ADR-Punkte für Neo4j:

- Warum Graph und nicht SQL oder Suchsystem?
- Welche Queries sind first-class, welche second-class?
- Welche Knoten und Beziehungen sind kanonisch?
- Welche Constraints müssen immer gelten?
- Welche Traversals sind im synchronen API-Pfad erlaubt?
- Wo endet der operative Graph und wo beginnt Analytics/GDS?
- Welche Cluster- und Backup-Entscheidungen folgen daraus?

---

2. Architektur & Kernkonzepte

2.1. Property-Graph-Modell

Neo4j basiert auf dem Property-Graph-Modell:

- Knoten repraesentieren Entitaeten.
- Beziehungen repraesentieren semantische Verknüpfungen.
- Beide können Properties tragen.
- Knoten können mehrere Labels tragen.
- Beziehungen haben genau einen Typ.

Der grosse Vorteil liegt darin, dass Beziehungen nicht nur "Fremdschluesselspalten mit Bedeutung" sind, sondern adressierbare, traversierbare Strukturen mit eigener Semantik.

2.2. Knoten, Beziehungen, Labels und Richtung

Wichtige Modellierungsregeln:

- Labels sollten fachliche Klassen oder Rollen markieren, nicht zufaellige Filter-Flags.
- Beziehungstypen sollten sprechend und stabil sein.
- Richtung ist fachlich bedeutend und sollte nicht beliebig wirken.
- Properties gehören dorthin, wo ihre Lebensdauer und fachliche Geltung klar ist.

Praxisbeispiel:

- `(:Customer)-[:USES_DEVICE]->(:Device)`
- `(:Customer)-[:OWNS_CARD]->(:Card)`
- `(:Transaction)-[:PAID_AT]->(:Merchant)`

Schon diese drei Beziehungstypen erlauben Queries, die in Tabellenmodellen oft schnell unuebersichtlich werden.

2.3. Pfade, Pattern Matching und lokale Expansion

Graphabfragen starten typischerweise an einem selektiven Einstiegspunkt und expandieren dann lokal über passende Kanten. Genau deshalb ist die Wahl des Startpunktes so wichtig.

Gute Startpunkte sind etwa:

- eindeutig identifizierte Knoten per Constraint oder Index.
- schmale Kandidatenmengen über selektive Property-Filter.
- bekannte Anchors wie `customer_id`, `transaction_id`, `device_fingerprint`.

Schlechte Startpunkte sind:

- breite Labels ohne zusätzliche Selektion.
- variable Pfade ohne Hop-Grenzen.
- globale Traversals über schlecht kontrollierte Supernodes.

2.4. Speicher, Indizes und Page Cache

Neo4j lebt stark von seinem Speicherprofil:

- Heap steuert JVM-seitige Verarbeitung und Query-Objekte.
- Native/off-heap-Speicher wird für interne Komponenten genutzt.
- Der Page Cache puffert Graphdaten und native Indizes und vermeidet teure Disk-Zugriffe.

Die offizielle Doku empfiehlt, Heap und Page Cache explizit zu konfigurieren, statt nur auf Heuristiken zu vertrauen. Für produktive Systeme ist das Pflicht, nicht Feintuning.

Wichtige Indexfamilien:

- Search-performance indexes für exakte Vergleiche, Bereiche, Sortierung und Startpunkte in `MATCH`.
- Full-Text-Indexes für textuelle Suche.
- Vector-Indexes für Embeddings und semantische Nachbarschaft.

2.5. Cypher-Planer, Runtime und `PROFILE`

Neo4j plant Queries nicht "magisch richtig", sondern auf Basis deines Modells, deiner Indizes und deiner Praedikate. Search-performance indexes werden vom Planner in `MATCH`-Klauseln automatisch verwendet, wenn die Voraussetzungen stimmen. Full-Text-Indexes dagegen werden nicht automatisch vom Planner genutzt, sondern über Prozeduren angesprochen. Vector-Indexes werden in aktuellen Linien bevorzugt über `SEARCH` abgefragt.

Wichtig für den Alltag:

- `EXPLAIN` vor dem produktiven Einsatz.
- `PROFILE` für reale Plan- und DB-Hit-Analyse.
- Operatoren wie `NodeByLabelScan`, `NodeIndexSeek`, `Expand(All)` und späte `Filter` bewusst lesen.

2.6. Transaktionen, Locks und Hotspots

Neo4j ist ein ACID-System, aber nicht lockfrei. Bei konkurrierenden Updates auf dieselben Knoten oder Beziehungen entsteht Lock Contention. Die Operations-Doku zeigt explizit:

- konkurrierende Updates auf dieselben Strukturen erhöhen Deadlock-Risiko.
- aktive Locks lassen sich über `SHOW TRANSACTIONS` und `CALL dbms.listActiveLocks(...)` analysieren.
- es gibt einen konfigurierbaren `db.lock.acquisition.timeout`.

Besonders relevant sind Hotspots:

- viele konkurrierende Updates auf denselben Knoten.
- globale Counter-Knoten.
- schlecht modellierte Supernodes im Write-Pfad.

2.7. Standalone, Cluster, Primaries und Secondaries

Neo4j-Cluster sind ein Enterprise-Thema und folgen heute einem Modell pro Datenbankkopie:

- Primaries tragen die Write-Verantwortung.
- Nur eine Primary ist zu einem Zeitpunkt der Writer für eine Datenbank.
- Writes werden synchron auf genug Primaries bestätigt.
- Secondaries replizieren asynchron und dienen primär dem Read-Scaling.

Die offizielle Clustering-Doku nennt als Kernpunkte:

- Writes bleiben verfügbar, solange eine einfache Mehrheit der Primaries funktioniert.
- Drei Primaries sind der typische Startpunkt für Fault Tolerance gegen einen Ausfall.
- Mehr Primaries können die Write-Latenz erhöhen.
- Secondaries skalieren Leseabfragen, können aber temporär hinterherhinken.

2.8. Mini-Projekt: Architekturentscheidung

Für die Zahlungsplattform aus Abschnitt 0 ist Neo4j sinnvoll, weil:

- Betrugsmuster über gemeinsam genutzte Ressourcen graphisch natuerlich sind.
- Ermittler erklaerbare Pfade sehen müssen, nicht nur Scores.
- lokale Beziehungsfragen operational relevant sind.

Architekturentscheidung:

- Operativer Graph für Echtzeitpruefungen und Erklaerpfade.
- klare Constraints für Identitäten.
- Cluster mit mehreren Primaries für Write-HA und Secondaries für Ermittler-Reads.
- schwere Community- oder Batch-Analysen ausserhalb des heissen OLTP-Pfads.

