

INOQ

Datenbanken

Datenmodelle, Skalierung,
Cloud

Inklusive
neuem Kapitel:

**Vektordaten-
banken**

Lucas Dohmen • Christine Koppelt
Robert Glaser • Marco Steinke

Datenbanken

Konzepte, Betrieb, Auswahl

**Lucas Dohmen
Christine Koppelt
Robert Glaser
Marco Steinke**

innoQ Deutschland GmbH
Krischerstraße 100 · 40789 Monheim am Rhein · Germany
Phone +49 2173 33660 · www.INNOQ.com

Layout: Tammo van Lessen with X₃L^AT_EX
Design: Murat Akgöz
Typesetting: André Deuerling

Datenbanken – Konzepte, Betrieb, Auswahl

Published by innoQ Deutschland GmbH
2. Auflage · Dezember 2024

Copyright © 2024 Christine Koppelt, Lucas Dohmen, Marco Steinke,
Robert Glaser

Inhaltsverzeichnis

1	Motivation	3
2	Datenmodelle	5
2.1	Aggregat-orientiert	5
2.2	SQL & NewSQL	7
2.3	Graphen	8
2.4	Datenstrukturen	9
2.5	Zeitreihen	10
2.6	Vektoren	10
2.7	Zusammenfassung	11
3	Skalierung	13
3.1	Was wollen wir skalieren?	14
3.2	Sharding	15
3.3	Replikation	16
3.4	Kombination von Sharding und Replikation	20
3.5	Skalierung von Vektordatenbanken	21
3.6	Verfügbarkeit	22
3.7	Konfliktentdeckung und -behandlung	24
3.8	Zusammenfassung	25
4	Betriebsaspekte	27
4.1	Trends	27
4.2	Database Reliability Engineering	28
4.3	Virtuelle Maschine, Managed Service oder Kubernetes? ..	33
4.4	Backup, Restore & Recovery	39
5	Beispiele	41
5.1	Relationale Datenbank: PostgreSQL	41
5.2	Datenstruktur Server: Redis	44
5.3	Zeitreihendatenbank: InfluxDB	46

5.4	Graphdatenbank: Neo4j	49
5.5	NewSQL Datenbank: CockroachDB.....	52
5.6	Vektordatenbanken: Moderne Suche für semantische Ähnlichkeiten.....	54
6	Auswahl einer Datenbank	59
7	Unser Angebot	63
8	Quellen	65
	Über die Autor*innen	67

Notizen

1 Motivation

Für die meisten Anwendungen ist die zuverlässige Speicherung und Bereitstellung von Daten eine zentrale Komponente. **Die Daten einer Anwendung leben oft länger als der Anwendungs Quellcode.** Quellcode lässt sich vergleichsweise einfach erweitern, refaktorisieren oder auf eine neue Frameworkversion anpassen, Bugfixes lassen sich meist relativ einfach deployen. Fehlerhafte oder verlorene Daten lassen sich dagegen nur schwer wiederherstellen. Änderungen an den Strukturen eines produktiven Datenbestands sind oftmals zeitintensiv und erfordern sorgfältige Planung. **Fehler in der Datenmodellierung werden uns entsprechend länger verfolgen.** Auch der Wechsel einer Datenbank ist teuer, zeitintensiv und manchmal verlustbehaftet. Ein Wechsel von MySQL zu PostgreSQL ist aufwendig, weil alle datenbankspezifischen Funktionen gefunden und ersetzt werden müssen. Ein Wechsel von MongoDB auf MySQL noch aufwendiger, weil es einen kompletten Wechsel des Datenmodells und des Schemamanagements erfordert. Bei der Auswahl eines Datenmodells und einer Datenbank sollte also entsprechend sorgfältig vorgegangen werden, um unliebsame Überraschungen zu vermeiden.

Das Angebot an Datenbanken und anderen Speichermöglichkeiten ist in den letzten Jahren immer weiter ausdifferenziert: Mittlerweile gibt es für viele Anwendungsfälle spezialisierte Angebote mit ganz unterschiedlichen Eigenschaften hinsichtlich Konsistenz, Datenmodell und Zugriffszeiten: Zeitreihendatenbanken für die schnelle Speicherung großer Mengen von Sensordaten, Key-Value Stores als In-Memory Datenbanken für extrem kurze Zugriffszeiten auf einfache Datenstrukturen, NewSQL-Datenbanken für Skalierbarkeit bei hohen Konsistenzgarantien, Object Stores für die Ablage großer Mengen an Binärdaten etc. Was die Wahl nicht einfacher macht.

Gleichzeitig haben sich Anwendungsarchitekturen, Organisationsmodelle und Betriebskonzepte, bedingt durch größere und komplexere Anwendungen, höhere Anforderungen an Skalierbarkeit und Liefergeschwindigkeit in den letzten Jahren stark gewandelt:

- Statt großer monolithischer Systeme werden zunehmend verteilte Systeme auf Basis von **Microservice Architekturen** oder Self-Contained-Systems gebaut, die jeweils ihre eigene Datenhaltung verantworten.
- Statt horizontalen, nach Funktion getrennten Organisationsstrukturen - wie Betrieb, Datenbankadministratoren, Backend-Entwicklern, Frontend-Entwicklern und Testern - geht die Entwicklung in Richtung **vertikaler Teams**, die entlang fachlicher Problemdomänen geschnitten sind und unterschiedliche Kompetenzen beinhalten.
- Statt individuell gemanagter virtueller Maschinen kommen **Cloudangebote** und Cluster Plattformen wie **Kubernetes** zum Einsatz. Methodiken wie **DevOps** und **Site Reliability Engineering** zielen auf einen höheren Grad an Automatisierung und ein besseres Fehlermanagement.

Daraus ergeben sich auch neue Herausforderungen für die Auswahl passender Datenbanken. In diesem Primer möchten wir einige Aspekte herausgreifen, die wir dabei für bedenkenswert halten. Die Themen reichen dabei von der Vorstellung unterschiedlicher **Datenmodelle**, über Aspekte der Skalierung wie **Replikation** und **Sharding**, Methodiken und Überlegungen für den **Betrieb in der Cloud und Kubernetes**, bis hin zur **Vorstellung von Datenbanken und für sie geeignete Einsatzbereiche**. Dabei beschränken wir uns aus Platzgründen auf Themen aus dem Bereich *Data at Rest* also auf die Art und Weise wie Daten gespeichert werden. Dem Thema *Data in Transit*, also wie Daten zwischen verschiedenen Systemen zuverlässig ausgetauscht und ggf. auch transformiert werden können, beispielsweise mittels Distributed Queues wie Kafka, RabbitMQ oder AWS SQS, widmen wir vielleicht noch einen eigenen Primer.

2 Datenmodelle

Das Datenmodell einer Datenbank ist das Interaktionsmodell durch welches wir die Daten sehen und manipulieren können. Als solches ist es unabhängig von der Art und Weise, in der die Daten gespeichert werden. Es handelt sich um die Benutzersicht und beeinflusst zum Beispiel die Abfragesprache.

Das relationale Datenmodell ist bis heute der Standard. Aber welche alternativen Datenmodelle gibt es? Und warum sind sie einen Blick wert?

2.1 Aggregat-orientiert

Es gibt zwei Motivationen Aggregat-orientierte Datenbanken gegenüber relationalen Datenbanken zu bevorzugen:

- **Skalierbarkeit:** Relationale Datenbanken sind traditionell schwer im Cluster zu betreiben, da sie für diesen Use Case nicht geschaffen wurden. Durch Features wie ACID Garantien, Transaktionen und Joins haben sie zudem eine eher langsame Lese- und Schreibgeschwindigkeit.
- **Impedance Mismatch:** Objektgraphen und Datenstrukturen müssen stets auf eine tabellarische Form gebracht werden.

Aggregat-orientierte Datenbanken bieten hier eine Alternative: Sie bieten die Möglichkeit, einen Objektgraphen (oder auch Aggregat) in einem Dokument zu speichern. Der einfachste Weg sich diese Aggregate vorzustellen ist in der Form eines JSON Dokuments.

Ein Blog Post könnte beispielsweise so aussehen:

```
{
  title: "Aggregate Oriented Databases",
  text: "...",
  tags: ["nosql"],
  comments: [
    { author: "Jane Doe", text: "I like those databases" },
    { author: "John Doe", text: "I hate those databases" }
  ]
}
```

Hier befinden sich die Daten, die in einer relationalen Datenbank üblicherweise über mehrere Tabellen verteilt würden in einem Dokument. Dieses Dokument kann als Ganzes abgefragt werden: Wollen wir nun also den Blogpost inklusive der Kommentare rendern brauchen wir nur ein Dokument abzufragen. Auf diese Art und Weise wird auch die Notwendigkeit von Joins kleiner: In dem Fall des Blogposts brauchen wir keinen Join, alle Daten sind im Dokument enthalten. Aus diesem Grund bieten viele Datenbanken dieser Kategorie keine Joins an oder haben sie erst sehr spät als Feature hinzugefügt. Viele dieser Datenbanken bieten auch keine (Cross-Collection) Transaktionen an, sondern beschränken ihre Konsistenz Garantien auf Änderungen an einzelnen Dokumenten.

Die einfachste Form einer solchen Datenbank wäre ein *Key/Value Store* wie memcached oder Riak CS. In dieser Modellierung ist der Schlüssel ein String, unter dem wir einen beliebigen Wert einfach als String oder Binärdaten ablegen. Sie geben keine Struktur (wie JSON) für diese Daten vor, und bieten folglich auch keine Möglichkeit Indexe auf diesen Daten anzulegen oder nur bestimmte Felder zurückzugeben (Projektion).

Eine erweiterte Form davon sind die sogenannten *Document Stores* wie CouchDB, MongoDB oder Riak. Hier legen wir strukturierte Dokumente (meist als JSON wie oben skizziert) unter einem Primärschlüssel ab. Zudem können wir wie bei einer relationalen Datenbank weitere Indexe hinzufügen. Bei der Abfrage können auch nur bestimmte Felder aus den Aggregaten zurückgegeben werden. Oftmals steht uns auch die Möglichkeit zur Verfügung, Materialized Views zu definieren. Eine Variante davon sind die sogenannten *Suchindexe* wie Elasticsearch oder Solr. Sie

sind den Document Stores sehr ähnlich, legen aber einen stärkeren Fokus auf die Volltextsuche und bieten weniger Konsistenzgarantien. Auch Document Stores wie MongoDB bieten eine Volltextsuche, wodurch die Grenzen zwischen diesen beiden Kategorien unscharf sind.

Auch wenn sich diese Datenbanken in der Verwendung unterscheiden, haben sie folgende Eigenschaften in der Modellierung gemeinsam:

1. Daten werden *denormalisiert* abgelegt.
2. Wir speichern Daten als *Aggregate*.
3. Joins versuchen wir dadurch zu vermeiden – falls nötig führen wir unsere Joins auf Applikationsseite durch.
4. ACID Garantien beziehen sich in den meisten dieser Datenbanken nur auf ein Dokument.

Durch diese Modellierung können die Daten leichter auf mehrere Knoten verteilt werden (Sharding wie im nächsten Kapitel beschrieben):

- Aggregat-orientierte Datenbanken sind oftmals für den Betrieb im Cluster konzipiert
- Sie bieten weniger Garantien und Features und eine höhere Anzahl von Lese- und Schreibzugriffen

2.2 SQL & NewSQL

Wie schon beschrieben bieten klassische SQL-Datenbanken wie PostgreSQL, MySQL, MariaDB, Oracle oder MSSQL eine mächtigere Abfragesprache und hohe Konsistenzgarantien über Tabellengrenzen. Auch wenn diese Eigenschaften die Skalierbarkeit einschränken, adressieren mittlerweile Datenbanken wie PostgreSQL die flexiblere Datenmodellierung: So ist es beispielsweise möglich, ein Feld vom Typ JSON anzulegen. Da in SQL eine Abfrage dieser Art nicht vorgesehen ist, muss es entsprechend erweitert werden. Das macht es weniger komfortabel als im Falle einer Abfragesprache, bei der dies von Anfang an vorgesehen war. Zudem muss die Datenbank Indexe innerhalb dieser JSON Felder unterstützen. Auch Feature wie Volltextsuchen oder Geo Indexe sind in

PostgreSQL zu finden. Eine ausführlichere Beschreibung von PostgreSQL findet sich im Kapitel 5.1.

Neben der Weiterentwicklung von existierenden SQL-Datenbanken gibt es auch Neuentwicklungen. Diese auch als NewSQL bezeichneten Datenbanken versuchen neben der flexibleren Modellierung auch die Möglichkeit im Cluster zu laufen zu verbessern. Dabei unterscheiden wir zwischen Datenbanken, die wir selber betreiben wie CockroachDB und Datenbanken, die uns als Software-as-a-Service Angebot bereitgestellt werden wie Azure Cosmos DB, Amazon Aurora und Google Spanner. Im Kapitel 4.3 gehen wir auf diesen Trade-Off im Detail ein. In beiden Fällen ist es einfach, weitere Knoten zum Cluster hinzuzufügen. Um den Umstieg zu erleichtern, bieten diese Datenbanken nach außen hin die API einer verbreiteten relationalen Datenbank (PostgreSQL oder MySQL/MariaDB).

2.3 Graphen

In einer Graphdatenbank modellieren wir unsere Daten als Knoten und Kanten. Der bekannteste Vertreter dieser Kategorie ist Neo4j. Wollen wir zum Beispiel ein Netzwerk modellieren so können wir die Maschinen als Knoten modellieren und die Netzwerkverbindungen als Kanten. Sowohl Knoten als auch Kanten können mit Attributen versehen werden. So könnten wir beispielsweise die jeweilige maximale Bandbreite als Eigenschaft an unsere Kanten schreiben. Mit Hilfe einer Graphdatenbank können wir so zum Beispiel ermitteln, welches die kürzeste Verbindung zwischen zwei Maschinen ist und wie viel Bandbreite dort zur Verfügung steht. Wir könnten auch den Ausfall von Verbindungen simulieren.

