FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Fachhochschule Aachen, Campus Jülich

Fachbereich 09 - Medizintechnik und Technomathematik
Studiengang Angewandte Mathematik und Informatik

Seminararbeit

Datenbanken für Microservices

Vergleich von Datenbankkonzepten zur Erfüllung der Anforderungen einer modernen Architektur

Autor:
Tom Schönijahn, 3520495

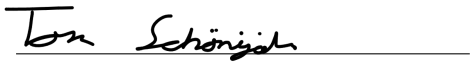
Betreuer:
Prof. Dr. rer. nat. Volker Sander
Lars Frauenrath, B.Sc.

Aachen, 4. Januar 2024

Eidesstatliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema *Datenbanken für Microservices* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.
Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Aachen, den 4. Januar 2024



Tom Schönijahn

Zusammenfassung

Die Softwareentwicklung in Unternehmen konzentrierte sich lange Zeit auf die Erstellung von monolithischen Softwareanwendungen, welche mit steigender Komplexität und Abhängigkeiten immer aufwändiger wurden. Microservice-Architekturen wurden entwickelt, um diesen Herausforderungen zu begegnen. Sie ermöglichen die Aufteilung von Funktionalitäten in verschiedene Services, die unabhängig voneinander programmiert und bereitgestellt werden können. In diesem Kontext entstanden unterschiedliche Architekturprinzipien und Design Patterns. Zentrale Konzepte sind dabei das Single-Responsibility-Principle, dass die Aufteilung von Funktionalitäten in separate Services fördert, sowie die klare Definition von Grenzen und die Portabilität von Microservices durch Container fordert. Darüber hinaus sollten Microservices ihre eigenen Daten verwalten und die Fähigkeit zur Skalierung und Automation bieten. Die Datenbanken müssen ebenfalls an moderne Infrastrukturen, die oft Container verwenden, angepasst werden. Es gibt verschiedene Ansätze zur Datenbankgestaltung. Dabei sind vor allem die Ideen "Database per Service" und "Shared Database"' verbreitet. Ähnlich zu den Microservices sollten die Datenbanken als verteilte Systeme betrachtet werden, wodurch das CAP-Theorem eine größere Bedeutung erhält. Darüber hinaus werden weiterhin bekannte Anforderungen an die Datenbanken und Datenbankmodelle gestellt, bei denen insbesondere das ACID-Prinzip und die Normalformen zu nennen sind. Betrachtet wird neben den Relationalen auch eine Auswahl von NoSQL Datenbanken. Dazu zählen sowohl Tabellenstrukturen in Form von Column-Based-Database, einfachere Ordnungen, beispielsweise Key-Value-Strukturen, oder neue Konzepte, wie die der Graphendatenbank. Ein weiterer Punkt der Betrachtung stellt das Speichersystem dar, bei welchem zwischen In-Memory-Systemen und On-Disk-Speichern unterschieden werden muss.

Die Untersuchung der Fragestellung nach Kriterien für die Auswahl einer Datenbankvariante zeigt, dass einfache Anforderungen von allen Systemen hinreichend erfüllt werden können. Dies macht es insbesondere in der Konzeptionsphase schwierig, die optimale Software auszuwählen. Meist müssen spezifische Aspekte der Applikation oder geplante zukünftige Verwendungen bedacht werden. Neben den theoretischen Entscheidungsfaktoren ist, aus unternehmerischer Sicht, immer eine Betrachtung der Kompetenzen innerhalb der Softwareentwickler notwendig und dadurch eine konkrete Auswertung und Erweiterung dieser Erkenntnisse erforderlich.

Inhaltsverzeichnis

1 Einleitung

In den letzten Jahren hat sich die Softwareentwicklung in vielen Aspekten weiterentwickelt, wobei die Einführung von Microservices-Architekturen als eine der zentralsten Veränderungen bekannt ist. Diese Herangehensweise an die Strukturierung von Softwareanwendungen in eigenständige Dienste hat die Methoden der Softwareentwicklung, Bereitstellung und Skalierung weitgehend verändert. Allerdings gehen mit diesen Fortschritten auch komplexe Herausforderungen in Bezug auf die Datenverwaltung einher.