2.9. So setzt du es um - Checkliste

- Definiere Ankerknoten für alle kritischen Queries.
- Plane Page Cache und Heap explizit.
- Lege fest, welche Indexfamilie welchem Query-Typ dient.
- Führe vor Cluster-Go-Live Failure- und Routing-Tests durch.
- Dokumentiere Hotspot-Risiken pro Schreibpfad.

2.10. Cluster-Realitaet: hohe Verfügbarkeit ja, beliebige Write-Skalierung nein

Ein wichtiger Realitaetscheck: Neo4j-Cluster verbessern Verfügbarkeit und Read-Scaling, aber ein einzelner logischer Graph bleibt kein frei shardbares Key-Value-System. Mehr Primaries bedeuten nicht automatisch lineare Beschleunigung, sondern zusätzliche Replikations- und Koordinationskosten pro Write.

Praxisfolgen:

- Modelliere den Write-Pfad schlank.
- Halte Transaktionen kurz.
- Vermeide unnötig breite `MERGE`-Muster.
- Plane analytische Schwerlast getrennt vom operativen Graph.

2.11. Speicher-Mentalmodell: Heap, Native, Page Cache

Für den Alltag hilft ein dreiteiliges Mentalmodell:

- Heap: Query-Zwischenzustand, Objekte, Ergebnismengen.
- Native/off-heap: DBMS- und Datenbank-Overhead ausserhalb des GC.
- Page Cache: möglichst viel des heissen Graphen und der nativen Indizes im Speicher halten.

Typische Fehler:

- Heap zu klein für komplexe Abfragen.
- Page Cache zu klein für heisse Datensets.
- Speicher "automatisch" belassen und erst im Incident hinsehen.

2.12. Architektur an SLOs koppeln

Architektur wird belastbar, wenn sie direkt an Fachziele gekoppelt ist:

- SLO: Risikopruefung pro Zahlung < 150 ms P95.
- Folge: Start immer an selektiven Knoten wie `Transaction` oder `Card`.
- Folge: Traversal-Budget maximal 2 bis 3 Hops im Sync-Pfad.
- Folge: komplexere Netzwerkanalysen asynchron.

Wenn dieser Zusammenhang fehlt, wird Grapharchitektur schnell zur Stilfrage statt zu einer belastbaren Produktionsentscheidung.

---

3. Datenmodellierung in der Praxis

3.1. Die wichtigste Regel: Modelliere von den Fragen rueckwaerts

In Neo4j ist die Startfrage nicht:

"Welche Entitaeten haben wir?"

Sondern:

"Welche fachlichen Fragen müssen in Millisekunden oder Sekunden beantwortbar sein, und entlang welcher Beziehungen laufen diese Fragen?"

Beispiel-Fragen:

- Welche anderen Kunden teilen mit diesem Kunden Karten, Geraete oder IPs?
- Welche Transaktionen liegen maximal 2 Hops von bereits bestätigten Fraud-Faellen entfernt?
- Welche Haendler sind über verdaechtige Ressourcen indirekt verbunden?

3.2. Kanonische Entitaeten und Identitäten festlegen

Lege für zentrale Knoten früh fest:

- Was ist die eindeutige Identität?
- Woher kommt sie?
- Darf sie sich ändern?
- Wie wird Duplikatbildung verhindert?

Typische Kandidaten:

- `Customer.customer_id`
- `Card.card_token`
- `Device.device_fingerprint`
- `Merchant.merchant_id`
- `Transaction.transaction_id`

Ohne diese Disziplin wird `MERGE` schnell gefährlich, weil Duplikate nicht mehr sauber verhindert werden.

3.3. Labels und Beziehungstypen bewusst schneiden

Bewaehrte Regeln:

- Labels für stabile Klassen oder Rollen.
- Beziehungstypen für fachlich aussagekraeftige Verbindungen.
- Nicht jede Berichtsperspektive braucht ein neues Label.
- Nicht jeder technische Join verdient eine eigene Beziehung.

Schlechter Stil:

- `:Entity`, `:Thing`, `:Object`
- `:RELATED_TO`
- Labels nur für temporäre Filterzwecke

Guter Stil:

- `:Customer`, `:Card`, `:Device`, `:Merchant`, `:Transaction`
- `:OWNS_CARD`, `:USES_DEVICE`, `:PAID_AT`, `:TRANSFERRED_TO`

3.4. Properties am richtigen Ort platzieren

Frage pro Eigenschaft:

- Gehört sie zum Knoten?
- Gehört sie zur Beziehung?
- Gehört sie zu einem Ereignisknoten?

Beispiel:

- `Merchant.name` gehört auf den Knoten.
- `:USES_DEVICE {first_seen_at, last_seen_at}` kann auf der Beziehung sinnvoll sein.
- Einzelne Zahlungsvorgaenge sind oft eigene Ereignisknoten `:Transaction` statt Beziehungs-Properties.

3.5. Constraints vor `MERGE` anlegen

Das ist keine Stilfrage, sondern Produktionsschutz. Neo4j bietet u. a.:

- Property uniqueness constraints.
- Property existence constraints. Enterprise Edition
- Property type constraints. Enterprise Edition
- Key constraints. Enterprise Edition

Praxisregel:

- Vor dem ersten produktiven `MERGE` für jede kanonische Identität mindestens Eindeutigkeit absichern.

Beispiel:

```
CREATE CONSTRAINT customer_id_unique IF NOT EXISTS
FOR (c:Customer) REQUIRE c.customer_id IS UNIQUE;

CREATE CONSTRAINT transaction_id_unique IF NOT EXISTS
FOR (t:Transaction) REQUIRE t.transaction_id IS UNIQUE;

CREATE CONSTRAINT device_fp_unique IF NOT EXISTS
FOR (d:Device) REQUIRE d.device_fingerprint IS UNIQUE;
```

3.6. Supernodes früh erkennen und entlasten

Supernodes sind Knoten mit extrem vielen Beziehungen. Sie entstehen oft bei:

- Sammelknoten wie "Alle Transaktionen von heute".
- generischen Tags oder Kategorien.
- Geraeten, IPs oder Haendlern mit sehr hoher Wiederverwendung.

Gegenmassnahmen:

- nicht alles an einen globalen Knoten hängen.
- Zeit- oder Bereichssegmentierung einziehen.
- für den heissen API-Pfad Traversal-Tiefe und Richtung strikt begrenzen.
- bei Bedarf Materialisierung oder Voraggregation einsetzen.

3.7. Zeit, Historie und Ereignisse modellieren

Viele Graphmodelle scheitern daran, dass Status und Ereignisse vermischt werden.