Graphdatenbanken bieten nicht dieselben Möglichkeiten der Skalierung wie Aggregat-orientierte Datenbanken: Sind die Knoten auf mehrere Maschinen verteilt, kann das Traversieren des Graphens bedeuten, dass wir das Netzwerk überwinden müssen. Das kann sogar mehrfach während einer Query passieren. Die meisten Graphdatenbanken bieten folglich kein Sharding an. Gerade bei Analysen und Auswertungen spielen sie jedoch ihre Stärke aus: Neo4j bietet beispielsweise ein praktisches Web Interface an, mit dem Queries ausprobiert und direkt visualisiert werden können.

Manche Datenbanken versuchen die Vorzüge von Aggregat-orientierten Datenbanken mit denen der Graphdatenbank zu kombinieren (beispielsweise ArangoDB). In diesen Datenbanken sind sowohl die Knoten als auch die Kanten Aggregate. Das ermöglicht eine flexible Modellierung.

Eine Beschreibung von Neo4j als Beispiel für eine Graphdatenbank findet sich in Kapitel 5.4.

2.4 Datenstrukturen

Natürlich handelt es sich auch bei Tabellen oder Graphen um Datenstrukturen. Ein Datenstruktur Server jedoch erlaubt es, die grundlegenden Datenstrukturen (Listen, Mengen...) abzulegen und abzufragen. Beispiel für eine solche Datenstruktur wäre beispielsweise eine Liste oder eine Menge. Auf diesen Datenstrukturen können dann verschiedene Operationen ausgeführt werden, wie zum Beispiel: Füge ein neues Element am Anfang der Liste hinzu oder finde alle Elemente der Liste, die einem bestimmten Kriterium entsprechen.

Die Modellierung ist dabei anders als bei einer Aggregat-orientierten Datenbank: Ein Aggregat wird auf verschiedene Datenstrukturen aufgeteilt. Bei unserem Blog Post von oben könnten beispielsweise der Titel und Text in einem Hash liegen, die Tags in einer Liste und die Kommentare ebenfalls in einer Liste. Wollen wir also alle Informationen abfragen, um unseren Blog Post anzuzeigen, brauchen wir mehrere Abfragen. Durch die Datenstrukturen sind dafür aber komplexere Operationen direkt in der Datenbank möglich.

Redis ist das prominenteste Beispiel von Datenbanken dieser Kategorie. Durch die Vielzahl von Datenstrukturen bietet es sich als Universalkleber für verteilte Systeme an: Es kann beispielsweise als Job Queue verwendet werden, für Publish/Subscribe, als Geo Index oder als Cache. Redis ist aber auch als primäre Datenbank einsetzbar. Eine umfangreichere Beschreibung von Redis findet sich in Kapitel 5.2.

2.5 Zeitreihen

In einer Zeitreihendatenbank ist der Primärschlüssel stets ein Zeitpunkt. Legen wir beispielsweise die Antwortgeschwindigkeiten unserer Web Anwendung in einer Zeitreihendatenbank ab, können wir mit den Aggregationsmöglichkeiten beispielsweise die durchschnittliche Antwortgeschwindigkeit pro Stunde ermitteln.

Populäre Vertreter dieser Kategorie sind InfluxDB, Prometheus und M3. ElasticSearch bietet ebenfalls eine eingebaute Zeitreihendatenbank. Alle diese Datenbanken werden hauptsächlich für Metriken und Monitoring verwendet. Jede ist dafür in eine Suite von Tools für diesen Zweck eingebunden (InfluxDB im Tick Stack, M3 in M3DB im Uber Stack – Prometheus bietet alle Features in einem Paket). Zeitreihendatenbanken können aber auch als primäre Datenbank eingesetzt werden wenn wir beispielsweise Sensordaten sammeln.

Eine Beschreibung von InfluxDB als Beispiel für eine Zeitreihendatenbank findet sich in Kapitel 5.3.

2.6 Vektoren

Eine vergleichsweise neue Kategorie sind Vektordatenbanken. Anders als traditionelle Datenbanken, die mit exakten Übereinstimmungen arbeiten, speichern sie Daten als numerische Vektoren in einem hochdimensionalen Raum. Diese Vektoren repräsentieren die semantische Bedeutung der Daten, wodurch Ähnlichkeitssuchen möglich werden.

Der Prozess läuft dabei in zwei Schritten ab: Zunächst werden die Eingabedaten (Texte, Bilder etc.) mittels KI-Modellen in Vektoren umgewandelt – ein Vorgang, der als “Embedding” bezeichnet wird. Diese Vektoren werden dann in der Datenbank gespeichert. Bei einer Suchanfrage wird diese ebenfalls in einen Vektor umgewandelt und die Datenbank findet die ähnlichsten gespeicherten Vektoren mittels spezieller Algorithmen wie Approximate Nearest Neighbor (ANN) oder der Kosinus-Ähnlichkeit.

Ein Vektor für den Satz “Der Hund spielt im Garten” könnte beispielsweise so aussehen:

```
[0.234, -0.089, 0.876, ..., 0.543]
```

Dieser Ansatz ermöglicht es, Suchanfragen auch dann erfolgreich zu beantworten, wenn sie nicht exakt mit den gespeicherten Daten übereinstimmen. So könnte eine Suche nach “Haustier im Freien” das obige Beispiel finden, weil die semantische Bedeutung ähnlich ist. Dies macht Vektordatenbanken besonders interessant für:

- Semantische Dokumentensuche
- Empfehlungssysteme
- Ähnlichkeitssuche für Produkte
- Bildsuche
- KI-getriebene Features und Chatbots

2.7 Zusammenfassung

In diesem Kapitel haben wir die wichtigsten Datenmodelle und ihre konzeptionellen Unterschiede vorgestellt. Die folgende Grafik liefert einen graphischen Überblick über die unterschiedlichen Datenmodelle, konzeptionellen Überschneidungen und konkrete Datenbankbeispiele.

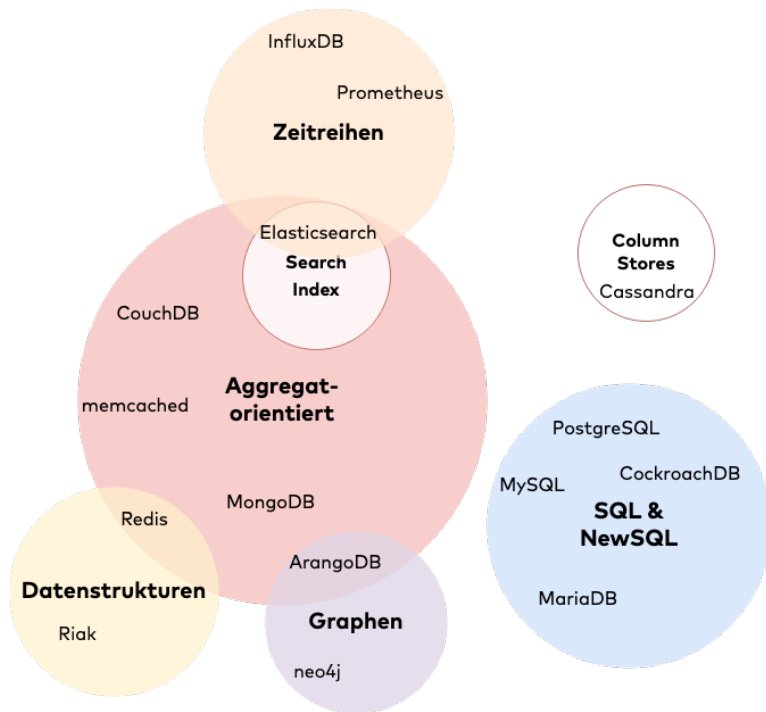


Abbildung 2.1: Datenmodelle

3 Skalierung

Für Jahrzehnte waren **relationale Datenbanken**, die SQL als ihre Abfragesprache benutzen, als populärste Lösung etabliert. Alternativen wie objektorientierte Datenbanken konnten sich nicht durchsetzen. Diese Datenbanken wurden üblicherweise als Einzelinstanzserver betrieben. Mit dem Erfolg von weltweit aufrufbaren **Webanwendungen** änderten sich jedoch die Anforderungen an Datenbanken: Die Anzahl an gleichzeitigen Zugriffen sowie deren Frequenz erhöhte sich drastisch und auch die Datenmengen wuchsen immens. Die klassischen relationalen Datenbanken waren jedoch nicht auf verteilte Zugriffe und verteilte Datenmengen ausgelegt. Probleme, die gerade die großen Internet-Unternehmen wie Amazon, Google und Facebook hatten.

Um diese zu adressieren entwickelte Google BigTable und MapReduce, Amazon entwickelte DynamoDB (alle 2004). Sie legten damit den Grundstein für die sogenannten **NoSQL**-Datenbanken, welche eine horizontale Skalierung datenhaltender Systeme ermöglichen. Da diese Systeme zuerst nur intern zur Verfügung standen und in wissenschaftlichen Artikeln beschrieben wurden, begannen andere Unternehmen ähnlicher Größe diese Ideen für ihre eigenen Datenbanken zu verwenden wie zum Beispiel 2008 Facebook mit Cassandra und 2009 LinkedIn mit Voldemort. Auch kleinere Unternehmen wie Basho (Riak in 2009), sowie einige Open Source Projekte (HBase in 2006) implementierten diese Konzepte. Bis heute dient das Datenmodell von BigTable (Wide Column Stores), die Abfrage über MapReduce, sowie die Techniken aus dem Dynamo Paper ¹ (Consistent Hashing, Vector Clocks, Quorums, Merkle Trees und Gossip-basierte Membership Protokolle) vielen Datenbanken als Inspiration. Zur gleichen Zeit (2005) begann Damien Katz mit seiner von Lotus Notes inspirierten Arbeit an CouchDB. 2009 startete zudem die Arbeit an MongoDB und Redis.

Über die Datenmodelle haben wir bereits gesprochen. In diesem Kapitel wollen wir über die **Skalierbarkeitsprobleme** sprechen, die diese Datenbanken adressieren.

¹<https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>

3.1 Was wollen wir skalieren?

Bei Datenbanken gibt es verschiedene Aspekte, die wir skalieren möchten:

- Lesezugriffe
- Schreibzugriffe
- Datenmengen

Ein weiterer Aspekt, den wir adressieren möchten, ist der Ausfall von Maschinen und wie wir dabei Datenverluste verhindern können.

In diesem Kapitel verwenden wir den Begriff Knoten wenn wir von einer Instanz unserer Datenbank sprechen. Dabei kann es sich um eine physikalische Maschine oder aber auch einen Docker Container handeln. Wenn wir von Konsistenz sprechen, dann benutzen wir hier einen intuitiven Konsistenzbegriff: Wenn ich Daten schreibe, dann kann ich und jeder andere sie ab diesem Zeitpunkt lesen. Unsere Datenbank mit all ihren Knoten wirkt also nach außen wie eine Einheit. Wir nehmen zudem vorerst an, dass wir ohne Zweifel feststellen können, ob ein Knoten erreichbar ist. Wenn er nicht mehr erreichbar ist kommt er auch nicht mehr zurück. Unsere Netzwerkverbindungen sind stabil: Entweder sie funktionieren oder sie funktionieren nicht.

Ähnlich wie bei einer Web Anwendung können wir auch für Datenbanken den Shared Nothing² Ansatz verwenden. Wir könnten von unserer Datenbank weitere Instanzen hochfahren zum Beispiel pro:

- Mandant in einer mandantenfähigen Anwendung
- Use Case

Eine weitere Möglichkeit, die uns dieser Ansatz ermöglicht ist, für verschiedene Daten jeweils eine andere Datenbank zu verwenden. Wir könnten beispielsweise die Warenkörbe in DynamoDB und die Bezahlvorgänge in PostgreSQL ablegen.

²Shared Nothing: Ein Architekturpattern für ein verteiltes System bei dem jeder Knoten eigenständig arbeitet. Eine ausführliche Beschreibung findet sich bei Wikipedia: https://de.wikipedia.org/wiki/Shared_Nothing_Architecture

Wenn wir Daten von mehreren Instanzen brauchen, müssen wir dies händisch in der Anwendung machen (ein Spezialfall davon wäre ein Join in einer SQL-Datenbank). Das würden wir aber in diesem Szenario versuchen zu verhindern.

Was aber, wenn es keinen fachlichen Weg gibt, unsere Daten aufzuteilen?

3.2 Sharding

Nehmen wir den Key-Value-Store memcached als Beispiel. In memcached kann der Client zu einem selbst gewählten Key beliebige Daten ablegen, er kann zu einem Key die Daten abfragen und er kann die Daten zu einem Key löschen. Wenn wir nun so viele Daten haben, dass wir sie nicht mehr im Hauptspeicher eines einzelnen Knotens ablegen können, wie können wir weitere memcached Knoten hinzufügen?

Ein erster Ansatz wäre, die Einträge nach den Anfangsbuchstaben ihrer Keys aufzuteilen. Alle Keys, die mit A-L beginnen landen auf dem ersten Knoten, alle mit M-Z auf dem zweiten Knoten. Mit diesem Ansatz könnte jeder Client anhand des Keys selber entscheiden, wo die angefragten Daten zu finden oder abzulegen sind. Ein Problem dieses Ansatzes ist jedoch, dass die Daten nicht gleichmäßig auf die Knoten verteilt werden. Um dies zu erreichen könnten wir den Key erst hashen und mit Hilfe einer modulo Operation auf die Knoten verteilen. Dieser Ansatz löst das Problem der gleichmäßigen Verteilung. Ein Nachteil bleibt jedoch: Wenn wir die Anzahl der memcached Knoten verändern, müssen wir alle Daten umverteilen. Um dieses Problem zu lösen, gibt es den Ansatz von Consistent Hashing.

Beim Consistent Hashing wenden wir die Hashing Funktion sowohl auf den Namen des Knotens als auch auf die Keys an. Das ergibt den Hashring in der Abbildung. Die Knoten teilen den Ring in Segmente auf. Die Keys werden dann auf dem Knoten abgelegt, der das Segment verwaltet. Durch Consistent Hashing können wir dann weitere Knoten hinzufügen oder auch Knoten entfernen – in beiden Fällen sind immer nur Teile der Daten betroffen. Aber es bleibt stets erkennbar, wo welche Daten abgelegt oder gefunden werden können.