Die Auswahl geeigneter Datenbanken für Microservices stellt einen zentralen Aspekt des Architekturdesigns dar, da sie maßgeblichen Einfluss auf die Leistungsfähigkeit, Skalierbarkeit, Datensicherheit und Wartbarkeit einer Anwendung hat. Ziel dieser Seminararbeit ist es, eine Betrachtung des Themas Microservices und Datenbanken vorzunehmen, um die grundlegenden Prinzipien, Anforderungen und Herausforderungen bei der Auswahl der richtigen Datenbanktechnologie für Microservices zu erörtern.

Es werden verschiedene Arten von Datenbanksystemen analysiert, beginnend mit relationalen Datenbanken und weiterführend zu den vielfältigen Optionen im Bereich NoSQL. Besonderes Augenmerk liegt auf den maßgeblichen Kriterien für die Auswahl einer individuellen, geeigneten Datenbank.

Im Verlauf dieser Ausarbeitung werden diese Themen vertiefend behandelt. Zum Abschluss dieses Beitrags sollen die Grundlagen geschaffen sein, fundierte Entscheidungen bei der Auswahl von Datenbanken für Microservices-Anwendungen zu treffen. Diese Entscheidungen werden nicht nur von Entwicklern und Softwarearchitekten getroffen, sondern bedürfen, aufgrund der entscheidenden Rolle bei der Gewährleistung einer effizienten und zukunftsorientierten Softwareentwicklung, der Integration aller Stakeholder, die möglicherweise weniger technisch orientiert sind.

2 Microservice Architekturen

Die Softwareentwicklung in Unternehmen hatte lange Zeit als Ziel, eine Software zu entwerfen und zu programmieren, welche alle Aufgaben erfüllen kann. Bei diesem Konzept entstehen sogenannte monolithische Softwareapplikationen oder kurz gesagt Monolithen. Mit einer wachsenden Anzahl an Aufgaben, Funktionen und Nutzern der Softwareprodukte ergaben sich verschiedene Probleme mit dieser Architektur. Zum einen wird der Code immer komplexer und zum anderen steigen die inneren Abhängigkeiten und Verknüpfungen an. Dies macht es für die Entwickler schwer einen Überblick über alle Facetten der Applikation zu behalten. Ein weiterer Nachteil der Monolithen ist die Tatsache, dass die meisten Installationen nur von einer einzigen oder wenigen Instanzen ausgehen und meist eine hohe Kopplung untereinander benötigen. Daraus resultiert die Notwendigkeit der vertikalen Skalierung, welche jedoch durch extrem leistungsfähige Hardware kostenintensiv ist und außerdem die Grenze des Möglichen erreichen kann. Microservices sind als Konzept entstanden, um diesen Problemen bei wachsenden Systemlandschaften zu begegnen. Eine zentrale Eigenschaft bildet die Möglichkeit, Funktionalitäten und damit Code auf verschiedene Projekte aufzuteilen. Dies erleichtert die Entwicklung und insbesondere die parallele Bearbeitung durch mehrere Programmierer. Als Schnittstelle zwischen den unterschiedlichen Microservices dient eine API, welche meistens in Form einer Web-API bereitgestellt wird. Solange diese unverändert bleibt, können die Services unabhängig voneinander angepasst, entwickelt und ausgerollt werden.