Bewaehrte Muster:

- Stabile Entitaeten als Knoten.
- Zeitpunkte, Transaktionen oder Vorfaelle als Ereignisknoten.
- Beziehungs-Properties für leichte Historien, nicht für komplette Event-Speicher.

Für das Fraud-Szenario ist ein eigener `:Transaction`-Knoten oft sinnvoll, weil:

- er mehrere Parteien verbindet.

- er Scores, Zeit, Betragsinformationen und Flags tragen kann.
- historische Abfragen dadurch sauberer bleiben.

3.8. Multi-Tenancy und fachliche Grenzen

Wenn mehrere Mandanten im selben Graph leben, muss die Grenze explizit sein:

- Tenant-Knoten oder Tenant-Property als harte Modellregel.
- Zugriff nur über tenant-selektive Startpunkte.
- Rollen- und Privilegienmodell daran koppeln.

Ein "wir filtern später irgendwie auf Tenant" ist im Graph besonders riskant, weil Traversals sehr schnell über unerwuenschte Grenzen laufen können.

3.9. Mini-Projekt: Beispielmodell

Ein sinnvolles Basismodell könnte so aussehen:

```
(:Customer)-[:OWNS_CARD]->(:Card)
(:Customer)-[:USES_DEVICE]->(:Device)
(:Customer)-[:SEEN_FROM]->(:IP)
(:Transaction)-[:INITIATED_BY]->(:Customer)
(:Transaction)-[:USES_CARD]->(:Card)
(:Transaction)-[:FROM_DEVICE]->(:Device)
(:Transaction)-[:FROM_IP]->(:IP)
(:Transaction)-[:PAID_AT]->(:Merchant)
```

Damit lassen sich schon sehr viele Risiko- und Nachbarschaftsfragen direkt formulieren.

3.10. Modellierungs-Anti-Patterns

- Relationale Join-Tabellen mechanisch als Kanten übernehmen, ohne fachliche Prüfung.
- Generische `RELATED_TO`-Beziehungen.
- Keine Constraints für kanonische Identitäten.
- Alles als Knoten modellieren, auch wenn ein Ereignis- oder Property-Muster besser wäre.
- Traversal-relevante Eigenschaften ohne passenden Startpunkt oder Index.

3.11. So setzt du es um - Checkliste

- Pro Kernentitaet existiert eine dokumentierte Identitaetsregel.
- Pro kritischer Frage ist das Traversal-Muster beschrieben.
- `MERGE` auf kanonischen Entitaeten läuft nur gegen abgesicherte Constraints.
- Supernode-Risiken sind für mindestens 3 zentrale Knotenklassen bewertet.
- Historie und Status sind bewusst getrennt modelliert.

3.12. Datenlebenszyklus, Archivierung und Löschung

Graphen werden schnell unuebersichtlich, wenn Lebenszyklen fehlen. Plane deshalb früh:

- Welche Ereignisknoten müssen online bleiben?
- Welche Beziehungen sind nur operativ relevant?
- Welche Daten müssen aus Compliance-Gruenden anonymisiert oder gelöscht werden?
- Welche Daten werden in einen Archiv- oder Analysegraph ueberfuehrt?

Gerade bei Transaktions- oder Security-Daten ist "nie wieder löschen" meist kein belastbares Betriebsmodell.

3.13. Modellentwicklung über Migrationspfade absichern

Ein Graphmodell ändert sich im Projektverlauf. Deshalb braucht ihr:

- versionierte Migrationsskripte.
- klare Benennungsregeln für Labels und Beziehungstypen.
- Backfill-Strategien für neue Properties oder Constraints.
- Tests, die Query-Vertraege absichern.

Ein gutes Neo4j-Modell ist nicht statisch, aber seine Änderungen müssen reproduzierbar und reviewbar sein.

---

4. Cypher - systematische Praxisreferenz

4.1. Grundmuster: `MATCH`, `WHERE`, `RETURN`

Cypher beschreibt Muster im Graphen. Ein einfaches Beispiel:

```
MATCH (c:Customer {customer_id: $customerId})-[:OWNS_CARD]->(card:Card)
RETURN c.customer_id, collect(card.card_token) AS cards;
```

Wichtige Prinzipien:

- starte möglichst selektiv.
- filtere früh.
- expandiere nur über fachlich erlaubte Kanten.
- gib nur benötigte Daten zurück.

4.2. Schreiben mit `CREATE`, `MERGE`, `SET`

`CREATE` erzeugt immer neue Strukturen, `MERGE` sucht erst ein Muster und erzeugt nur bei Bedarf.

```
MERGE (c:Customer {customer_id: $customerId})
ON CREATE SET c.created_at = datetime()
SET c.last_seen_at = datetime();
```

Praxisregel:

- `CREATE` für bewusst neue Ereignisse.
- `MERGE` für kanonische Identitäten und idempotente Upserts.

4.3. `MERGE` richtig einsetzen

`MERGE` ist stark, aber gefährlich, wenn Constraints fehlen oder Muster zu breit sind. Die Cypher-Doku beschreibt explizit All-or-Nothing-Semantik und verweist darauf, bei Eindeutigkeit und besonders bei Relationship-Uniqueness mit gebundenen Knoten zu arbeiten.

Praxisregeln:

- `MERGE` zuerst auf Knoten mit eindeutiger Identität.

- Danach Beziehungen separat `MERGE`n.
- Keine riesigen Muster auf einmal `MERGE`n.

Besser so:

```
MERGE (c:Customer {customer_id: $customerId})
MERGE (d:Device {device_fingerprint: $deviceFingerprint})
MERGE (c)-[r:USES_DEVICE]->(d)
ON CREATE SET r.first_seen_at = datetime()
SET r.last_seen_at = datetime();
```

4.4. Aktualisieren und Löschen

Hauefige Werkzeuge:

- `SET` für Properties und Labels.
- `REMOVE` für Labels oder Properties.
- `DELETE` für einzelne Elemente ohne abhängige Beziehungen.
- `DETACH DELETE` für Knoten inklusive ihrer Beziehungen.

```
MATCH (t:Transaction {transaction_id: $txId})
SET t.risk_score = $riskScore,
    t.risk_reason = $riskReason,
    t.updated_at = datetime();
```

Löschungen im Graph sind fachlich oft heikel. Prüfe vorher:

- braucht ihr Historie?
- braucht ihr Soft Deletes?
- können Beziehungen für spätere Audits relevant sein?