Wir sind mit diesem Ansatz jedoch nicht gegen Datenverluste geschützt. Im Gegenteil: Die Wahrscheinlichkeit für Datenverluste steigt sogar. Im Fall von

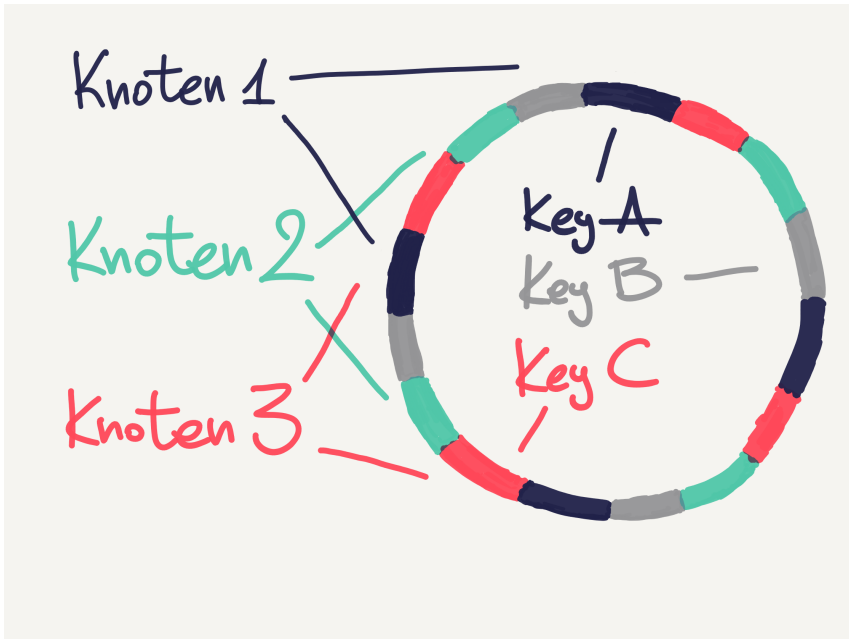


Abbildung 3.1: Consistent Hashing

memcached ist das okay, da es sich um einen Cache handelt. Was aber wenn wir Sharding in unserer SQL-Datenbank verwenden wollen? Der Verlust eines Knotens bedeutet hier den Verlust aller Daten dieses Knotens. Um dieses Problem zu lösen, müssen wir über ein anderes Feature sprechen: Replikation.

3.3 Replikation

Replikation bedeutet, dass Kopien eines Datensets auf verschiedenen Knoten abgelegt werden. Replikation ermöglicht somit sowohl einen schnelleren Zugriff wie auch einen Schutz vor Datenverlust. Bei der Replikation gibt es drei grundsätzliche Ansätze: Single Leader, Multi Leader und Leaderless. Diese Ansätze sind nicht besser oder schlechter, jeder hat seine Vor- und Nachteile.

3.3.1 Single Leader

In einem Single Leader Setup müssen alle Schreiboperationen an einen bestimmten Knoten, den Leader, geschickt werden. Lesezugriffe dürfen hingegen auf allen Knoten ausgeführt werden. Der Leader ist dafür zuständig, die Schreibvorgänge auf die Follower zu verteilen.

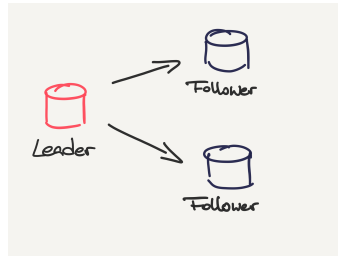


Abbildung 3.2: Single Leader

In diesem Setup können wir beim Verlust eines Followers einen neuen Follower aufsetzen, der dann die Historie aufholen und somit den alten Follower ersetzen kann. Beim Verlust des Leaders kann ein Follower zum neuen Leader ernannt werden. Diese Wahl eines neuen Leaders kann manuell durch Administratoren passieren oder automatisiert. Dabei müssen wir den Clients mitteilen, dass die Schreibzugriffe nun an einen anderen Knoten gehen. Hier gibt es Möglichkeiten von Konfigurationsänderungen bis zu Request Routing.

Bei einer Single Leader Replikation können wir die Lesezugriffe skalieren indem wir Lesezugriffe auf die Follower zulassen. Schreibzugriffe können auf diese Weise hingegen nicht skaliert werden. Je nach gewähltem Model werden diese sogar langsamer.

Beispiele: Redis, MariaDB, PostgreSQL und MongoDB.

Synchron-Asynchron

Daten können synchron oder asynchron repliziert werden. In einer synchronen Replikation wird eine Transaktion erst als durchgeführt angesehen, wenn alle Replikate diese bestätigt haben. Um dies zu ermöglichen, können Verfahren wie Two-Phase Commits eingesetzt werden. Das Hinzufügen von weiteren Replikaten senkt in diesem Fall die Schreibgeschwindigkeit, da der Schreibvorgang immer höchstens so schnell ist wie der langsamste Replikatknoten plus die Netzwerkverbindung.

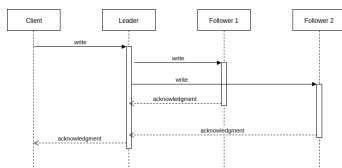


Abbildung 3.3: Synchrone Replikation

Um die Schreibgeschwindigkeit zu verbessern, wird asynchrone Replikation verwendet. Dabei wird die Transaktion als geschrieben angesehen, sobald ein Knoten diese Transaktion bestätigt hat. Wenn der Leader den Zugriff schon bestätigt hat, aber die Replikation noch nicht abgeschlossen wurde, bekommen Clients von verschiedenen Knoten für die gleiche Anfrage verschiedene Antworten (wir nennen das “Stale Read”). Der Absturz des Leaders zu einem ungünstigen Zeitpunkt kann zudem bedeuten, dass wir schon bestätigte Schreibzugriffe verlieren. Der Absturz eines Followers kann bedeuten, dass wir den Schreibzugriff bestätigen, dann aber nicht auf die gewünschte Anzahl von Replikaten unserer Daten kommen.

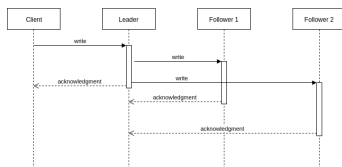


Abbildung 3.4: Asynchrone Replikation

Es gibt auch eine Mischform, die semi-synchrone Replikation. In diesem Fall legen wir die Anzahl von Replikaten fest, die den Schreibzugriff bestätigen müssen bevor er dem Client gegenüber bestätigt werden kann.

3.3.2 Multi Leader

Bei einem Multi Leader Setup heben wir die Einschränkung auf, das nur auf einen Knoten geschrieben werden darf. Alle Knoten können nun Schreibzugriffe entgegennehmen.

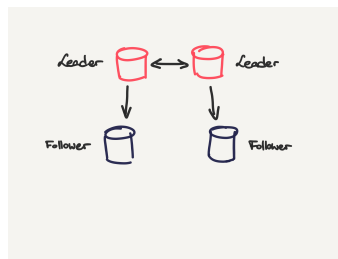


Abbildung 3.5: Multi Leader

Auch hier könnten wir die Daten synchron replizieren. Dabei würde aber die Schreibperformance schlechter werden als bei einem Single Leader mit synchroner Schreibperformance, was diese Option wenig attraktiv macht. Bei der asynchronen Replikation müssen wir nun aber mit Schreibkonflikten rechnen: Zwei Knoten könnten widersprüchliche Schreibzugriffe entgegennehmen und bestätigen. Wir sprechen über dieses Problem und Lösungen dafür im Detail in einem späteren Abschnitt.

Beim Multi Leader Setup können wir den Verlust von Knoten tolerieren und weitere Knoten hinzufügen. Beim Knoten Verlust kann es jedoch zum Verlust von Daten kommen, die noch nicht repliziert wurden. Wir können in diesem Setup sowohl Lese- als auch Schreibzugriffe skalieren.

Beispiele: CouchDB

3.3.3 Leaderless

Bei der Leaderless Replikation sendet der Client Lese- und Schreibzugriffe nicht zu einem, sondern zu mehreren Knoten. Im Hintergrund werden die Daten zwischen den Knoten asynchron repliziert. Dabei kann es zu widersprüchlichen Informationen kommen. Auch hier verweisen wir auf unseren Abschnitt zu Schreibkonflikten.

Eine Leaderless Datenbank arbeitet dabei mit einem sogenannten Quorum. Das Read-Quorum gibt an, von wie vielen Knoten eine Antwort auf die Anfrage zurückkommen muss. Das Write-Quorum gibt dies äquivalent für Schreibzugriffe an. Haben wir also beispielsweise 5 Knoten und ein Schreib- und Lesequorum von jeweils 3, wissen wir, dass unter der Voraussetzung, dass die gleichen Knoten verfügbar sind, bei jedem Lesezugriff mindestens einer der Knoten bei einem Schreibvorgang beteiligt war. Dabei ist es möglich, das Quorum je nach Use-Case zu wählen: Ein besonders kritischer Schreibvorgang wird mit einem hohen Quorum versehen, ein unkritischer nur auf einen Knoten geschrieben.

In diesem Setup können wir den Ausfall beliebiger Knoten tolerieren. Erst bei einem Ausfall von einer Anzahl Knoten höher dem Schreibquorum kann es zu Datenverlusten kommen. Es können damit sowohl Lese- als auch Schreibzugriffe skaliert werden.

Beispiele: Riak, Cassandra

3.4 Kombination von Sharding und Replikation

Sharding und Replikation schließen sich keineswegs aus. Shards können repliziert werden, um große Datenmengen ausfallsicherer abzulegen.

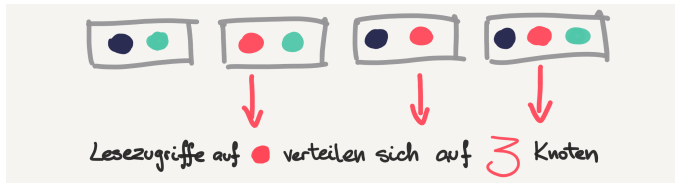


Abbildung 3.6: Sharding und Replikation

3.5 Skalierung von Vektordatenbanken

Vektordatenbanken stellen durch ihre spezielle Arbeitsweise besondere Anforderungen an die Skalierung. Die Hauptherausforderung liegt dabei in der effizienten Verteilung und Suche in hochdimensionalen Vektorräumen.

3.5.1 Sharding

Bei der Aufteilung der Daten auf mehrere Knoten können Vektordatenbanken nicht einfach nach einem Schlüssel sharden, da zusammengehörende (ähnliche) Vektoren dann möglicherweise auf verschiedenen Knoten landen würden. Stattdessen werden meist Verfahren wie Local Sensitive Hashing (LSH) eingesetzt, die ähnliche Vektoren mit hoher Wahrscheinlichkeit dem gleichen Shard zuordnen. Dadurch kann die Suche nach ähnlichen Vektoren zunächst auf wenige relevante Shards beschränkt werden.

Shard A: [Vektoren für "Hunde", "Katzen", "Haustiere"]
Shard B: [Vektoren für "Autos", "Motorräder", "Transport"]

3.5.2 Replikation

Die Replikation folgt bei Vektordatenbanken den bekannten Mustern. Allerdings ist zu beachten, dass die Vektoren selbst meist unveränderlich sind - neue Versionen eines Dokuments erzeugen neue Vektoren. Das vereinfacht die Replikation, da Konflikte bei gleichzeitigen Änderungen nicht auftreten können. Herausfor-

dernd ist hingegen die Größe der Vektorindizes, die ebenfalls repliziert werden müssen.

3.5.3 Performance

Die Suche nach ähnlichen Vektoren ist rechenintensiv. Um die Performance zu verbessern, werden oft Approximate Nearest Neighbor (ANN) Algorithmen eingesetzt, die einen Trade-off zwischen Genauigkeit und Geschwindigkeit ermöglichen. Dabei werden die Vektoren in einer Baumstruktur oder einem Graph organisiert, um nicht jeden einzelnen Vektor mit der Suchanfrage vergleichen zu müssen.

Exakte Suche:	100% Genauigkeit, lange Suchzeit
ANN-Suche:	98% Genauigkeit, deutlich schnellere Suchzeit

Anders als bei relationalen oder dokumentenorientierten Datenbanken liegt der Schwerpunkt bei der Skalierung von Vektordatenbanken weniger auf der Konsistenz der Daten, sondern mehr auf der effizienten Verteilung der Rechenlast für Ähnlichkeitssuchen. Die Herausforderung besteht darin, eine Balance zwischen Suchgeschwindigkeit, Genauigkeit der Ergebnisse und Ressourcenverbrauch zu finden.

3.6 Verfügbarkeit

Das funktioniert alles gut, bis irgendwas schief geht. Wir haben schon mehrfach den Ausfall eines Knotens angesprochen und die damit verbundenen Möglichkeit von Datenverlust. Aber auch das Netzwerk kann Probleme verursachen.

Was ist eigentlich Verfügbarkeit? Ein Knoten ist verfügbar, wenn er Requests verarbeiten kann. Ein Gesamtsystem ist voll verfügbar (total availability), wenn jeder Knoten im System, der erreichbar ist, auch verfügbar ist.

Bei einer Netzwerkpartition wird unser Cluster in zwei Teile geteilt. Innerhalb der beiden Partitionen können die Knoten noch miteinander kommunizieren, aber eine Kommunikation zwischen den Partitionen ist nicht mehr möglich. In dieser

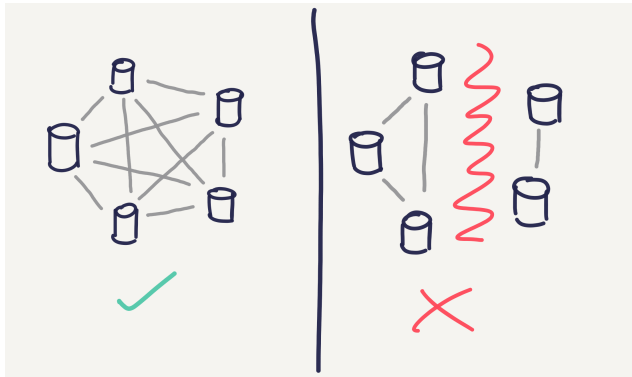


Abbildung 3.7: Netzwerkpartition

Situation haben wir nun zwei grundlegende Möglichkeiten (unter der Voraussetzung, wir haben die Partition richtig erkannt):

1. Wir hören auf, Requests entgegen zu nehmen. Unser System ist dann nicht mehr verfügbar (die Knoten sind erreichbar, verarbeiten aber keine Requests mehr). Aber wir erhalten die Konsistenz des Systems.
2. Wir nehmen weiter Requests entgegen. Nun sind die Antworten zumindest während der Partition nicht mehr konsistent, da Schreiboperationen in der einen Partition in der anderen nicht bekannt sind.