2.1 Architekturprinzipien

Architekturprinzipen, die ebenfalls als Entwurfsmuster bezeichnet werden, stellen ein kontroverses Thema der Softwareentwicklung dar. Sie geben einige Ideen und Entwürfe für die Konzeption und Implementation einer Software. In vielen Fällen werden sie bedeutsamer interpretiert und ihnen wird eine eher Dogma-artige Bedeutung zugeschrieben. Dies führt unter Umständen zu einer Umsetzung dieser Muster ohne das sie für die konkrete Software sinnvoll und geeignet sind. Neben den Design Pattern als Zielbilder gibt es Anti-Pattern, welche einige der üblichen Fehlentscheidungen abbilden und somit vermieden werden sollten. Die bekanntesten Entwurfsmuster der Softwareentwicklung beziehen sich auf objektorientierte Programmierung sowie die daraus entstehende Software. Sie wurden im Jahr 1994 von der “Gang of Four“ (GOF) in dem Buch “Design Patterns: Elements of Reusable Object-Oriented Software“ gesammelt und veröffentlicht. Seit diesem Zeitpunkt haben sie sich nahezu zum Standardwerk für Pattern in objektorientierter Softwareentwicklung weiterentwickelt. Außerdem entstanden im Laufe der Zeit viele Ideen, um diese Inhalte zu erweitern. [24] Das Konzept des Patterns und Anti-Patterns hat sich seit dem in weiteren Bereichen der Softwareentwicklung etabliert. Einige neue Ansätze beziehen sich nicht mehr nur auf die Erstellung des Codes, sondern auch auf übergreifende Komponenten, wie den Workflow der Entwickler oder die

Infrastruktur. Für die weitere Betrachtung sind insbesondere jene Design Pattern interessant, welche speziell für Microservices entstanden sind. Ein Großteil der Listen für diese wird von verschiedenen Communitys gepflegt, wodurch keine einheitlichen Definitionen existieren. Auf die unterschiedlichen Publikationen zu diesem Thema sollte sich aufgrund der Schnelllebigkeit dieser Architektur nicht ausschließlich verlassen werden.

2.1.1 Microservices

Das wichtigste Pattern in der Entwicklung von Microservices ist das Pattern der Microservices selbst. Es handelt sich bei diesen um ein Architekturmuster [13], weshalb damit wie mit allen anderen bekannten Design Patterns verfahren und diese nicht als Dogma gesetzt werden sollten.

2.1.2 Single Responsibility Principle

Das Pattern "Separation of Concerns" ist eines der fundamentalen Entwurfsmuster in der Softwareentwicklung und nicht ausschließlich auf Microservices beschränkt. Es beschreibt die Aufteilung von Funktionalitäten und den Aufbau der Applikation in mehreren kleinen Teilaspekten. Diese sind in der Regel einfacher zu entwickeln und zu verwalten als eine große, verwobene Komponente. [11] Eine Weiterentwicklung dieser Idee zeigt sich im ersten Aspekt des SOLID-Patterns, dem "Single Responsibility Principle". Services, die nach diesem Pattern definiert werden, begrenzen ihre Funktionalität auf einen Teilbereich der Software. Dadurch wird aus jeder Zuständigkeit innerhalb der Planungsphase ein eigener Service gebildet wird anstelle einer reinen Trennung des Codes in der monolithischen Applikation. Die nächste Entwicklungsstufe und damit der Grad der Trennung stellt das Konzept "Single Concern Principle" [5] dar. Bei diesem werden die Services nicht mehr auf Ebene der zugehörigen Themen definiert, sondern beziehen sich nur auf eine einzelne Funktionalität, die diese Applikation erfüllen soll. Das Ziel ist, dass ein Service nur genau eine Funktionalität gewährleisten soll. Der Übergang zwischen Single Responsibility Principle und Single Concern Principle ist fließend, weshalb in vielen Dokumentationen nur eine der beiden Varianten aufgeführt ist.