4.5. `WITH`, `UNWIND` und Subqueries

Diese drei Werkzeuge machen Cypher im Alltag deutlich stabiler:

- `WITH` begrenzt Scope und strukturiert mehrstufige Abfragen.
- `UNWIND` verarbeitet Listen sauber batchweise.
- Subqueries isolieren Logik und verbessern Lesbarkeit.

```
UNWIND $deviceFingerprints AS fp
MERGE (d:Device {device_fingerprint: fp})
RETURN count(*) AS processed;
```

```
MATCH (c:Customer {customer_id: $customerId})
CALL {
  WITH c
  MATCH (c)-[:USES_DEVICE]->(d:Device)<-[:USES_DEVICE]-(other:Customer)
  WHERE other <> c
  RETURN collect(DISTINCT other.customer_id) AS linked_customers
}
RETURN c.customer_id, linked_customers;
```

4.6. Aggregationen und Nachbarschaftsabfragen

Cypher ist stark, wenn du lokale Muster aggregieren willst:

```
MATCH (t:Transaction)-[:FROM_DEVICE]->(d:Device)<-[:FROM_DEVICE]-(other:Transaction)
WHERE t.transaction_id = $txId AND other <> t
RETURN d.device_fingerprint, count(other) AS neighbouring_tx
ORDER BY neighbouring_tx DESC
LIMIT 10;
```

Wichtige Regel:

- Aggregiere möglichst nach lokaler Selektion.
- Vermeide globale Zaehlungen über breite Labelscans im Request-Pfad.

4.7. Variable Pfade und kuerzeste Wege

Variable Muster sind verlockend:

```
MATCH p = (c:Customer)-[:USES_DEVICE|OWNS_CARD|SEEN_FROM*1..3]-(other:Customer)
WHERE c.customer_id = $customerId AND other <> c
RETURN other.customer_id, length(p)
LIMIT 20;
```

Aber:

- Begrenze Hop-Zahlen hart.
- Nutze selektive Startpunkte.
- Führe solche Abfragen nur dann synchron aus, wenn das Traversal-Budget kontrolliert ist.

Ohne Grenzen werden variable Pfade schnell zu Produktionsrisiken.

4.8. Import: `LOAD CSV` und `neo4j-admin database import`

Für moderate Datenimporte ist `LOAD CSV` oft ausreichend. Für initiale Grossimporte ist `neo4j-admin database import` der richtige Weg. Laut Operations Manual gibt es dafür zwei Modi:

- `full` für leere, noch nicht existierende Datenbanken.
- `incremental` für gestueckelte Initialimporte.

`LOAD CSV` Beispiel:

```
LOAD CSV WITH HEADERS FROM $url AS row
MERGE (c:Customer {customer_id: row.customer_id})
SET c.country = row.country;
```

Offline-/Bulk-Import Beispiel:

```
bin/neo4j-admin database import full --nodes=import/customers.csv neo4j
```

4.9. Indizes und Constraints im Alltag

Such- und leistungsrelevante DDL-Beispiele:

```
CREATE INDEX transaction_created_at IF NOT EXISTS
FOR (t:Transaction) ON (t.created_at);

CREATE INDEX merchant_country IF NOT EXISTS
FOR (m:Merchant) ON (m.country);
```

Full-Text:

```
CREATE FULLTEXT INDEX merchant_text IF NOT EXISTS
FOR (m:Merchant) ON EACH [m.name, m.city, m.category];
```

Vector:

```
CREATE VECTOR INDEX transaction_embedding IF NOT EXISTS
FOR (t:Transaction) ON (t.embedding)
OPTIONS {indexConfig: {
  `vector.dimensions`: 1536,
  `vector.similarity_function`: 'cosine'
}};
```

4.10. Full-Text und Vector Search

Wichtige Unterschiede:

- Full-Text-Indexes werden nicht automatisch vom Planner in normalen `MATCH`-Abfragen benutzt.
- Full-Text-Queries laufen über `db.index.fulltext.queryNodes()` oder `...queryRelationships()`.
- Vector-Indexes lassen sich in aktuellen Linien bevorzugt über `SEARCH` ansprechen.

Full-Text-Beispiel:

```
CALL db.index.fulltext.queryNodes("merchant_text", "electronics AND berlin")
YIELD node, score
RETURN node.name, score
ORDER BY score DESC
LIMIT 10;
```

Vector-Beispiel:

```
MATCH (t:Transaction {transaction_id: $txId})
MATCH (candidate:Transaction)
  SEARCH candidate IN (
    VECTOR INDEX transaction_embedding
    FOR t.embedding
    LIMIT 5
  ) SCORE AS score
RETURN candidate.transaction_id, score;
```

Die aktuelle Doku nennt für Vector-Indexes u. a.:

- Similarity Functions `cosine` und `euclidean`
- HNSW-Parameter wie `vector.hnsw.m`
- Quantisierung als Performance-/Speicherhebel

4.11. `EXPLAIN`, `PROFILE` und Plan-Smells

Pflichtblick beim Tuning:

- startet die Query mit `NodeIndexSeek` oder fällt sie auf breitere Scans zurück?
- tauchen späte Filter auf, die früher hätten greifen müssen?
- ist die Ergebnismenge früh genug klein?
- steigen DB-Hits unverhaeltnismaessig?

Plan-Smells:

- `AllNodesScan`
- breite `NodeByLabelScan` im heissen Pfad

- ungebremstes `Expand(All)`
- viel Arbeit vor der eigentlichen Selektion

4.12. So setzt du es um - Checkliste

- Jede kritische Query hat eine klare Anchor-Strategie.
- `MERGE` wird nur auf abgesicherten Identitäten verwendet.
- Variable Pfade haben feste Hop-Grenzen.
- Für Full-Text und Vector sind eigene Zugriffspfade dokumentiert.
- `PROFILE`-Ergebnisse sind für Kernqueries archiviert.

4.13. Typische Cypher-Anti-Patterns

- `MERGE` über riesige, ungebundene Muster.
- globale `MATCH`-Scans mit nachgelagertem Filter.
- mehrere verschiedene fachliche Schritte in einer unlesbaren Monsterquery.
- `UNWIND` riesiger Listen ohne Memory-Grenzen.
- im API-Pfad "nur mal schnell" variable Pfade ohne harte Grenzen.

---

5. Transaktionen, Konsistenz, Cluster, Verfügbarkeit

5.1. ACID und Transaktionsgrenzen

Neo4j ist transaktional. Das ist ein grosser Vorteil gegenüber vielen lockeren Graph- oder Suchsystemen:

- mehrere Schreibschritte können atomar zusammengefasst werden.
- Konsistenzregeln über Constraints greifen innerhalb dieses Modells.
- Transaktionen sind rollbackfaehig.

Praxisregel:

- halte Transaktionen kurz.
- fasse fachlich zusammengehoerige, aber nicht unnötig breite Operationen zusammen.
- vermeide langlaufende Lese-Transaktionen im heissen Produktionspfad.