Das ist das Verhalten während der Partition. Was aber passiert nun nachdem die Netzwerkkonexion wieder hergestellt wurde? Das System welches keine Requests mehr entgegen genommen hat, kann wie gewohnt weiter arbeiten. Das andere System hat nun aber unterschiedliche Informationen auf den Knoten. Diese – teilweise vielleicht widersprüchlichen – Informationen müssen nun wieder auf einen Stand gebracht werden. Ist es möglich, die Informationen wieder auf einen Stand zu bringen, sprechen wir von Eventual Consistency: Auch wenn es Zeitpunkte gibt, zu denen das Gesamtsystem nicht konsistent ist, zu einem späteren Zeitpunkt wird sie immer wiederhergestellt. Um das zu erreichen, brauchen wir eine Konfliktbehandlungsstrategie.

3.7 Konfliktentdeckung und -behandlung

Ein einfacher Weg, mit Konflikten umzugehen, ist, die Zeitstempel der Schreibzugriffe zu vergleichen und den neusten zu übernehmen. Diese Strategie nennen wir Last-Write-Wins. Leider gibt es hier aber immer die Möglichkeit, das wir Daten verlieren. Als Beispiel sei ein Play Counter genannt, der zählen soll, wie oft ein Video angeschaut wurde. Zu einem Zeitpunkt x sind 5 Zugriffe gezählt. Nun greifen drei Personen auf das Video über Knoten 1 zu und vier Personen über den Knoten 2. Der Play Counter ist nun auf dem einen Knoten bei 8, auf dem anderen bei 9. Zu diesem Zeitpunkt ist der Play Count nicht konsistent. Synchronisieren sich nun die beiden Knoten und lösen den Konflikt über die Last-Write-Wins Strategie ist nun der Play Count 8 oder 9 (je nachdem, welcher den höheren Zeitstempel hat), obwohl der korrekte Play Count 12 wäre. Wir haben zwar nun wieder einen konsistenten Play Count, er ist aber inhaltlich falsch.

Möchten wir das umgehen, müssen wir erst einmal feststellen, das ein Konflikt aufgetreten ist. Dafür verwenden wir logische Uhren wie zum Beispiel Vector Clocks³. Eine Vector Clock gibt einem die Information, auf welchem Informationsstand eine Änderung durchgeführt wurde. Wurden zwei unabhängige Änderungen durchgeführt, müssen diese wieder zusammengeführt werden.

Um zwei Informationsstände wieder zusammenzuführen, können wir den Konflikt mit Hilfe der Benutzer oder mit Applikationslogik auflösen. Das Auflösen eines Konflikts durch den User kann eine sehr unangenehme Benutzererfahrung sein – vergleichbar mit einem Git Merge Konflikt. In unserem View Count Beispiel hingegen könnten statt der neuen, absoluten Zahl der Views die Veränderungen des View Counts als die Anzahl der Views, die dazu gekommen sind gespeichert werden. Diese können dann wieder zusammengeführt werden.

Auch komplexere Szenarien können automatisch gelöst werden. Dabei kann man auf Datenstrukturen zurückgreifen, die selber Konflikte lösen können, die sogenannten Conflict-free Replicated Data Types⁴. Diese ermöglichen es, gleichzeitig stattfindende Änderungen konfliktfrei zusammenzuführen. Sie definieren Organisation und Operationen der Daten. Sie garantieren, das sich die Daten

³ <https://zoo.cs.yale.edu/classes/cs426/2012/lab/bib/fidge88timestamps.pdf>

⁴ https://hal.inria.fr/hal-00932836/file/CRDTs_SSS-2011.pdf

in einem konsistenten Zustand befinden sobald alle Knoten alle Änderungen erhalten haben.

3.8 Zusammenfassung

Wir haben dargelegt, wieso es sich bei vielen Entscheidungen um Trade-Offs handelt: Ist es uns wichtiger, dass die Daten zu jedem Zeitpunkt konsistent sind – oder brauchen wir höhere Verfügbarkeit, schnelleren Zugriff oder geografische Verteilung? Die Entscheidung ist dabei nicht eine binäre Entscheidung, sondern hat verschiedene Abstufungen.

Wir müssen uns aber auch darüber bewusst sein, dass es sich dabei nicht um eine technische, sondern um eine fachliche Entscheidung handelt. Ähnlich wie beim Caching müssen wir von der fachlichen Seite fragen, wie viel Inkonsistenz für die Benutzer und Benutzerinnen hinnehmbar ist. Genauso ist es eine geschäftliche Entscheidung, welche Antwortzeiten akzeptabel sind oder ob unsere Anwendung weltweit verfügbar sein soll. Wie so oft gibt es keine magische Lösung, die alle Probleme löst: Wir müssen Entscheidungen treffen, und dabei bewusst mit den Nachteilen umgehen.

Noch weit mehr Informationen zum Thema Skalierung von Datenbanken finden Sie in [\[1\]code](#)

4 Betriebsaspekte

Die Zuverlässigkeit einer Anwendung wird von der ihr zugrunde liegenden Infrastruktur bestimmt – also auch von der Zuverlässigkeit der Datenbanken. Deshalb ist es entscheidend, dass wir Datenbanken so betreiben, dass ein Ausfall oder Datenverlust vermieden werden kann, denn dieser würde unweigerlich auch zu einem Ausfall der abhängigen Anwendungen führen.

4.1 Trends

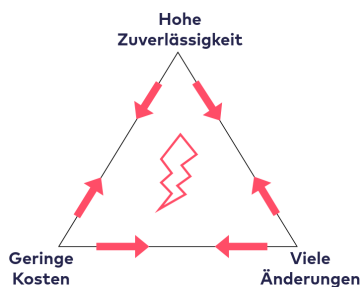
Viele Anwendungen werden heute mit einer **Microservice** oder **Self-contained Systems**¹ Architektur realisiert. Dabei verantwortet häufig jeder Service seine eigene Datenhaltung. Das kann ein eigenes Schema innerhalb einer von mehreren Services genutzten PostgreSQL Datenbank sein oder auch eine Spezialdatenbank, welche die Anforderungen des jeweiligen Service besonders gut erfüllt, beispielsweise eine Zeitreihendatenbanken für Sensordaten. Letzteres ist insbesondere sinnvoll, wenn der entsprechende Service sehr spezialisiert ist oder mit sehr großen Datenmengen arbeitet die effizient verwaltet werden müssen. Dabei gilt es abzuwägen, zwischen der optimalen Datenbank für den jeweiligen Microservice und der Vermeidung von unkalkulierbaren Betriebsaufwänden durch eine Vielzahl unterschiedliche Datenbanken.

Es gibt jedoch auch gegenläufige Trends, die auf eine Verringerung der Betriebsaufwände abzielen. Da ist zum einen die zunehmende Nutzung von Cloudangeboten. Diese bieten üblicherweise eine Reihe von Managed Services für die Datenhaltung, beispielsweise Object Stores zum Speichern großer Mengen an Binärdaten oder Standardsetups für relationale Datenbanken. Dabei übernimmt der jeweilige **Cloud** Anbieter die Wartung der Infrastruktur wie Backups, Patching und Skalierung wodurch Betriebsaufwände für Nutzer*innen deutlich verringert werden können. Mehr zu den Vor- und Nachteilen dieses Betriebsmodells finden Sie im Abschnitt 4.3. Zum anderen legen viele neuentwickelte Datenbanken einen Fokus auf einfache Betreibbarkeit. Hier seien die Buzzwords **Operational Simplicity** und **Cloud Native Database** genannt. Diese beziehen sich vor allem

¹<https://scs-architecture.org>

auf Aspekte wie Elastizität gegenüber schwankenden Datenmengen, Hochverfügbarkeit und automatische Fehlerbehandlung. Ein Beispiel dafür ist die NewSQL Datenbank CockroachDB, welche speziell für Cluster-Setups konzipiert wurde. Weitere Details zu CockroachDB finden sich im Kapitel 6.5. Daneben gibt es im Betriebsbereich in den letzten Jahren große methodische Fortschritte, die auch vor dem Betrieb von Datenbanken nicht halt machen. Ein Beispiel hierfür ist Database Reliability Engineering.

4.2 Database Reliability Engineering



Häufig bewegen sich Betriebsinfrastruktur und die zugehörigen Betriebsabläufe von Systemen in einem Spannungsfeld sich widersprechender Ziele: Hohe Zuverlässigkeit mit geringen Kosten bei gleichzeitig hoher Änderungsgeschwindigkeit.

Site Reliability Engineering (SRE) ist eine Methodik, die es ermöglicht für eine Anwendung einen möglichst guten Kompromiss zwischen diesen Zielen zu finden. Sie ist eng mit DevOps verwandt, wurde ursprünglich von Google entwickelt und wird mittlerweile von zahlreichen Unternehmen eingesetzt. Ihre Kernbestandteile sind:

- Das Erfassen von Messwerten für die Servicequalität, sogenannten **Service Level Indicators** aus Benutzersicht und deren Abgleich mit Grenzwerten, sogenannten **Service Level Objectives** via Monitoring
- Ein strukturiertes **Risikomanagement** für mögliche Ursachen von Störungen und Ausfällen
- Ersetzen manueller Tasks (*Toil*) durch **Automatisierung**
- Erstellen von **Postmortems** nach dem Beheben einer Störung mit dem Ziel, systembedingte Schwachstellen herauszuarbeiten und zu adressieren

Eine umfangreiche Beschreibung von SRE findet sich in [2]code und [3]code. [4]code erläutert darüber hinaus detailliert den Einsatz im Datenbankbereich.

4.2.1 Service Level Indicators (SLIs)

SLIs sind mengenbezogene Messwerte (*Metriken*), welche die Servicequalität eines Dienstes aus Nutzersicht messen sollen. Das bedeutet: Sollten aus Nutzersicht Probleme bei der Nutzung eines Dienstes auftreten, sollte dies aus den SLIs erkennbar sein. Die Definition eines SLIs beinhaltet dabei, was ein “guter” Messwert ist, beispielsweise eine Antwortzeit von weniger als 100ms. Außerdem sollte festgelegt werden, wo dieser Messwert erfasst wird: An der Datenbank, am Datenbank Loadbalancer, innerhalb einer Anwendung welche die Datenbank verwendet etc. Für Datenbanken sind insbesondere die folgenden SLIs interessant:

- **Latenz:** Wie lange dauert es bis eine Anfrage beantwortet wird?
- **Verfügbarkeit:** Ist das System in der Lage Anfragen erfolgreich zu beantworten?
- **Durchsatz:** Wie viele Anfragen werden in einer gegebenen Zeiteinheit erfolgreich beantwortet?
- **Precision/Recall:** Im Falle der Vektorsuche: Wie präzise sind die Suchergebnisse? Welcher Anteil relevanter Dokumente wird gefunden?
- **Dauerhaftigkeit:** Werden alle Schreiboperationen erfolgreich dauerhaft gespeichert oder sind Daten, beispielsweise durch einen Systemabsturz, verloren gegangen? Falls ja, wie viele?
- **Kosten/Effizienz:** Wieviel kostet eine spezifische Aktion?

Zwei interessante Datenpunkte die im Auge behalten werden sollten sind darüber hinaus noch:

- **Mean Time Between Failures (MTBF):** Mittlere Dauer zwischen Ausfällen
- **Mean Time To Recover (MTTR):** Mittlere Wiederherstellungszeit nach einem Ausfall – inkl. Adressierung von Datenverlust und Inkonsistenzen

4.2.2 Service Level Objectives (SLOs)

SLOs legen Ziele fest, welcher prozentuale Anteil der Messwerte eines SLIs “gute” Messwerte sein sollten. Wird dieser Anteil nicht erreicht oder droht nicht erreicht zu werden, ist ein menschliches Eingreifen notwendig. Ein gewisser Anteil an “schlechten” Messwerten ist bei den meisten SLIs tolerierbar und kann gerade

in einer komplexen, vernetzten Systemlandschaft immer mal wieder auftreten, beispielsweise durch Systemaktualisierungen, Cluster Rebalancing, Backups oder ETL-Jobs. In vielen Fällen erledigen sich die Probleme von selbst und erfordern kein explizites Eingreifen. Welcher Anteil an “schlechten” Werten tolerabel ist, muss individuell für jedes System und jeden SLI betrachtet werden.

Erwägenswert sind hier insbesondere zwei Ziele:

- **Recovery Point Objective:** Wie viele Daten dürfen maximal verlorgen gehen?
- **Recovery Time Objective:** Wie lange darf es maximal dauern, bis die Datenbank wieder verfügbar ist?

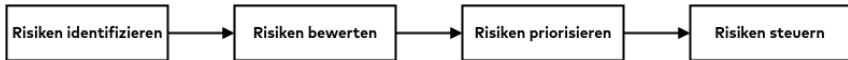
Monitoring

Das Monitoring eines Dienstes sollte die SLIs möglichst gut und genau erfassen und visualisieren. Im Idealfall sind die Monitoring Dashboards hierarchisch aufgebaut: Das oberste Dashboard zeigt Kacheln für die SLIs und zugehörige SLOs, sowie ggf. deren Verletzung an. Von den Kacheln der einzelnen SLIs gelangt man zu technisch detaillierteren Dashboards welche die Frage “Was ist kaputt und warum?” adressieren. Potentielle Metrik-Kandidaten für letzteres sind:

- **Metriken der Infrastruktur:** CPU-Auslastung, Arbeitsspeicher, Storage I/O, Netzwerk I/O, Server und Service Neustarts, Konfigurationsänderungen
- **Metriken der Datenbank:** Anzahl der Datenbankverbindungen und ihr Status, Anzahl/Durchsatz/Latenz von Schreib-/Lese-/Schemaoperationen, Zeitdifferenz der Schreiboperationen zwischen Leader und Follower (Replication Lag), Locks und Rollbacks

4.2.3 Risikomanagement

Um Risiken die zum Auftreten kritischer Systemausfälle führen können zu verfolgen, empfiehlt sich die Etablierung eines Risikomanagementprozesses. Dieser sollte bei jeder größeren Änderung der Systemlandschaft neu durchlaufen werden und bei schwerwiegenden Systemausfällen entsprechend angepasst werden. Ein solcher Prozess kann beispielsweise aus vier Schritten bestehen:



Identifikation

Zunächst wird eine Sammlung aller wesentlichen, erkennbaren Risiken zusammengestellt. Zentral ist dabei die Frage: Was kann schiefgehen so dass ein SLO verletzt wird? Typische Risiken im Datenbankumfeld sind:

- Hohe Latenz bedingt durch Lastspitzen welche beispielsweise durch Marketingkampagnen oder verkaufsintensive Tage wie Weihnachten oder Black Friday erzeugt werden
- Entwickler führen Änderungen an einer Anwendung durch welche die Anzahl der Datenbankabfragen vervielfachen
- Schlecht geplante Schemaänderungen erfordern einen längerdauernden Lock einer Tabelle, währenddessen keine Schreibzugriffe möglich sind
- Der primäre Datenbankserver fällt aus, der automatische Failover auf einen Replikatsserver funktioniert nicht

Bewertung

Im Rahmen der Risikobewertung wird anschließend beurteilt wie hoch die Eintrittswahrscheinlichkeit jedes Risikos ist und welche Auswirkungen dessen Eintritt hätte.