2.1.3 Discrete Boundaries

Ähnlich zu den inhaltlichen Grenzen des Microservice aus dem vorherigen Konzept, sollten auch die physischen klar definiert sein. Insbesondere die Grenze zu der Umgebung sollte eindeutig sein, sodass alle notwendigen Logiken und Daten in einer einzigen Einheit zum Deployment gebündelt werden können. Diese können sowohl sprachspezifisch sein, wie es bei einem Node.js-Package oder einer JAR-Datei der Fall ist, als auch unabhängig sein, wie beispielsweise bei Containern. Die Trennung bezieht sich in den meisten Fällen nicht nur auf die veröffentlichte Version, sondern beginnt bereits bei der Entwicklung durch isolierte Speicherorte. Diese werden mittels eigener Repositories oder eigenen Ordnern in einem Monorepo geschaffen. Fortgeführt wird die Separierung durch spezifische Prozesse zum Beispiel in der CI/CD-Umgebung. [5]

2.1.4 Transportable microservices

Aus der Umsetzung des Prinzips der diskreten Grenzen ergibt sich die Möglichkeit, die Applikation unabhängig von einem konkreten System bereitzustellen und im nächsten Schritt eine einfache Verschiebung zwischen Umgebungen zu ermöglichen. Die meistgenutzt Variante dazu stellen die Container dar. Diese erlauben den Betrieb in vielen verschiedenen Umgebungen, ohne dass die einzelne Komponente für diese Systeme optimiert werden muss. Ein Container beschreibt die laufende Instanz, welche aus dem jeweiligen (Container-) Image erzeugt wird. Dieses beinhaltet alle Notwendigkeiten zum Start der Applikation, unter anderem eine Runtime-Umgebung oder spezielle Treiber. Das Image kann über eine spezielle Art von Repository bereitgestellt werden und anschließend in jede beliebige Umgebung, die dieses Image ausführen kann, kopiert werden. Die Transportfähigkeit der Microservices erlaubt den Einsatz von Mechanismen, die eine Automation in Hinblick auf Deployments erlauben. [5]

2.1.5 Carry-its-own-data

Ähnlich zu der Isolierung der Funktionalitäten in einzelnen Microservices sollen die Daten einem Service zugeordnet werden. Der Austausch von Daten geschieht in diesem Fall mithilfe verschiedener APIs. Häufig kann dieses Pattern im Widerspruch zu dem Ansatz von "Don't repeat yourself" stehen, da gewisse Redundanzen von Daten und Abfragelogiken in den Services notwendig werden. Dieses Konzept meint explizit nicht, dass alle Daten innerhalb der einen Anwendung gespeichert werden müssen. Es besagt nur, dass die Speicherorte exklusiv von einer Applikation genutzt werden. [5]

2.1.6 Inherently ephemeral

Eine Eigenschaft, die durch die Nutzung von Container impliziert wird und auch explizit in der Entwicklung bedacht werden muss, ist die Flüchtigkeit und Kurzlebigkeit der Instanzen. [5] Der Hauptaspekt, in denen dies einen Einfluss hat, ist der Lebenszyklus der Applikation. Eine Instanz dieser muss jederzeit erzeugt und zerstört werden können, ohne einen großen Aufwand zu erzeugen oder Nebeneffekte zu produzieren. In der konkreten Umsetzung erfordert dies zum einen den Umgang mit Start und Stopp Verhalten ohne weitere Einflüsse und zum anderen die Unkenntnis über den sonstigen Zustand der Infrastruktur. Ein wichtiger Aspekt ist Konfiguration der Umgebungsdaten zur Laufzeit, anstelle der Definition in der Entwicklungsphase, wie es bei monolithischen Applikationen möglich wäre.

2.2 Veränderungen der Infrastruktur

Aus dem Entwurfsmuster der Transportfähigkeit ergibt sich in aller Regel eine Bereitstellung der Software innerhalb von isolierten Containern. Diese werden in den meisten Fällen nicht auf einem einzigen Host ausgeführt, sondern nutzen verschiedene Technologien, welche von einem Cluster für diese Software bereitgestellt werden. In vielen Fällen wird für diese Cluster eine Cloud Umgebung genutzt, in der sie als Software-as-a-Service-Produkt eingekauft werden. In diesen geclusterten Umgebungen gibt es, neben den Anforderungen, die von den Microservices

bekannt sind, weitere Kriterien, denen eine Software genügen muss, um sinnvoll eingesetzt werden zu können. Applikationen, die speziell an diese Anforderungen angepasst wurden, werden häufig als Cloud-Native-Applications bezeichnet. [14] Im Folgenden soll eine kleine Auswahl dieser Kriterien vorgestellt werden.