5.2. Cluster-Konsistenz praktisch lesen

Die offizielle Clustering-Doku formuliert drei Punkte, die in der Praxis zentral sind:

- Jede Datenbank hat ihre eigene Topologie aus Primaries und Secondaries.
- Writes bleiben verfügbar, solange eine einfache Mehrheit der Primaries funktioniert.
- Neo4j bietet, wenn vom Client genutzt, kausale Konsistenz im Sinn von "read your own writes".

Das heisst für den Alltag:

- leseverteilte Queries auf Secondaries sind stark für Ermittler- oder Analyse-Reads.
- kritische Sofort-Lesevorgaenge nach einem Write müssen Routing und Konsistenzbedarf beachten.

5.3. Primaries, Secondaries und Routing

Wichtige Praxisaussagen:

- Secondaries replizieren asynchron und können hinterherhinken.
- Der Verlust einer Secondary reduziert Durchsatz, aber nicht die Write-Verfuegbarkeit.
- Zu viele Primaries erhöhen Write-Koordination und damit potenziell Latenz.

Deshalb:

- drei Primaries sind oft der produktive Startpunkt.
- Secondaries nur dort erhöhen, wo Read-Last es wirklich verlangt.
- Routing-Regeln an reale Lastprofile koppeln, nicht pauschal.

5.4. Retry-Semantik und Fehlerklassen

Im verteilten Betrieb müssen Anwendungen mit transienten Fehlern umgehen können:

- kurze Cluster-Umschaltungen
- Lock-Konflikte
- zeitweilige Ressourcenengpaesse

Praxisregel:

- nutze Treiberfunktionen für gemanagte Transaktionen und Wiederholungen dort, wo sie fachlich sicher sind.
- halte Schreiboperationen idempotent oder explizit schutzbewehrt.
- protokolliere, welche Fehler retrybar und welche fachlich terminal sind.

5.5. Lock Contention und Deadlocks

Die offizielle Doku weist klar darauf hin, dass konkurrierende Updates auf denselben Knoten oder Beziehungen Lock-Wartezeiten und Deadlocks erzeugen können.

Typische Ursachen:

- gleichzeitige Updates auf denselben `Customer`-, `Card`- oder `Device`-Knoten.
- viele parallele `MERGE`-Operationen auf dieselben Identitäten.
- globale Sammelknoten.

Gegenmassnahmen:

- Transaktionen kurz halten.
- Schreibreihenfolgen deterministisch gestalten.
- Hotspot-Modelle ueberdenken.
- `db.lock.acquisition.timeout` bewusst setzen.

5.6. Konsistenzprofil für das Mini-Projekt

Für die Zahlungsplattform bietet sich an:

- Write-Pfad für neue Transaktionen auf Primaries.
- unmittelbare Fraud-Pruefung an den gleichen frischen Daten.
- Ermittler- und Recherche-Reads bevorzugt auf Secondaries.
- schwere Graphanalysen ausserhalb des Request-Pfads.

So bleibt der operative Kern stabil, ohne investigative Use Cases zu blockieren.

5.7. So setzt du es um - Checkliste

- Pro Datenbank ist die Topologie dokumentiert.
- Kausale Konsistenzanforderungen sind pro Anwendungspfad benannt.
- Retry-Regeln sind fachlich definiert, nicht nur technisch.
- Lock-Hotspots sind für kritische Writes bekannt.
- Failure-Drills mit Primary-Ausfall und Secondary-Lag wurden getestet.

5.8. Failure-Drills, die wirklich helfen

Sinnvolle Uebungen:

- Ausfall einer Primary im 3-Primary-Setup.
- gezielt erzeugter Secondary-Lag.
- Lock-Contention durch parallele Updates auf denselben Identitäten.
- Restore-Test nach Backup auf isolierter Umgebung.

Wichtig:

- nicht nur "Cluster war wieder gruen" messen.
- vor allem API-Verhalten, Fehlermuster und Runbook-Qualitaet bewerten.

5.9. Multi-Region-Realitaet

Neo4j kann resilient über mehrere Datenzentren oder Regionen geplant werden, aber Latenz bleibt Physik. Mehr Regionendistanz kann die Write-Latenz in Primary-Topologien spuerenbar erhöhen. Deshalb müssen Multi-Region-Designs immer an reale SLOs gekoppelt werden:

- Wo müssen Writes entstehen?
- Wo dürfen Reads stale sein?
- Welche Datenbanken brauchen wirklich geo-resiliente Topologien?

---

6. Betrieb, Monitoring & Troubleshooting

6.1. Deployment-Optionen

Typische Betriebsformen:

- Standalone für Entwicklung oder kleinere produktive Spezialsysteme.
- Self-managed Cluster auf VMs oder Bare Metal.
- Docker oder Kubernetes für standardisierte Deploymentablaeufe.
- Aura, wenn Plattformbetrieb bewusst ausgelagert werden soll.

Wichtig ist weniger das Schlagwort als die Betriebsfaehigkeit:

- Wer macht Backups?
- Wer testet Restores?
- Wer betreut Konfiguration, Memory, Query-Logs und Security?

6.2. Backups, Dumps und Restore

Neo4j unterscheidet operativ zwischen:

- online Backups laufender Enterprise-Server über `neo4j-admin database backup`
- offline Dumps/Loads für entsprechende Betriebsfaelle

Die Operations-Doku nennt für Online-Backups explizit:

- Full- und Differential-Backups
- `neo4j-admin database backup`
- Backup-Port-Haertung
- kein Support für dieses Verfahren in Aura

Beispiel:

```
bin/neo4j-admin database backup --to-path=/path/to/backups/neo4j neo4j
```

Wichtige Regel:

- Ein Backup ist erst dann belastbar, wenn ein Restore-Test erfolgreich durchgefuehrt wurde.

6.3. Logs und Metriken

Die offizielle Logging-Doku nennt standardmaessig u. a.:

- `debug.log`
- `http.log`
- `query.log`
- `security.log`

Die Security-Log-Doku zeigt ausserdem, dass erfolgreiche Logins bei hohem Maschinenverkehr bewusst konfigurierbar sind und strukturierte JSON-Layouts möglich sind.

Metriken:

- Neo4j bietet Metriken und deren Export, etwa für Prometheus.
- Wichtige Betriebswerte betreffen Speicher, Query-Laufzeiten, Clusterzustand, Replikationsstatus und Ressourcenverbrauch.