Priorisierung

Anhand der Eintrittswahrscheinlichkeit und der Schwere der Auswirkungen werden die Risiken priorisiert und entschieden, welche Risiken adressiert werden sollen und welche ignoriert werden können.

Steuerung

Zur Steuerung der Risiken kommen im Wesentlichen die drei folgenden Maßnahmen in Frage?

- Risikovermeidung durch technische Maßnahmen und Automatisierung, beispielsweise durch den Einbau automatischer Skalierungsmöglichkeiten und deren Überprüfung mittels Lasttests
- Risikoverminderung durch Monitoring, beispielsweise der Anzahl der Lese- und Schreibzugriffe auf der Datenbank

- Erstellung detaillierter Handlungsanweisungen (*Playbooks*) für den Fall, dass das Risiko eintritt, beispielsweise ein manuelles Umschalten des primären Datenbankservers erfolgt

Bei Auswahl der Maßnahmen für das jeweilige Risiko gilt es natürlich auch eine Kostenabwägung zu treffen, schließlich sollen die Kosten der Sicherungsmaßnahmen nicht die Kosten der Schäden des Risikoeintritts übersteigen.

4.3 Virtuelle Maschine, Managed Service oder Kubernetes?

Im Wesentlichen gibt es drei Optionen um Datenbanken in einer (Cloud-)umgebung zu betreiben:

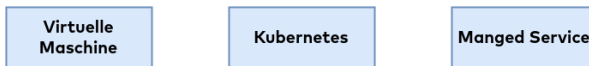


Abbildung 4.1: Betriebsoptionen

4.3.1 Virtuelle Maschine oder Bare Metal Server

Die Installation einer Datenbank in einer virtuellen Maschine oder direkt auf einem Hardware Server (*Bare Metal*) bietet maximale Auswahl und Konfigurationsoptionen: Es kann nahezu jede Datenbank verwendet werden und alle Möglichkeiten bezüglich Plugins und Konfigurationseinstellungen ausgeschöpft werden. Auch die Wahl der Datenspeicher muss gut überlegt sein: Für die Datenbank selbst muss Speicher gefunden werden, welcher den Anforderungen hinsichtlich Netzwerk und Platten I/O genügt, für Backups ein günstiger Speicher der die anfallenden Datenmengen effizient verwalten kann. Alles in allem bietet die Verwendung einer virtuellen Maschine viele Möglichkeiten – nicht zuletzt die Gewissheit, dass die Daten im eigenen Rechenzentrum liegen – erfordert aber auch einiges an Aufwand: Aspekte wie Installation, Updates, Skalierung, Datenschutz, Datensicherheit, Failover, Backups etc. müssen selbst konfiguriert und gemanagt werden.

Um dabei Effizienz und Nachvollziehbarkeit zu gewährleisten empfiehlt sich die Zuhilfenahme eines “Infrastructure as Code”² Tools wie Ansible, Terraform oder Pulumi.

4.3.2 Kubernetes

Kubernetes ist ein Open-Source Clustermanagementsystem zur Verwaltung von Containern und erfreut sich seit mehreren Jahren stetig wachsender Beliebtheit als Plattform für den Betrieb von Anwendungen. Auch Datenbanken lassen sich grundsätzlich damit betreiben. Da es sich bei Kubernetes um eine Standardsoftware handelt, lässt sich damit ein Provider Lock-in weitgehend vermeiden. Alle größeren Cloud Provider haben mittlerweile Kubernetes Cluster im Angebot, auch ein selbstbetriebenes Hosting ist möglich. Um Datenbanken in einem Kubernetes Cluster betreiben zu können, müssen im Kubernetes Cluster sogenannte Persistent Volumes zur Verfügung stehen, also Storage in denen die eigentlichen Daten der Datenbank abgelegt werden können. Es sollte darauf geachtet werden, dass der Cluster die Anforderungen der Datenbanken bezüglich Latenz und Durchsatz von Storage und Netzwerk I/O sowie der Menge an Arbeitsspeicher mit hinreichend Puffer für Spitzenlasten erfüllen kann.

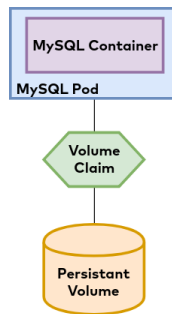


Abbildung 4.2: Kubernetes

² „Infrastructure as Code“ (IaC) bezeichnet das automatisierte Installieren und Einrichten von Serverinfrastruktur mittels spezieller Skripting Tools.

Grundsätzlich gibt es zwei Möglichkeiten wie Datenbanken in Kubernetes installiert und betrieben werden können: Mittels **Helm Charts** oder mittels **Kubernetes Operatoren**.

Bei Helm handelt es sich um einen Paketmanager für Kubernetes, bei Operatoren um Controller für das automatisierte Management komplexer Anwendungen. Helm Charts³ ermöglichen es, alle Ressourcen, die von einer Anwendung benötigt werden, einheitlich zu konfigurieren und Aktualisierungen von Containern unkompliziert auszurollen. Zustandsbehaftete Anwendungen wie Datenbanken erfordern häufig komplexe Administrationsprozesse, bei deren Automatisierung Helm Charts an ihre Grenzen stoßen.

Mit Operatoren lassen sich auch solche Prozesse abbilden, beispielsweise Backups und Replikation. Operatoren sind allerdings noch vergleichsweise neu. Es gibt einige Unternehmen, beispielsweise Zalando⁴, die Kubernetes Operatoren erfolgreich für ihre Datenbanksetups nutzen, insgesamt ist dieser Ansatz aber noch mit Vorsicht zu betrachten und es sollte vor dem Einsatz eines Operators sorgfältig geprüft werden, welche Features und welchen Reifegrad er besitzt. Die Seite OperatorHub⁵ stellte eine Auflistung verfügbarer Operatoren bereit, mit dem Operator SDK⁶ lassen sich eigene Operatoren erstellen.

4.3.3 Managed Services

Alle größeren Cloudanbieter haben mehrere Datenbanken als Managed Service im Angebot und auch einige Datenbankhersteller bieten ihre Datenbanken in dieser Form an. Dabei stellt der Anbieter eine vorkonfigurierte Datenbank bereit und übernimmt deren Betrieb. Die Nutzung eines Managed Services **vereinfacht die Konfiguration vieler Betriebsaufgaben** erheblich. Backups, Skalierungsverhalten, Hochverfügbarkeitssetups etc. lassen sich einheitlich über die Kommandozeilentools oder APIs des Cloudproviders oder auch mittels Infrastructure-as-Code Tools einrichten. Gleichzeitig sind die Konfigurationsmöglichkeiten aber auf die Vorgaben des jeweiligen Anbieters beschränkt (siehe Abbildung). Auch ist mit

³ Sammlung von Helm Charts: <https://github.com/helm/charts>

⁴ <https://github.com/zalando/postgres-operator>

⁵ Sammlung von Kubernetes Operatoren <https://operatorhub.io>

⁶ <https://github.com/operator-framework/operator-sdk>

höheren Kosten zu rechnen als bei reinen Infrastrukturdiensten wie virtuellen Maschinen.

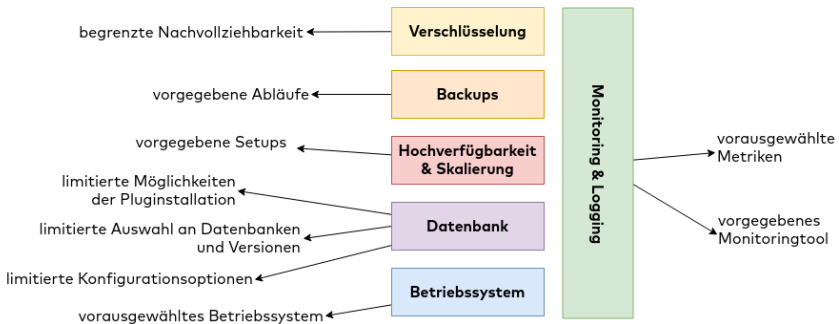


Abbildung 4.3: Datenbanken als Managed Service

Von den großen Cloud Providern werden üblicherweise relationale Datenbanken, Key-Value-Stores und Dokumentendatenbanken angeboten. Bei einigen Providern stehen darüber hinaus auch Graph- oder Zeitreihendatenbanken zur Auswahl. Meistens handelt es sich dabei um etablierte Open-Source Projekte beispielsweise PostgreSQL oder Redis. Es gibt aber auch proprietäre, **auf Cloud-Skalierbarkeit optimierte Datenbanken** wie beispielsweise Google Cloud Spanner, Azure Cosmos DB oder Amazon Aurora. Auch bei der Verwendung einer etablierten Open-Source Datenbank, lässt sich jedoch ein gewisser Vendor Lockin nicht gänzlich vermeiden. So ist beispielsweise das User Management je nach Cloud Provider unterschiedlich, auch das Erstellen und Wiedereinspielen von Backups wird unterschiedlich gehandhabt. Praktisch jeder größere Cloud Provider bietet neben gemanagten Datenbanken außerdem einen sogenannten **Marketplace** an. Dort können Drittanbieter wie Datenbankhersteller vorkonfigurierte Datenbankimages bereitstellen die beispielsweise auf virtuellen Maschinen betrieben werden können. Diese Images werden üblicherweise gegen Entgelt angeboten und beinhalten in vielen Fällen auch Supportleistungen.

4.3.4 Kriterien die für Auswahl

Bei der Auswahl einer geeigneten Betriebsplattform und damit u.U. auf für oder gegen eine Datenbank sollten die folgenden Aspekte miteinbezogen werden:

- **Datenschutz/Datensicherheit:** Können die Daten verschlüsselt gespeichert werden? Ist es OK, wenn ausländische Sicherheitsbehörden auf die Daten zugreifen können?
- **Setup- und Wartungsaufwand:** Kann ein vorhandenes Deployment Tool auch für die Datenbank verwendet werden? Wie aufwendig sind Versionswechsel, Aufsetzen einer neuen Datenbank, Skalierung, Backup, Failover zu konfigurieren?
- **Infrastrukturkosten:** Welche Kosten entstehen für den Datenbankserver, Storage, Backup und Traffic?
- **Backup und Recovery:** Wie können Backups erstellt werden? Wo werden sie gespeichert? Wie funktioniert ein Recovery?
- **Hochverfügbarkeit:** Welche Replikations- und Clusterszenarien werden unterstützt?
- **I/O-Durchsatz:** Welche Limits gibt es bezüglich Netzwerk I/O, Platten I/O und IOPS⁷?
- **Vendor Lock-in:** Welche Features des Cloud Anbieters oder der Deploymentumgebung erschweren einen Wechsel?
- **Monitoring:** Kann ein vorhandenes Monitoring Tool auch für die Datenbank verwendet werden?

4.3.5 Besonderheiten beim Betrieb von Vektordatenbanken

Vektordatenbanken stellen einige spezifische Anforderungen an den Betrieb:

Ressourcenplanung Anders als bei traditionellen Datenbanken ist nicht nur der Speicherplatz entscheidend, sondern auch die verfügbare GPU-Leistung. Diese wird sowohl für die Erzeugung der Embeddings als auch für effiziente Ähnlichkeitssuchen benötigt. Bei der Planung der Infrastruktur muss daher berücksichtigt werden:

⁷ IOPS: Input/Output Operations Per Second. Eine ausführliche Beschreibung findet sich bei Wikipedia: https://de.wikipedia.org/wiki/Input/Output_operations_Per_Second

- Speicheranforderungen für die Rohdaten
- Speicheranforderungen für die Vektoren (typischerweise 384-1024 Dimensionen pro Vektor)
- GPU-Kapazität für Embedding-Generierung und Suche
- RAM für Vektorindizes

Managed Services Vektordatenbanken werden zunehmend als Managed Services angeboten:

- Pinecone als spezialisierter Vektordatenbank-Service
- AWS mit Kendra und OpenSearch für Vektorsuche
- MongoDB Atlas mit integrierter Vektorsuche
- Azure Cognitive Search mit Vektorsuche

Der Vorteil dieser Services liegt in der automatischen Skalierung der benötigten Ressourcen und der Integration mit den jeweiligen KI/ML-Diensten für die Embedding-Generierung.

Monitoring Neben klassischen Metriken wie Latenz und Durchsatz müssen bei Vektordatenbanken zusätzlich überwacht werden:

- Qualität der Suchergebnisse (Precision/Recall)
- Performance der Embedding-Generierung
- GPU-Auslastung und -Verfügbarkeit
- Index-Performance und -Größe

Backup und Recovery Bei Vektordatenbanken muss unterschieden werden zwischen:

- Backup der Rohdaten
- Backup der generierten Embeddings
- Backup der Indexstrukturen

Da die Embedding-Generierung rechenintensiv ist, empfiehlt sich meist die Sicherung aller drei Komponenten. Allerdings können Embeddings bei Bedarf auch neu generiert werden, was bei einem Recovery mehr Zeit in Anspruch nimmt, dafür aber weniger Speicherplatz für Backups benötigt.