2.2.1 Scalability

Der zu Beginn erwähnte Nachteil vom Monolithen in der begrenzten vertikalen Skalierung, wird in den meisten neuen Infrastrukturen durch eine horizontale Skalierung umgangen. [25] Zur Vorbereitung der Software darauf dient unter anderem das Konzept "Inherently Ephemeral" [5].

2.2.2 Automation

Viele Entwicklungsteams wandeln sich vom klassischen Programmieren zu Teams, die DevOps oder Abwandlungen davon umsetzen. Damit ist gemeint, dass die Entwickler (Developer) zusätzlich am Betrieb der Applikation (Operations) beteiligt sind und entsprechende Einflüsse haben. Zur Vereinfachung der Arbeit wird in vielen Fällen eine möglichst hohe Automation in den Release-Prozessen angestrebt. Stichwörter in diesem Kontext sind CI/CD-Pipelines oder Infrastructure-as-Code. [23] Darüber hinaus gibt es Automationen in den Laufzeitumgebungen, die dafür sorgen können, dass eine fehlerhafte Applikation neu gestartet oder diese bei erhöhter Last durch neue Instanzen besser verteilt wird.

2.3 Rolle von Datenbanken

Die Kernaufgabe der Datenbanken in einem größeren Softwareprojekt ist das Speichern von Datensätzen und hat sich durch den Wechsel von monolithisch orientierter Struktur zu Microservices nicht verändert. Im Gegensatz dazu hat sich die Positionierung der Datenbanken geändert. Früher hat eine einzelne Datenbank den Großteil der Daten eines Projekts gespeichert, während diese heutzutage enger an einzelne Services geknüpft sind. Aufgrund der neuen Infrastrukturen und Umgebungen, in denen Software betrieben wird, verändert sich zugleich die Umgebung, in der die Datenbank betrieben werden. Daraus ergibt sich, dass die Datenbanken diese Anforderungen ebenfalls umsetzen müssen.

3 Anforderungen an Datenbanken

Datenbanken stellen häufig eine zentrale Funktionalität für verschiedenste Softwareprodukte bereit. Zur Auswahl und Klassifizierung der verschiedenen vorhandenen Datenbanksysteme müssen einige konkrete Anforderungen definiert werden. Die meisten Anforderungen an die Datenbank sind lange vor den Microservices entstanden. Diese wurden in der Entwicklung von Microservices durch weitere, feste Anforderungen für diese Struktur ergänzt. Im Folgenden soll eine Auswahl aus sowohl klassischen als auch spezifisch für Microservices geschaffenen Anforderungen erläutert werden, welche zum Teil den Aspekten des vorherigen Kapitels ähneln. Im Allgemeinen ist zu beachten, dass nicht jede Anforderung für alle Softwareanforderungen relevant ist und deshalb hier ein Fokus auf möglichst allgemeingültige Kriterien gelegt wurde.

3.1 Klassische Anforderungen an Datenbanken

Im Jahr 1983 erschufen Andreas Reuter und Theo Härder das Akronym ACID, welches die Anforderungen an Datenbanken und spezifischer an Transaktionen in diesen beschreibt [10]. Eine Transaktion in einer Datenbank stellt eine Abfolge von mehreren logisch gekoppelten Operationen auf den Daten dar. Das Konzept der Transaktion ist nicht in jeder Datenbank vorhanden, wird jedoch meist durch ähnliche Funktionalitäten nachgebildet.