6.4. Diagnosebefehle für den Alltag

Bewaehrte Erstbefehle:

```
SHOW DATABASES;
SHOW INDEXES YIELD name, type, state, populationPercent, readCount;
SHOW CONSTRAINTS;
SHOW FULLTEXT INDEXES;
SHOW VECTOR INDEXES;
SHOW TRANSACTIONS YIELD currentQueryId, currentQuery, elapsedTime, status;
```

Bei Lock-Fragen:

```
CALL dbms.listActiveLocks($queryId);
```

6.5. Query-Management und Incident-Analyse

Im Incident helfen oft nicht "noch mehr Dashboards", sondern eine klare Reihenfolge:

1. Welche Query oder welcher Dienst erzeugt Last?
2. Ist das Problem CPU, Speicher, I/O, Locking oder Cluster-Lag?
3. Hat sich das Datenmodell oder nur die Last verändert?
4. War die Query schon immer schlecht oder ist ein neuer Pfad hinzugekommen?

Gute Neo4j-Incidents werden auf Query-Ebene untersucht, nicht nur auf Host-Ebene.

6.6. Index- und Constraint-Lebenszyklus

Indizes und Constraints sind nicht "einmal anlegen und vergessen":

- neue Queries brauchen oft neue Startpunkte.
- alte Indizes können redundant werden.
- Constraints müssen mit Migrationen und Datenqualitaetsschritten zusammenspielen.

Die Doku weist darauf hin, dass `SHOW INDEXES` u. a. `lastRead`, `readCount` und weitere Spalten liefert. Das ist wertvoll für Redundanzpruefungen.

6.7. Datenimport, Migration und CDC

Operativ wichtige Wege:

- `LOAD CSV` für kontrollierte kleinere oder mittlere Importe.
- `neo4j-admin database import` für initiale Bulk-Last.
- Connectoren und CDC für Integrationslandschaften.

Wichtig:

- Imports nie nur nach Durchsatz bewerten.
- immer auch Datenqualität, Constraint-Verletzungen und spätere Query-Faehigkeit prüfen.

6.8. Typische Fehlerbilder

- Duplikate durch fehlende Constraints und schlechtes `MERGE`.
- plötzliche Langlaeufer nach Modell- oder Lastveraenderung.
- Lock Contention auf wenigen stark genutzten Knoten.
- unzureichender Page Cache mit vielen teuren Disk-Zugriffen.
- Secondaries mit zu grossem Lag für investigative Near-Real-Time-Abfragen.
- ungesicherter Backup-Port oder zu breite Admin-Rollen.

6.9. Mini-Projekt: Runbook-Skizze

Für die Zahlungsplattform sollte pro Kernthema mindestens dokumentiert sein:

- Fraud-API: Soll-Latenz, Startknoten, erlaubte Hop-Zahl.
- Import-Job: erwartete Last, Batch-Groesse, Abbruchkriterien.
- Cluster: Primaries/Secondaries je Datenbank.
- Security: Rollen, Backup-Zugriff, Query-Log-Umgang.
- Incident: erste 5 Diagnosebefehle, Eskalationskette, Restore-Pfad.

6.10. So setzt du es um - Checkliste

- Online-Backups oder passende Alternativen sind eingerichtet.
- Restore-Tests finden regelmäßig statt.
- Query-, Debug- und Security-Log sind bewusst konfiguriert.
- Kernmetriken werden zentral gesammelt.
- Es existieren Runbooks für Locking, Langlaeufer, Cluster-Ausfall und Restore.

6.11. Post-Incident-Lernen

Nach jedem Incident sollten mindestens diese Fragen beantwortet werden:

1. Was war der erste messbare Indikator?
2. Welche Query oder welches Modell war ursprünglich beteiligt?
3. Welche Metrik oder welches Log hätte früher helfen können?
4. Welche Migration, welches Constraint oder welches Runbook wird angepasst?
5. Welche fachliche Annahme war falsch?

So wird Betrieb zur Lernschleife statt zur reinen Feuerloescher-Routine.

---

7. Performance, Skalierung & Kostenkontrolle

7.1. Performance beginnt im Modell, nicht im Schraubenzieherkasten

Die größten Neo4j-Performancegewinne kommen meist aus:

- besseren Startpunkten
- saubereren Constraints
- sinnvolleren Indizes
- begrenzteren Traversal-Budgets

Hardware und Konfiguration helfen erst dann richtig, wenn das Modell operational tragfähig ist.

7.2. Query-Tuning mit `PROFILE`

Pflichtfragen pro kritischer Query:

- Wie viele DB-Hits entstehen?
- Wo startet die Query?
- Wie gross werden Zwischenmengen?
- Welche Operatoren dominieren?
- Ist die Resultatmenge klein, weil der Planner schlau ist, oder nur weil wir spät filtern?

Ein Query-Review ohne `PROFILE` ist in Neo4j meist unvollständig.

7.3. Supernodes und Traversal-Budgets

Traversal-Budgets gehören in jedes Produktionsdesign:

- maximale Hop-Zahl
- maximale erwartete Nachbarschaftsbreite

- erlaubte Beziehungstypen
- Sync vs. Async pro Query-Klasse

Dadurch werden Graphabfragen planbar, statt latent unendlich zu wirken.

7.4. Speicher- und JVM-Tuning

Die Operations-Doku empfiehlt klar:

- `server.memory.heap.initial_size` und `server.memory.heap.max_size` bewusst zu setzen, idealerweise identisch.
- `server.memory.pagecache.size` explizit zu konfigurieren.
- `neo4j-admin server memory-recommendation` als Startpunkt zu verwenden.

Praxisregeln:

- Heap für Query-Komplexitaet und Parallelitaet dimensionieren.
- Page Cache für heisse Daten und native Indizes dimensionieren.
- grosse Listen, breite `UNWIND`s und schwere Pfadexpansion als Memory-Risiko behandeln.

7.5. Import- und Batch-Strategien

Batcharbeit beeinflusst Kosten und Stabilität stark:

- grosse Einzeltransaktionen sind riskant.
- besser kontrollierte Batches mit klarer Idempotenz.
- Bulk-Importe ausserhalb der heissen Betriebsfenster.
- nach Importen Constraint- und Query-Validierung durchfuehren.

7.6. Skalierungsmuster

Neo4j skaliert typischerweise über:

- bessere Query-Startpunkte
- mehr Speicher für heisse Daten
- Read-Scaling über Secondaries
- Trennung von OLTP und schweren Analysen
- ggf. Aufteilung auf mehrere Datenbanken oder spezialisierte Architekturen

Weniger gut skaliert es über:

- unkontrolliertes Erhöhen von Primaries
- breite Mehrhop-Queries ohne Budget
- analytische Batcharbeit auf demselben operativen Graph ohne Isolation

7.7. Kostenkontrolle bei Full-Text und Vector

Full-Text- und Vector-Funktionen sind wertvoll, aber nicht kostenlos:

- Lucene- und Vektorindizes belegen Speicher und Platte.
- Vector-Indizes haben eigene Konfigurations- und Aktualisierungskosten.
- Quantisierung kann Speicherbedarf senken, aber Genauigkeit beeinflussen.