4.4 Backup, Restore & Recovery

Es gibt viele Gründe, warum das regelmäßige Anfertigen von **Backups**, also der doppelten Speicherung des Datenbestands einer Datenbank an einem anderen Ort, sinnvoll sein kann (siehe auch [\[4\]code](#)):

- **Geplante Einsatzszenarien** bei denen die Daten noch an anderer Stelle benötigt werden, beispielsweise zum Klonen von Umgebungen, für ETL & Daten Pipelines oder zur Durchführung von Betriebstests
- **Ungeplante Einsatzszenarien** bei Datenkorruption oder -verlust durch Anwendungsfehlern, Konfigurationsfehlern oder Hardwarefehlern

Bei einem **Restore** wird eine Datenbank aus den Backup Dateien wiederhergestellt. Mit **Recovery** bezeichnet man den kompletten Prozess des Wiederherstellens eines konsistenten Datenbestandes, dies kann ein Restore oder auch nur das Zurückrollen einzelner Transaktionen umfassen.

Backups können auf unterschiedliche Art und Weise erstellt werden. Welche davon gewählt werden hängt von den verfügbaren Tools, der verwendeten Datenbank und dem Anwendungskontext ab:

Physisch: Es wird ein konsistenter Snapshot der Dateien der Datenbank erzeugt, z.B. mittels ZFS⁸ Snapshots **Logisch:** Die Daten werden aus der Datenbank exportiert und in ein portables Dateiformat überführt.

Online: Das Backup erfolgt im laufenden Betrieb. Dafür muss es möglich sein, einen konsistenten Snapshot der Daten zu erstellen **Offline:** Die Datenbank wird vor dem Erstellen des Backups heruntergefahren

Voll: Es wird ein Backup des kompletten Datensets der Datenbank erstellt **Inkrementell:** Es werden nur die Änderungen seit dem letzten Backup weggespeichert

Backups unterscheiden sich auch in der Häufigkeit in der sie erstellt werden. Je kürzer die Intervalle desto geringer der Datenverlust bei einem Restore – aber desto höher unter Umständen die Belastung für das Livesystem. Im Fehlerfall ist es am komfortabelsten, wenn ein sogenanntes *Point-in-time Recovery* möglich ist,

⁸ ZFS: Dateisystem für die Verwendung im Server- und Rechenzentrumsbereich.

bei dem der Datenbestand eines frei wählbaren Zeitpunktes in der Vergangenheit wieder hergestellt wird.

In jedem Fall sollte regelmässig die Funktionsweise des Restore getestet werden. Ein Backup welches im Einsatzfall nicht zeitnah eingespielt werden kann, ist wertlos.

Eine besondere Herausforderung ist oftmals die Erstellung von Backups in Cluster-Setups mit verteilten Datenbeständen, da es dort üblicherweise nicht möglich ist, ein konsistentes Backup zu erstellen. Manchmal bleibt hier nur die Möglichkeit einen Cluster doppelt zu betreiben.

5 Beispiele

5.1 Relationale Datenbank: PostgreSQL

<https://www.postgresql.org>

PostgreSQL ist eine klassische relationale Datenbank (mit einigen Erweiterungen), d.h. sie ist schemabasiert, die Daten werden in Tabellenform gespeichert und sie unterstützt ACID Transaktionen und referentielle Integrität. Eingesetzt wird sie vor allem im **Backendbereich von Enterprise- und Web Anwendungen**.

PostgreSQL ist ein **Open Source Projekt** und steht unter der PostgreSQL License, einer Variante der **MIT License**. Entwickelt wird es von einer Community unter Beteiligung mehrerer Unternehmen. Alle Daten sind stets in aktuelle Form auf der Festplatte. PostgreSQL verwendet den vorhandenen Hauptspeicher, um schnelle Zugriffe zu ermöglichen. Es gibt zahlreiche Parameter, mit denen sich das Speicher- und Performanceverhalten der Datenbank – je nach Nutzlast beeinflussen lassen.

5.1.1 Datenmodell

PostgreSQL verwendet ein tabellenbasiertes, relationales Datenmodell. Es unterstützt übliche Datentypen für Boolesche Werte, Zahlen, Zeichenketten und Datumsangaben. Daneben aber auch JSON, Arrays, UUIDs, Key-Value Pairs, Netzwerkadressen, geometrische Figuren und geographische Koordinaten – teilweise müssen dafür allerdings Plugins installiert werden.

Beispiel

id	name	street	city	style	opening_hours
282b6db9-f62d-4875-868f-205b1f8f16c7	Kleiner Drachen	Maximilianstr. 5	München	chinesisch, dim sum	Montag-Freitag: 11-21 Uhr
f5b99363-151f-41ef-a80d-f3fb10eb2b9c	Toskana	Balandstr. 123	München	italienisch	Dienstag-Sonntag: 17-23 Uhr
b4d00799-d409-4036-a543-2c9903da4573	The Bavarian Burger	Amalienstr. 45	München	burger, bayrisch	Dienstag, Mittwoch: 9-13 Uhr Donnerstag: 9-16 Uhr Freitag-Sonntag: 9-18 Uhr

Abbildung 5.1: Datenmodell PostgreSQL

Der Datenzugriff erfolgt mittels SQL unter Verwendung eines proprietären TCP Protokolls. Für die Abfrage von JSON- und Array-Elementen wird SQL um zusätzliche Operatoren ergänzt.

Einen Datensatz schreiben

```
INSERT INTO takeaway (id,name,street,city,style,opening_hours)
VALUES ( '282b6db9-f62d-4875-868f-205b1f8f16c7',
        'Kleiner Drachen', 'Maximilianstr. 5', 'München',
        '{chinesisch,dim sum}',
        '{"Montag-Freitag":"11-21 Uhr"}' );
```

Einen Datensatz lesen

```
SELECT name, street, opening_hours FROM takeaway
WHERE city = 'München' and 'chinesisch' = ANY (style);
```

5.1.2 Betrieb

PostgreSQL beinhaltet Konfigurationsoptionen für **Single-Leader Replikation**. Multi-Leader Replikation ist via Drittanbieterlösungen möglich, deren Konflikt-auflösung basiert allerdings auf Last-Write-Wins.

Um **Backups** zu erstellen gibt es mehrere Möglichkeiten: Die erste Möglichkeit ist das PostgreSQL eigene Tool `pg_dump` zu verwenden, mit dem Backups der kompletten Datenbank erstellt werden können. Als zweite Möglichkeit können Snapshots der Datenbank Files erstellt werden, sofern dies vom Dateisystem unterstützt wird. Dies ist beispielsweise bei ZFS der Fall. Eine dritte Möglichkeit ist es kontinuierlich den Write-Ahead-Log (WAL) zu sichern. In diesem speichert PostgreSQL jede Änderung an der Datenbank. Diese Möglichkeit ist am kompliziertesten aufzusetzen und erfordert viel Speicherplatz ermöglicht jedoch ein Point-in-time recovery bei dem im Einsatzfall nur eine sehr geringe Anzahl an Änderungen verloren geht. Daneben gibt es noch weitere Tools von Drittanbietern, welche die genannten Möglichkeiten zuverlässig und einfach einsetzbar machen, wie beispielsweise Barman¹.

Praktisch alle größeren Cloudanbieter z.B. AWS, Azure, Google Cloud und Digital Ocean haben eine **managed PostgreSQL Datenbank** im Angebot. Zu beachten sind dabei die Details: Nicht alle Anbieter unterstützen Point-in-Time Recovery und es gibt Unterschiede bezüglich maximal verfügbarem Speicherplatz und der Anzahl gleichzeitig möglicher Verbindungen. Zwar unterstützen alle Anbieter das automatische Einrichten von Read Replicas, doch auch hier gibt es Unterschiede, wie beispielsweise die Unterstützung der mit PostgreSQL 10 eingeführten logischen Replikation. Mit AWS Aurora gibt es daneben noch eine zu PostgreSQL weitgehend kompatible Datenbank mit flexibleren Skalierungsmöglichkeiten.

Für den Betrieb einer PostgreSQL Datenbank mittels **Kubernetes Operator** gibt es gleich mehrere Optionen. Am ausgereiftesten erscheinen uns derzeit die Operatoren von Zalando² und CrunchyData³.

¹<https://www.pgbarman.org>

²<https://github.com/zalando/postgres-operator>

³<https://github.com/CrunchyData/postgres-operator>

5.2 Datenstruktur Server: Redis

<https://redis.io>

Redis ist ein Datenstruktur Server und Key-Value-Store, d.h. Daten werden als Schlüssel-Wert-Paare gespeichert. Eingesetzt wird Redis vor allem als **Cache z.B. im Frontendbereich von Enterprise- und Web Anwendungen, als Session Store oder als Buffer beim Austausch von Nachrichten** zwischen Anwendungen.

Die Datenhaltung erfolgt **In-Memory**, was einen extrem schnellen Zugriff auf die Daten ermöglicht. Es gibt aber auch mehrere Optionen die Daten gleichzeitig – mit Auswirkungen auf die Performance – auch zu persistieren, entweder in festen Zeitabständen einen Abzug der gesamten Datenbank mittels RDB (Redis Database File) oder mit jeder Schreiboperation mittels AOF (Append Only File).

Redis wird vom Unternehmen Redis Labs entwickelt. Die Datenbank selbst steht unter einer Open-Source-Lizenz (BSD-3), die zusätzlich angebotene Enterprise Version bringt ergänzende Funktionen wie beispielsweise eine CRDT-basierte Multi-Leader Replikation oder Disaster Recovery Funktionen. KeyDB⁴ ist ein Redis kompatibler Fork, der mit Multi-Leader Support und noch höherer Performance wirbt.

5.2.1 Datenmodell

Redis verwendet ein Key-Value basiertes, schemafreies Datenmodell. Die Values können dabei nicht nur Strings sein, sondern auch komplexe Datentypen wie Listen, Bit Arrays oder Hash Maps. Redis bietet zu jedem Datentyp verschiedene Operationen an. So kann beispielsweise in einer Operation ein Eintrag aus einem Set gelöscht und in ein anderes hinzugefügt werden.

⁴<https://keydb.dev>

Beispiel

Key	Value
user:123:session	time: 10927359 username: hellojoe
page.index.html	<html><head>...
foobar:elements	[105, 309, 403, 405, 407]
r:key	789

Abbildung 5.2: Datenmodell Redis

Für die Kommunikation mit dem Redis Server wird ein eigenes, TCP-basiertes Protokoll namens RESP (REdis Serialization Protocol) verwendet. Es gibt Clients in einer Vielzahl von Programmiersprachen.

Einen Datensatz schreiben (mit Python):

```
redis_client.set('r:key', '789')
```

Einen Datensatz lesen (mit Python):

```
redis_client.get('r:key')
```

5.2.2 Betriebsoptionen

Redis beinhaltet eine Unterstützung für **Read Replicas**. Defaultmäßig wird dabei asynchrone Replikation verwendet. Multi-Leader Replikation wird nur von der Enterprise Version unterstützt. **Redis Cluster** bietet die Möglichkeit, Daten über mehrere Knoten zu verteilen. Es unterstützt zusätzlich außerdem Read Replicas. Redis Sentinel ermöglicht Hochverfügbarkeit in dem es einen Follower Knoten zum neuen Leader ernannt, wenn der Leader Knoten nicht mehr erwartungsge-
mäß funktioniert.

Ein **Backup** ist bei einigen Einsatzbereichen, z.B. als Cache, unter Umständen gar nicht notwendig, da die Datenbank ggf. aus anderen Quellen neu befüllt werden kann. Falls es notwendig ist, kann Persistenz aktiviert werden und die geschriebenen AOF oder RDB Dateien gesichert werden. Redis bietet an dieser Stelle verschiedene Mechanismen an um korrupte Dateien zu verhindern oder zu reparieren.

Viele Cloudanbieter bieten Redis als **Managed Service** an, es gibt beispielsweise Amazon ElastiCache, Google Cloud Memorystore, Azure Cache for Redis und Digital Ocean Redis. Die einzelnen Angebote unterscheiden sich in der Unterstützung von Persistenz, den angebotenen Replikationssetups, der Unterstützung von Sharding und der angebotenen Backup Optionen. Redis Labs bietet einen **Kubernetes Operator** für Redis Enterprise an. Daneben gibt es für Redis noch mehrere Operatoren von anderen Anbietern. Am ausgereiftesten erscheint momentan der Operator von Spotahome⁵.

5.3 Zeitreihendatenbank: InfluxDB

<https://www.influxdata.com>

InfluxDB ist eine Zeitreihendatenbank (time series database), d.h. alle Einträge werden mit einem Zeitstempel versehen und die Datenbank beinhaltet erweiterte zeitbasierte Funktionen. Eingesetzt wird InfluxDB vor allem für die Speicherung von Messwerten von Sensoren (Stichwort IoT) sowie für Logdaten.

InfluxDB unterstützt, wie die meisten Zeitreihendatenbanken, **Retention Policies**, damit kann festgelegt werden, ob Daten nach einer gewissen Zeit automatisch gelöscht werden sollen und wie die Daten gespeichert werden sollen. Mit **Downsampling** können Daten aggregiert werden um Speicherplatz zu sparen, beispielsweise kann für Messwerte die alle 10 Sekunden erhoben werden nur noch der Mittelwert über 5 Minuten Intervalle gespeichert werden. Daneben gibt es noch umfangreiche Spezialfunktionen zur Abfrage und Auswertung von Zeitreihen.

⁵ <https://github.com/spotahome/redis-operator>

Die Datenhaltung erfolgt standardmäßig im Storage mit einem In-Memory Index, wobei durch die Verwendung einer für das Datenmodell optimierten Speicherung eine sehr hohe Performance erreicht wird.

Entwickelt wird InfluxDB vom Unternehmen InfluxData. Es gibt eine kostenlose Open Source Version (MIT Lizenz) und eine kostenpflichtige Enterprise Version, welche zusätzliche Funktionen und Support beinhaltet.

5.3.1 Datenmodell

Eine Datenbank in InfluxDB beinhaltet ein oder mehrere Zeitreihen und ist schemafrei. Datensätze werden in sogenannten Measurements gruppiert, wobei jeder Datensatz genau einem **Measurement** zugeordnet ist. Ein Datensatz verfügt daneben immer über einen **Zeitstempel**, sowie über ein oder mehrere sogenannte **Fields**, optional können auch noch ein oder mehrere **Tags** vorhanden sein. Fields und Tags bestehen jeweils aus Key-Value Paaren. Tags beinhalten üblicherweise Metadaten und können nur Stringwerte als Value haben, Fields beinhalten Metrikwerte oder Messdaten und können auch Integer, Float oder Boolean Werte haben. Tags werden automatisch indiziert, Values nicht, so dass bei Abfragen immer alle Werte durchsucht werden müssen. Datensätze mit identischem Measurement, identischen Tags und identischem Field Key bilden jeweils eine Zeitreihe.

Beispiel

Measurement: Humidity

time	value	station	city
1574948990000	0.81	n01	Munich
1574949290000	0.81	n01	Munich
1574949590000	0.82	n01	Munich

Abbildung 5.3: Datenmodell InfluxDB

Der Datenzugriff erfolgt mittels HTTP-API und einer an SQL angelehnten Syntax zur Abfrage. Anfragen können direkt mit einem HTTP-Client geschickt werden, mit einer der für zahlreiche Programmiersprachen verfügbaren Bibliotheken oder mit dem `influx` Kommandozeilentool von InfluxDB.

Der Zeitstempel kann, wie im folgenden Beispiel, entweder explizit mitgegeben werden oder er wird von der Datenbank auf die aktuelle Systemzeit gesetzt.

Einen Datensatz schreiben

```
curl -i -XPOST \  
  'http://example.com/write?db=weather&precision=ms' \  
  --data-binary 'humidity,station=n01,city=Munich value=0.81 \  
  1574948990000'
```

- humidity: Measurement
- station, city: Tags
- value: Field
- 1574948990000: Zeitstempel in Millisekunden

Einen Datensatz lesen

```
curl -G 'http://example.com/query?pretty=true&db=weather' \  
  --data-urlencode \  
  "q=SELECT \"value\" FROM \"humidity\" WHERE \"city\"='Munich'"
```

5.3.2 Betrieb

Die Open Source Variante von InfluxDB ist nur als Standalone Instanz betreibbar. Die Enterprise Variante bietet die Möglichkeit einen **Cluster** zu betreiben bei dem Daten, Schreib- und Lesezugriffe über mehrere Knoten verteilt werden (Sharding) und auch repliziert werden können.

InfluxDB beinhaltet Tools für **Backup** und Recovery der kompletten Datenbank. InfluxDB Enterprise bietet darüber hinaus die Möglichkeit inkrementelle Backups zu erstellen, welche nur die Änderungen seit dem letzten Backup beinhalten. Ein

kontinuierliches Wegschreiben der Änderungen und Point-in-Time Recovery ist mit Boardmitteln nicht möglich.

InfluxData bietet mit InfluxDB Cloud einen **Managed Service** an. Daneben ist InfluxDB bei AWS und Google Cloud über den jeweiligen Marketplace verfügbar, für Azure ist das geplant.

InfluxData rät aktuell davon ab, InfluxDB in **Kubernetes** zu betreiben und empfiehlt den Betrieb auf virtuellen Maschinen. Sie stellen aber eine experimentelle Implementierung eines Kubernetes Operators sowie Helm Charts zur Verfügung.

5.4 Graphdatenbank: Neo4j

<https://neo4j.com>

Neo4j ist eine Graphdatenbank, d.h. ihr Datenmodell basiert auf Graphen, also auf Knoten und Verbindungen zwischen diesen Knoten (Kanten). Eingesetzt werden kann sie für alle Anwendungsfälle in denen Beziehungen zwischen Objekten im Mittelpunkt stehen beispielsweise für die **Abbildung von Wissensgraphen (Asset Management, Content Management)**, **soziale Graphen**, **Infrastrukturmonitoring oder Empfehlungssysteme**.

Es speichert Daten auf der Platte, verfügt aber über einen sogenannten *Page Cache*, welcher Indizes und Graphdaten cached.

Neo4j wird vom gleichnamigen Unternehmen entwickelt. Es gibt eine GPL3 lizenzierte Community Edition, sowie eine kostenpflichtigen Enterprise Edition. Letztere ergänzt unter anderem Funktionen im Bereich Backup und Clustering.

5.4.1 Datenmodell

Das Datenmodell von Neo4j ist ein Graph. Dieser besteht aus **Knoten** und **Kanten**. Beide haben **Properties** in Form von Key-Value Paaren, wobei die Werte vom Typ Zahl, String, Boolean, Point (Spatial), Zeit/Datum oder auch ein Array dieser Typen sein können. Knoten haben außerdem ein oder mehrere **Label**. Label

können genutzt werden um Knoten mit ähnlichen Eigenschaften zu gruppieren. Sie können z.B. genutzt werden um Constraints innerhalb einer Gruppe von Knoten zu gewährleisten und können jederzeit hinzugefügt und entfernt werden. Im untenstehenden Beispiel gibt es die Label “Project” und “Consultant”. Kanten haben einen Start- und einen Endknoten, zeigen in eine Richtung und verfügen neben den Properties über genau einen **Relationship Type**, der analog zu Labeln dazu dient Kanten zu gruppieren. Im Beispiel ist dies “EINGESETZT_IN”.

Beispiel

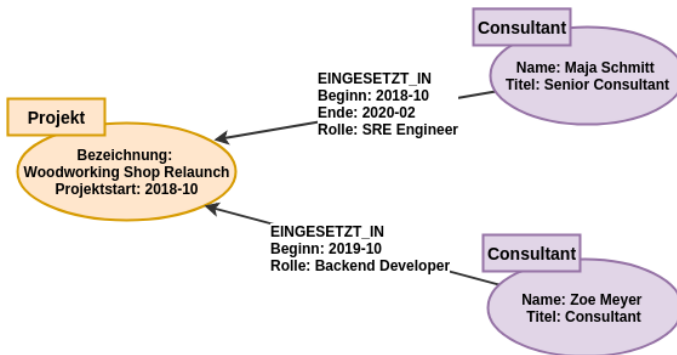


Abbildung 5.4: Datenmodell neo4j

Anfragen werden mit der Neo4j eigenen Sprache **Cypher Query Language** gestellt, die an SQL angelehnt ist, aber auf Graphabfragen ausgelegt ist. Als Protokoll können dabei HTTP oder Bolt verwendet werden. Bei Bolt⁶ handelt es sich um ein von Neo4j entwickeltes Binärprotokoll welches über TCP oder Websockets abgewickelt wird. Durch das kompaktere Format bringt die Verwendung von Bolt Performancevorteile mit sich.

Knoten und Kante anlegen (mit Cypher)

⁶<https://boltprotocol.org>

```
CREATE (shopRelaunch:Project { \
    title:'WoodWorking Shop Relaunch',
    start:'2018-10'})
CREATE (mschmitt:Consultant {name:'Maja Schmitt',
                             title:'Senior Consultant'})
CREATE (mschmitt)-
[:EINGESETZT_IN {role:'SRE Engineer', start:'2018-10'}]->
(shopRelaunch)
```

Einen Datensatz lesen (mit Cypher)

```
MATCH (c:Consultant)-[:EINGESETZT_IN]-(projects)
WHERE c.name = 'Maja Schmitt' RETURN projects
```

5.4.2 Betriebsoptionen

Mit der Community Version kann nur ein einzelner Knoten betrieben werden, die Enterprise Version beinhaltet auch ein **Clustering** Modul namens Causal Clustering. Dieses unterstützt sowohl Multi Leader Replikation wie auch Read Replicas.

Die Community Version unterstützt im Gegensatz zur Enterprise Version keine Hot-**Backups**, d.h. die Datenbank muss zum Erstellen eines Backups gestoppt werden. Die Enterprise Edition unterstützt auch inkrementelle Backups. Das kontinuierliche Wegsichern von Änderungen und Point-in-Time Recovery wird aktuell allerdings nicht unterstützt.

Neo4j bietet mit Neo4j Aura einen eigenen **Managed Service** an. Für die Google Cloud ist ein solcher geplant. Daneben stellt Neo4j mehrere Setup Optionen über die Marketplaces von AWS, Google Cloud und Azure bereit.

Einen offiziellen Kubernetes Operator gibt es bislang nicht, die existierenden Community Operatoren wirken wenig ausgereift. Es gibt aber Helm Charts im offiziellen Helm Repository.

5.5 NewSQL Datenbank: CockroachDB

<https://www.cockroachlabs.com>

CockroachDB ist eine NewSQL Datenbank. Wie bei einer klassischen relationalen Datenbank werden die Daten in Tabellenform gespeichert. Sie bietet ihnen gegenüber aber bessere **horizontale Skalierungsmöglichkeiten** und wird deshalb für Anwendungen mit hohen Schreib-/Leseanforderungen oder geografischer Verteilung eingesetzt. Sie unterstützt **Transaktionen und starke Konsistenz**.

CockroachDB unterstützt sowohl eine Datenhaltung auf Storage wie auch im Arbeitsspeicher. Dies ist beim Starten der Datenbank konfigurierbar. Weiterhin gibt es noch die Möglichkeit einen konfigurierbar großen Cache zu verwenden.

CockroachDB ist eine frei verfügbare Datenbank, deren Quellcode offen liegt. Sie steht unter der Business Source License die auch von MariaDB verwendet wird. Diese Lizenz wurde gewählt, um Cloudanbietern wie AWS zu verbieten ohne vertragliche Vereinbarung, eine kommerzielle Version von CockroachDB als Service anzubieten.

5.5.1 Datenmodell

CockroachDB unterstützt wie PostgreSQL ein relationales, tabellenbasiertes Datenmodell. Es unterstützt neben Datentypen für Boolesche Werte, Zahlen, Zeichenfolgen und Datumsangaben auch JSON und UUID.

Soll eine PostgreSQL Datenbank nach Cockroach importiert werden, können die Datentypen automatisch oder explizit gemappt werden.

Beispiel

CockroachDB unterstützt SQL. Das verwendete Datenübertragungsprotokoll ist bis auf wenige Ausnahmen kompatibel zu PostgreSQL, so dass PostgreSQL Client Tools, Treiber und OR-Mapper vielen Fällen weiter verwendet werden können. Inkompatibilitäten sind auf der Webseite von CockroachDB dokumentiert.

Soll eine PostgreSQL Datenbank nach Cockroach importiert werden, können die Datentypen automatisch oder explizit gemappt werden.

id	title	abstract	trainer	begin_time	end_time
de4af130-392b-11ea-a137-2e728ce88125	Handhabung einer Oberfräse	...	3c9195f0-392c-11ea-a137-2e728ce88125	2020-01-23 10:00:00	2020-01-23 17:00:00
de4af3c4-392b-11ea-a137-2e728ce88125	Arbeiten mit dem Handhobel	...	3c9195f0-392c-11ea-a137-2e728ce88125	2020-01-24 09:00:00	2020-01-24 19:00:00
de4af52c-392b-11ea-a137-2e728ce88125	Schärfen von Werkzeugen	...	3c919988-392c-11ea-a137-2e728ce88125	2020-01-25 09:30:00	2020-01-25 17:00:00

Abbildung 5.5: Datenmodell CockroachDB

Einen Datensatz schreiben

```
INSERT INTO woodwork.course (id, title, abstract, trainer,
    begin_time, end_time)
VALUES ( 'de4af130-392b-11ea-a137-2e728ce88125',
        'Handhabung einer Oberfräse',
        'Rückwandfalze, Zierprofile und \
        Bilderrahmenleisten',
        'bfa1073a-f459-4ad7-97dd-b98d164aeb73',
        '2020-01-23 10:00:00',
        '2020-01-23 17:00:00');
```

Einen Datensatz lesen

```
SELECT c.title FROM woodwork.course c, woodwork.trainer t
    WHERE t.name='Florian Schreiner';
```

5.5.2 Betriebsoptionen

Traditionell ist es für ein Datenbanksetup mit hoher Verfügbarkeit erforderlich:

1. Explizit Knoten als Leader oder Follower zu deklarieren.
2. Einen Watchdog oder ähnliches aufzusetzen, der bei Ausfall des Leaders die Wahl eines neuen Leaders anstößt.
3. Falls ein Knoten nach einem Ausfall wieder erreichbar ist, den Knoten wieder auf den aktuellen Stand zu bringen, indem wir ihn als Follower konfigurieren.

Bei CockroachDB ist das anders. Wir fahren auf jedem Knoten eine Instanz der Datenbank hoch und konfigurieren nur, wo die anderen Knoten zu finden sind. Die Knoten organisieren sich anschließend selbst und befreien die Administrator*innen von den oben genannten Arbeitsschritten. Fügen wir weitere Knoten hinzu oder ändern die Replikationszahl, kümmert sich die Datenbank automatisch um das Umverteilen der Knoten, um unsere Anforderungen zu erfüllen (*Automatic Rebalancing*).

CockroachDB beinhaltet Backup und Recovery Tools, welches volle und inkrementelle Backups sowie **Point-in-Time Restores** unterstützen.

In der Google Cloud und AWS kann CockroachDB über die jeweiligen **Marketplaces** installiert werden. Cockroach Labs plant außerdem mit **CockroachCloud** einen eigenen Managed Service anzubieten, dieser befindet sich momentan in der geschlossenen Betaphase.

Von CockroachDB selbst wird kein Kubernetes Operator angeboten, dafür allerdings **Helm Charts**.

5.6 Vektordatenbanken: Moderne Suche für semantische Ähnlichkeiten

Verschiedene Tools und Anbieter, u.a. PostgreSQL (pgvector), MongoDB, AWS Kendra, ChromaDB, Elastic Vector

Vektordatenbanken stellen eine neue Art der Informationssuche dar, die auf semantische Ähnlichkeiten zwischen Daten fokussiert ist, statt exakter Schlüsselwort-Matches. Sie bieten Lösungen für Anwendungsfälle wie Produktsuche, Produktempfehlungen oder das Finden semantisch ähnlicher Inhalte, z.B. in Blogs oder juristischen Dokumenten.

Anders als klassische Suchmaschinen speichern sie keine Texte oder Begriffe direkt, sondern numerische Repräsentationen – sogenannte Vektoren. Diese repräsentieren die Bedeutung von Inhalten und ermöglichen die Suche basierend auf

konzeptuellen Zusammenhängen, unabhängig von der konkreten Formulierung einer Suchanfrage.