Praxisfrage:

- Braucht wirklich jede Entitaet Embeddings, oder nur ausgewählte Knoten?

7.8. Mini-Projekt: Tuning-Loop

1. Kernqueries für Fraud-Screening und Ermittler-Suche messen.
2. Startpunkte und Indizes prüfen.
3. Supernode-Risiken bewerten.
4. Heap/Page Cache gegen echte Last justieren.
5. Cluster-Routing für Reads validieren.
6. Erneut auf P95/P99, Lag und Ressourcenverbrauch messen.

7.9. So setzt du es um - Checkliste

- Alle Kernqueries haben `PROFILE`-Nachweise.
- Heap und Page Cache sind explizit gesetzt.
- Traversal-Budgets sind dokumentiert.
- Read-Scaling über Secondaries ist begründet und gemessen.
- Full-Text- und Vector-Indizes sind fachlich gerechtfertigt.

7.10. Benchmarking, das wirklich Aussagekraft hat

Ein belastbarer Neo4j-Benchmark bildet mindestens ab:

- reale Knotengrade
- echte Hotspots
- echte Query-Mischung
- reale Parallelitaet
- Cluster- oder Routing-Verhalten

Mittelwerte allein sind kaum hilfreich. Wichtig sind:

- P95/P99
- Reaktion auf Lastspitzen
- Verhalten nach Warmup des Page Cache
- Erholung nach Batch- oder Importphasen

7.11. OLTP-Graph sauber von Analytics und GDS trennen

Viele Teams ueberladen den operativen Graph mit analytischen Erwartungen. Besser ist:

- Operativer Graph für schnelle, lokale Fachfragen.
- Schwere Community-, Zentralitaets- oder Batch-Analysen separat planen.
- GDS-Workloads nicht stillschweigend in denselben heissen Pfad legen.

So bleibt der Kern stabil, während Analytics trotzdem möglich ist.

7.12. Release-Gates für Graphsysteme

Ein sinnvoller Release-Gate-Satz für Neo4j:

- keine Kernquery verliert ihren selektiven Startpunkt.
- P99 verschlechtert sich nicht über definierten Schwellwert.

- keine neuen Constraint-Verletzungen.
- kein unerklaerter Anstieg in Lock-Wartezeiten.
- Cluster-Lag und Backup-Status bleiben im Soll.

So wird Graphstabilitaet Teil des Softwareprozesses und nicht nur eine Plattform-Sorge.

---

8. Sicherheit, Governance & Compliance

8.1. Sicherheitsgrundsatz

Ein produktiver Graph ist fast immer sensibel, weil Beziehungen selbst vertraulich sein können. In Fraud-, Personen-, Lieferketten- oder Wissensgraphen sagt oft schon die Existenz einer Beziehung viel aus. Sicherheit muss deshalb nicht nur Knotenwerte, sondern auch Zugriff auf Pfade und Teilgraphen absichern.

8.2. Authentifizierung und Rollen

Neo4j bietet Role-Based Access Control:

- built-in roles
- benutzerdefinierte Rollen
- Privilegien auf DBMS- und Datenbankebene

Die Doku verweist ausserdem auf:

- LDAP-Integration
- Single Sign-On-Integration
- Property-based und attribute-based access control

Praxisregeln:

- keine gemeinsam genutzten Admin-Konten für Anwendungen.
- technische Nutzer pro Dienst.
- Least Privilege für Query-, Import- und Admin-Pfade.

8.3. Teilgraph-Zugriff und feingranulare Rechte

In vielen Graphsystemen ist nicht nur der Datensatz sensibel, sondern die Struktur. Deshalb sind feingranulare Leserechte besonders wichtig:

- welche Labels dürfen gelesen werden?
- welche Beziehungstypen dürfen gesehen werden?
- welche Properties bleiben verborgen?

Gerade in Fraud- oder HR-Szenarien ist das oft wichtiger als reine Tabellenrechte.

8.4. TLS, Ports und Backup-Haertung

Wichtige Punkte:

- Bolt/HTTP und Cluster-Kommunikation verschluesseln.
- Backup-Port hart absichern.

Die Backup-Doku warnt explizit davor, den unter `server.backup.listen_address` konfigurierten Port extern offen zu lassen, weil sonst unautorisierte Kopien der Datenbank möglich sein können.

8.5. Logs, Auditierbarkeit und Datenschutz

Sinnvolle Mindestbausteine:

- Query-Log bewusst konfigurieren.
- Security-Log aktiv auswerten.
- erfolgreiche Authentifizierungen bei hohem Maschinenverkehr gezielt handhaben.
- sensible Query-Parameter und personenbezogene Daten im Logging ueberdenken.

Beziehungen selbst können personenbezogen oder hochsensibel sein. Deshalb ist Governance bei Graphen oft anspruchsvoller als bei isolierten Tabellen.

8.6. Plugins, APOC und Erweiterungen

Erweiterungen sind nützlich, aber sicherheitsrelevant:

- nur benötigte Plugins aktivieren.
- Prozedur- und Funktionsrechte einschränken.
- APOC und andere Erweiterungen nicht als stillen Admin-Hintereingang behandeln.

Ein "wir aktivieren erstmal alles, was praktisch wirkt" ist in Produktivumgebungen kein gutes Sicherheitsmodell.

8.7. Governance durch Migrations- und Schema-Vertraege

Governance in Neo4j bedeutet nicht nur Rechtevergabe, sondern auch:

- versionierte Schemata
- reviewte Label- und Relationship-Typen
- dokumentierte Constraints
- klare Benennungsregeln
- nachvollziehbare Datenherkunft

So bleibt der Graph auch nach Monaten noch erklärbar.

8.8. So setzt du es um - Checkliste

- Rollenmodell folgt Least Privilege.
- Backup-Port und Admin-Pfade sind gesichert.
- Query-, Debug- und Security-Logs sind bewusst konfiguriert.
- Plugins und Prozeduren sind auf Notwendiges reduziert.
- Schemamigrationen laufen versioniert und reviewt.