5.6.1 Datenmodell

Eine Vektordatenbank verwendet Vektoren, die aus Texten oder anderen Datenquellen mithilfe vortrainierter Embedding-Modelle wie Word2Vec oder text-embedding-ada erzeugt werden. Die Vektoren werden in einem hochdimensionalen Raum gespeichert. Bei einer Suchanfrage wird die Eingabe ebenfalls mit dem gleichen Embedding-Modell in einen Vektor umgewandelt und die Ähnlichkeit zwischen diesem Vektor und denen in der Datenbank wird berechnet. Eine Vektordatenbank liefert immer Ergebnisse, was einen weiteren Unterschied zu bisherigen Datenmodellen darstellt.

Beispiel: Datenmodell

Ein Beispiel zeigt, wie Produktdaten in einer Vektordatenbank gespeichert werden könnten:

1. **Input-Daten:** JSON-Dokument mit Produktname, Beschreibung und Kategorien.
2. **Embedding-Prozess:** Umwandlung dieser Daten in numerische Vektoren.
3. **Suche:** Berechnung der semantischen Ähnlichkeit zwischen dem Anfragevektor und gespeicherten Vektoren.

Vektor-Speicherung in PostgreSQL mit pgvector

```
CREATE TABLE products (  
    id SERIAL PRIMARY KEY,  
    name TEXT,  
    description TEXT,  
    embedding VECTOR(300)  
);
```

Die Zeile `embedding VECTOR(300)` beschreibt die Spalte in welcher die Vektoren abgelegt werden. Die Vektoren haben dabei jeweils 300 Einträge.

```
-- Einen Vektor speichern
INSERT INTO products (name, description, embedding)
VALUES ('Laptop XY', 'Leistungsstarker Laptop für Office und
        Gaming',
        '[0.234, 0.567, ..., 0.789]');
```

Vektor-Suche

```
SELECT name, description
FROM products
ORDER BY embedding <-> '[0.234, 0.567, ..., 0.789]'
LIMIT 5;
```

Diese Suchanfrage gibt die fünf semantisch ähnlichsten Produkte aus der Vektordatenbank zurück. Das hier verwendete Embedding [0.234, 0.567, ..., 0.789] wurde dabei aus der Nutzereingabe berechnet.

5.6.2 Betrieb

Der Betrieb von Vektordatenbanken erfordert eine Infrastruktur, die große Datenmengen und schnelle Abfragen im hochdimensionalen Raum bewältigen kann. Ähnlich wie bei indexbasierten Suchsystemen können auch Vektordatenbanken verteilt betrieben werden. Dies geschieht durch die Aufteilung des Vektorraums in kleinere Teilräume (z.B. Locality-Sensitive-Hashes).

Verfügbare Tools und Anbieter:

- **Elasticsearch:** Unterstützt klassische und Vektorsuchen; ideal für hybride Suchstrategien.
- **PostgreSQL mit pgvector:** Open-Source-Erweiterung für Vektorsuche in relationalen Datenbanken.
- **MongoDB:** Relationale Datenbank, welche auch Vektorsuche unterstützt
- **ChromaDB:** In-memory Datenbank für schnelle Tests und Prototyping.
- **Cloud-Services:** AWS Kendra, Azure Cognitive Search und Google Cloud bieten integrierte Lösungen.

Backups und Skalierbarkeit: Für Backups können klassische Datenbanksicherungsstrategien wie Snapshots oder das kontinuierliche Sichern von Änderungen genutzt werden. Die Skalierbarkeit hängt von der gewählten Implementierung ab. Einige Systeme unterstützen automatisierte Partitionierung und Verteilung des Vektorraums.

5.6.3 Anwendungsfälle

1. **Produktsuche:** Suche nach ähnlichen Produkten basierend auf Eigenschaften und Beschreibungen.
2. **Dokumentenmanagement:** Präzise Suche in Textabschnitten, z.B. in juristischen Akten.
3. **Produktempfehlungen:** Vorschläge für ähnliche Produkte, z.B. in Online-Shops.

5.6.4 Fazit

Vektordatenbanken sind besonders geeignet, wenn semantische Ähnlichkeit eine Rolle spielt. Durch die Kombination mit klassischen Suchsystemen können sie vielseitige Sucherlebnisse schaffen. Dank Open-Source-Tools wie PostgreSQL pgvector und leicht zugänglicher Cloud-Services ist der Einstieg einfach und flexibel.

6 Auswahl einer Datenbank

Die Auswahlkriterien für die Wahl einer Datenbank werden wesentlich durch die **Anforderungen der Anwendung** bestimmt, für welche die Datenbank verwendet werden soll:

- Welche Konsistenzanforderungen bestehen? Ist beispielsweise bei einer Single Leader Replikation eine asynchrone Replikation mit einem Replication Lag von mehreren Minuten OK?
- Welche Arten von Abfragen und wie viele Schreib- und Lesezugriffe gibt es?
- Wie sind die Daten strukturiert? Welche Arten von Daten liegen vor?
- Gibt es Analytics Workloads bei denen eine Vielzahl von Aggregatsfunktionen auf dem Datenbestand ausgeführt wird?
- Ist eine verteilte Datenhaltung erforderlich? Beispielsweise weil Daten auch auf Mobilegeräten gespeichert und geändert werden?

Nicht außer Acht gelassen werden sollte aber auch die **Projektumgebung**: Die Risiken einer neu zu implementierenden Anwendung hinsichtlich Kosten, Umsetzungsdauer und Wartungsfreundlichkeit korrelieren mit der **Anzahl an Unbekanntem** oder Neuem in einem Projekt. Je größer die Unbekannten auf der Businessseite – Businessmodell, Funktionsumfang, unerfahrene Leitung – desto “langweiliger” sollten die verwendeten Technologien sein – inklusive der Datenbank. Umgekehrt bietet ein Projekt bei dem wenig Unwägbarkeiten auf der Businessseite bestehen, eine gute Möglichkeit Erfahrung mit neuen Technologien und eben eventuell auch einer neuen Datenbank zu sammeln.

Neben den klassischen Auswahlkriterien ist in den letzten Jahren die Frage der **semantischen Suche** hinzugekommen. Während traditionelle Datenbanken sehr gut für exakte Übereinstimmungen (exact matches) und strukturierte Abfragen geeignet sind, erfordern moderne Anwendungsfälle oft die Suche nach ähnlichen Inhalten. Beispiele hierfür sind:

- Produktempfehlungen basierend auf Ähnlichkeit
- Semantische Dokumentensuche in großen Textbeständen
- Chatbots und AI-gestützte Assistenzsysteme
- Bildsuche nach ähnlichen Motiven

In diesen Fällen bieten sich Vektordatenbanken an. Dabei gilt es abzuwägen:

- Ist die semantische Ähnlichkeitssuche ein Kernfeature der Anwendung oder nur eine Ergänzung?
- Rechtfertigt die Größe des Datenbestands eine spezialisierte Lösung?
- Stehen ausreichend Ressourcen für die Embedding-Generierung zur Verfügung?
- Können die erhöhten Infrastrukturkosten (GPU) durch den Mehrwert gerechtfertigt werden?

Auch wenn zahlreiche anwendungs- und projektspezifische Überlegungen die Hauptrolle bei der Auswahl einer Datenbank spielen, haben wir uns an einem einfachen, allgemeinen Auswahlschema für Datenbanken versucht:

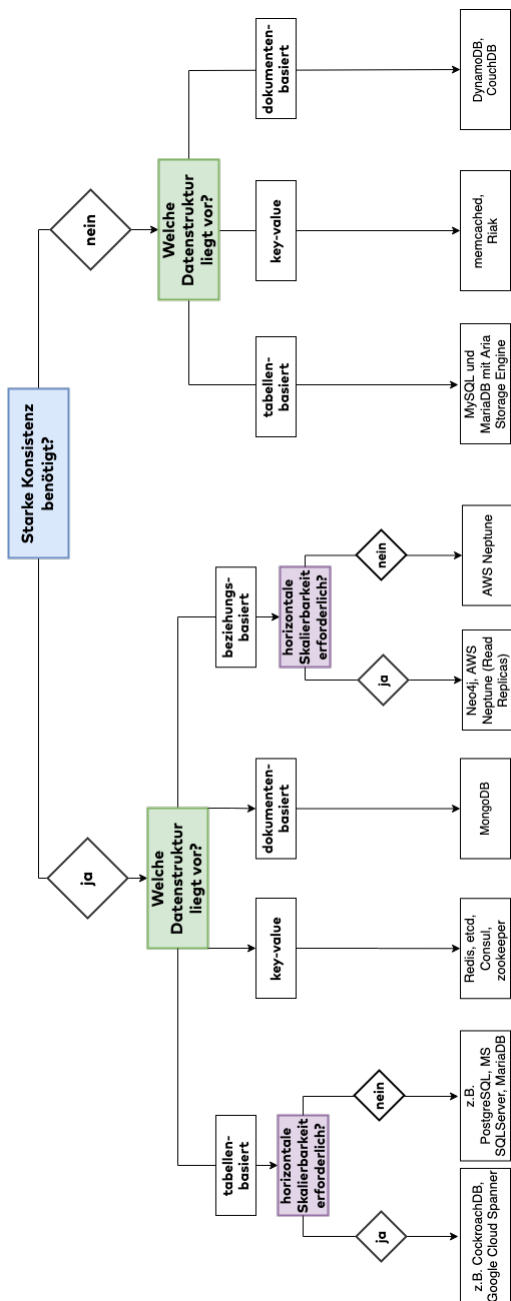


Abbildung 6.1: Auswahl einer Datenbank

7 Unser Angebot

Eine passgenaue Datenbank und eine entsprechende Modellierung Ihrer Daten ermöglichen Ihnen, langlebige, performante Anwendungen zu implementieren und Aufwände gering zu halten.

Wir verfügen über viele Jahre Erfahrung in der Auswahl von Datenbanken, der Erstellung zugehöriger Datenmodelle, sowie der nachfolgenden Evolution und, falls notwendig, Migration oder Performanceoptimierung bestehender Systeme.

Gerne erstellen wir mit Ihnen **gemeinsam** entsprechende Konzepte oder bieten konkrete Unterstützung. Bei Bedarf konzipieren wir auch gerne individuelle Schulungen entsprechend Ihrer Anforderungen.

Weitere Infos zu unseren Beratungsleistungen und Schulungen finden Sie unter www.innoq.com oder Sie schreiben uns an: info@innnoq.com



Wir beraten ehrlich, denken innovativ und entwickeln leidenschaftlich gern. Das Ergebnis: Erfolgreiche Softwarelösungen, Infrastrukturen und Geschäftsmodelle.

Als Technologieunternehmen fokussieren wir uns auf Strategie- und Technologieberatung, Softwarearchitektur und -entwicklung, Methoden- und Technologietraining sowie Plattform-Infrastrukturen.

Wir unterstützen mit über 150 Mitarbeiter*innen an Standorten in Deutschland und der Schweiz Unternehmen und Organisationen bei der Konzeption und Umsetzung komplexer Vorhaben und der Verbesserung bestehender Softwaresysteme.

Wir engagieren uns in Open-Source-Projekten und dem iSAQB e.V. Wir teilen unser Wissen und unsere Erfahrung auf Konferenzen und Meetups sowie in zahlreichen Büchern und Fachartikeln.

8 Quellen

- [1] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly, 2017.
- [2] B. Beyer, C. Jones, J. Petoff, und N. R. Murphy, *Site Reliability Engineering*. O'Reilly, 2016 [Online]. Verfügbar unter: <https://landing.google.com/sre/sre-book/toc>
- [3] B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, und S. Thorne, *The Site Reliability Workbook*. O'Reilly, 2018 [Online]. Verfügbar unter: <https://landing.google.com/sre/workbook/toc>
- [4] L. Campbell und C. Majors, *Database Reliability Engineering: Designing and Operating Resilient Database Systems*. O'Reilly, 2017.

Über die Autor*innen



Lucas Dohmen

Lucas war bis August 2023 Senior Consultant bei INNOQ und beschäftigt sich mit der Architektur, Konzeption und Umsetzung von Web Anwendungen in Front- und Backend. Er programmiert in Ruby und JavaScript und hilft bei der Entscheidung für verschiedene NoSQL Lösungen. Außerhalb seiner Arbeit beschäftigt er sich mit Open Source und Community Arbeit (z. B. Organisation und Coaching beim lokalen CoderDojo) und ist Teil des Podcasts Nerdkunde. [🐦 @moonbeamlabs](#)



Christine Koppelt

Christine Koppelt arbeitet als Senior Consultant bei der INNOQ, wo sie sich vorwiegend mit der Realisierung von Digitalisierungsprojekten für mittelständische Unternehmen beschäftigt. Ihre Schwerpunkte sind die Themen DevOps, Data Engineering und Machine Learning. Sie ist Organisatorin des Munich Data Engineering Meetups. [🐦 @ckoppelt](#)



Robert Glaser

Robert Glaser ist Head of Data & AI bei INNOQ. Mit einem Hintergrund im Software Engineering und einer Leidenschaft für benutzerfreundliche Webanwendungen begleitet er heute Unternehmen durch die KI-Landschaft. Dabei unterstützt er sie bei der Entwicklung von Strategien und Produkten für schwierige technische Herausforderungen. Fasziniert von der Vielseitigkeit generativer KI, moderiert er den Podcast „AI und jetzt“. Ihm ist die Brücke zwischen Technologie und Business ein Herzensanliegen. In seiner Freizeit erkundet er die lokale Food Szene.



Marco Steinke

Marco Steinke ist Consultant bei INNOQ. Sein Schwerpunkt liegt in der Software-Architektur. Außerdem beschäftigt er sich mit künstlicher Intelligenz und dabei vor allem mit der Architektur und der Integration von KI-Systemen.



Entwickler:innen haben heute eine Vielzahl von Optionen, wenn es um die Wahl einer Datenbank für ihre Anwendung geht.

Dieses Buch gibt einen Überblick über Datenmodelle, Konzepte zur Skalierung und Betriebsformen – von relationalen Systemen über dokumenten- und graphbasierte Ansätze bis hin zu Zeitreihen- und Vektordatenbanken. Es bietet Hilfestellung bei der Auswahl und Einordnung für unterschiedliche Anwendungsszenarien.

innoq.com