8.9. Rollenrezertifizierung und Drift-Kontrolle

Wie bei anderen Plattformen entsteht das Risiko oft nicht am ersten Tag, sondern durch schleichende Rechteausweitung. Bewaehrte Praxis:

- quartalsweise Rollenreview
- technische Nutzer gegen aktuelle Service-Landschaft prüfen

- ueberprivilegierte Konten systematisch abbauen

8.10. Security-Drills

Sinnvolle Szenarien:

- kompromittiertes technisches Konto
- ungewollt offener Backup-Port
- ungeplante Plugin-Aktivierung
- Query- oder Security-Log enthält sensible Inhalte

Ziel:

- Erkennungszeit messen
- Reaktionskette testen
- Auditfaehigkeit prüfen

8.11. Governance-Metriken

Sinnvolle Kennzahlen:

- Anteil technischer Konten mit dokumentierter Rolle
- Anzahl ausstehender Rollenrezertifizierungen
- Anzahl kritischer Prozeduren mit enger Rechtevergabe
- Restore-Test-Quote
- Anteil produktiver Kernqueries mit dokumentiertem Constraint- und Indexpfad

---

9. Weiterführende Links + Uebungsaufgaben

9.1. Offizielle Kernquellen (Neo4j)

Dokumentations-Start:

- Operations Manual
- Cypher Manual

Architektur und Cluster:

- Clustering Introduction
- Concurrent Data Access

Cypher, Schema und Tuning:

- MERGE
- Constraints
- Index Impact / Query Performance
- Full-Text Indexes
- Vector Indexes

Betrieb:

- Import

- Online Backup
- Memory Configuration
- Logging
- Metrics
- Authentication and Authorization

Weiteres Oekosystem:

- GraphAcademy
- Graph Data Science Docs

9.2. Uebungsaufgaben (steigende Schwierigkeit)

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

- Modelliere `Customer`, `Card`, `Device`, `Merchant`, `Transaction`.
- Lege Eindeutigkeits-Constraints für alle kanonischen Identitäten an.
- Importiere Testdaten und prüfe, ob `MERGE` idempotent bleibt.

2. **Cypher-Praxis (leicht bis mittel)**

- Formuliere drei Fraud-Fragen als Cypher-Queries.
- Führe `EXPLAIN` und `PROFILE` aus.
- Dokumentiere Startpunkte und DB-Hits.

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

- Erzeuge bewusst einen Hotspot, z. B. ein vielfach genutztes `Device`.
- Miss Query- und Lock-Verhalten.
- Entlaste das Modell durch Segmentierung oder Query-Budgetierung.

4. **Cluster-Drill (mittel)**

- Simuliere Ausfall einer Primary im 3-Primary-Setup.
- Beobachte Write-Verhalten, Routing und Read-Pfade.
- Dokumentiere Runbook und Recovery.

5. **Vector-Search-Integration (mittel)**

- Lege Embeddings für einen Teilgraphen an.
- Erzeuge einen Vector-Index.
- Kombiniere semantische Ähnlichkeit mit einer Graph-Traversal-Regel.

6. **Locking und Transaktionsgrenzen (schwer)**

- Simuliere parallele Updates auf dieselben Identitäten.
- Analysiere mit `SHOW TRANSACTIONS` und `dbms.listActiveLocks`.
- Reduziere Konflikte durch Transaktionsschnitt und Modellanpassung.

7. **Security Hardening (schwer)**

- Haerte Rollen, Prozedurzugriffe und Backup-Port.
- Aktiviere sinnvolles Logging.
- Führe einen Restore-Test mit Sicherheitsreview durch.

9.3. Abnahmekriterien für "produktionsreif"

- Kernqueries haben stabile, selektive Startpunkte.
- Kanonische Entitaeten sind durch Constraints abgesichert.
- P95/P99-Ziele sind unter realistischer Last nachgewiesen.
- Restore-Tests sind erfolgreich durchgefuehrt.
- Rollen, Logs, Ports und Erweiterungen sind gehaertet.
- Cluster- und Lock-Runbooks sind vorhanden und getestet.

9.4. Abschluss: Was "Pro-Level Neo4j" in der Praxis bedeutet

Pro-Level Neo4j heisst nicht, möglichst komplizierte Pfade zu schreiben. Pro-Level heisst:

- du weisst genau, warum ein Graph hier fachlich passt.
- du modellierst Identitäten und Beziehungen bewusst.
- du setzt Constraints vor Produktivlast.
- du liest `PROFILE` nicht nur, sondern nutzt es für echte Entscheidungen.
- du betreibst Cluster, Backups, Logs und Security mit derselben Sorgfalt wie das Datenmodell.

Wenn diese Punkte stehen, wird Neo4j vom spannenden Spezialwerkzeug zu einer belastbaren Produktionskomponente.

9.5. So setzt du es um - Checkliste

- Offizielle Primaerquellen sind im Team-Wiki verlinkt.
- Kernqueries sind mit `PROFILE` dokumentiert.
- Migrations- und Constraint-Konzept sind vereinbart.
- Restore- und Failure-Drills finden regelmäßig statt.
- Security- und Governance-Metriken werden aktiv verfolgt.

9.6. 90-Tage-Umsetzungsfahrplan

Tage 1-30

- Fachliche Kernfragen finalisieren.
- Modell und Constraints festziehen.
- erste produktionsnahe Query-Baseline aufbauen.

Tage 31-60

- Backups, Restore-Tests, Logs und Metriken operationalisieren.
- Cluster- und Routing-Entscheidungen validieren.
- Security-Rollen und Plugin-Policy haerten.

Tage 61-90

- Failure- und Lock-Drills fahren.
- Performance-Gates in Releases integrieren.
- Kostenkontrolle für Indizes, Speicher und sekundäre Workloads etablieren.

9.7. Self-Assessment für Teams

Bewertet euch ehrlich mit 0 bis 2 Punkten je Aussage:

- Wir haben klare fachliche Graphfragen und dokumentierte Traversals.
- Kanonische Entitaeten sind durch Constraints abgesichert.
- Unsere Kernqueries haben `PROFILE`-Nachweise.
- Wir kennen unsere Hotspots und Supernode-Risiken.
- Restore-Tests laufen regelmäßig.
- Cluster-, Routing- und Retry-Regeln sind fachlich abgestimmt.
- Security- und Plugin-Zugriffe folgen Least Privilege.

Auswertung:

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

9.8. Nächste Ausbaustufen

Wenn die Basis stabil ist, sind sinnvolle nächste Schritte:

- Query-Gates und `PROFILE`-Reviews in Pull Requests verankern.
- Full-Text- und Vector-Nutzung an klare Fachfaelle koppeln.
- operative und analytische Graphpfade sauber trennen.
- quartalsweise Graph-Health-Reviews durchfuehren: Welche Knotenklassen wachsen am staerksten? Wo entstehen neue Hotspots? Welche Indizes werden nie gelesen? Welche Beziehungen sind fachlich tot? Welche Security-Regeln greifen real?

Genau so bleibt ein Neo4j-System über Jahre wartbar: nicht als "besonderer Spezialgraph", sondern als aktiv gesteuerte Datenplattform mit klaren Verträgen zwischen Fachseite, Entwicklung und Betrieb.

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/neo4j/