3.1.1 Atomacity

Die Atomicity, beziehungsweise die Atomarität einer Transaktion bedeutet, dass man eine Transaktion als unteilbare Einheit betrachtet, die entweder vollständig oder überhaupt nicht ausgeführt wird. Enthält diese mehrere Operationen, werden sie alle oder keine von ihnen erfolgreich abgeschlossen. Teilergebnisse sind nicht möglich. Die Atomarität verhindert, dass die Datenbank in einem inkonsistenten Zustand verbleibt, wenn eine Transaktion fehlschlägt. [1]

3.1.2 Consistency

Die Konsistenz gewährleistet, dass eine Transaktion die Datenbank von einem konsistenten Zustand in den nächsten konsistenten Zustand überführt. Dies bedeutet, dass die Integritätsregeln und Bedingungen, die in der Datenbank definiert sind, sowohl vor als auch nach der Transaktion erfüllt sein müssen. Nur während der Ausführung einer Transaktion darf gegen diese Anforderungen verstoßen werden. Wenn beim Abschluss dieser die Bedingungen nicht erfüllt sind, wird sie abgelehnt, und die Datenbank bleibt dadurch in einem konsistenten Zustand. Die Konsistenz sorgt dafür, dass die Datenbank keine widersprüchlichen oder inkonsistenten Daten

enthält. [1] Zur Vermeidung von Missverständnissen wird diese Definition der Konsistenz auch teilweise als Integrität der Daten bezeichnet.

3.1.3 Isolation

Durch die Isolation wird gewährleistet, dass Transaktionen unabhängig voneinander ablaufen. Sie werden als eigenständig angesehen und nicht von Aktionen beeinflusst, die zeitgleich auf dieselben Datenbankressourcen zugreifen. [1]

3.1.4 Durability

Die Dauerhaftigkeit garantiert, dass die Ergebnisse einer abgeschlossenen Transaktion nach einem Systemausfall oder Neustart der Datenbank erhalten bleiben. Sobald eine Transaktion erfolgreich beendet ist, werden ihre Änderungen in der Datenbank dauerhaft gespeichert und können nicht verloren gehen. Dadurch wird sichergestellt, dass die Datenbank auch bei unerwarteten Ereignissen oder Hardwarefehlern zuverlässig und persistent ist. Insbesondere bei dieser Anforderung sollten Konzepte beachtet werden, die außerhalb der jeweiligen Software liegen, zum Beispiel ein Write-Cache der Speichermedien. [1]

3.1.5 Normalformen

Die Normalformen stellen keine Anforderungen an die eigentliche Datenbankapplikation, sondern an die Modellierung der Daten. Ziele der Normalformen sind unter anderem die Strukturierung der Dateien, Vermeidung von Redundanzen und innere Integrität der Daten. Es gibt verschiedene Normalformen, von denen die ersten drei am häufigsten verwendet werden. Die erste Normalform (1NF) ist die grundlegendste und stellt sicher, dass alle Daten in einer Tabelle atomar sind. Das bedeutet, dass keine wiederholten Gruppen oder Arrays von Werten in einer einzelnen Zelle vorkommen. Jede enthält dabei nur einen einzelnen, unteilbaren Wert. Dies erleichtert das Abrufen, Aktualisieren und Verarbeiten von Daten erheblich. Die zweite Normalform (2NF) beseitigt Redundanz in Tabellen, die einen zusammengesetzten Primärschlüssel haben. Sie verlangt, dass alle Nicht-Schlüsselattribute von einem oder mehreren Teilen des Primärschlüssels abhängen. Folglich sollten Daten in separaten Tabellen organisiert werden, um Redundanz zu minimieren und die Datenintegrität sicherzustellen. Die dritte Normalform (3NF) beseitigt transitive Abhängigkeiten zwischen Nicht-Schlüsselattributen. Das bedeutet, dass diese nicht voneinander abhängen dürfen. Wenn eine solche Abhängigkeit besteht, sollten die betreffenden Attribute in separate Tabellen ausgelagert werden, um die Datenbankstruktur weiter zu optimieren und Redundanz zu minimieren. Höhere Normalformen, wie die Boyce-Codd-Normalform (BCNF), die vierte Normalform (4NF) und die fünfte Normalform (5NF), können in komplexeren Datenbanken ebenfalls relevant sein. Diese Normalformen bauen auf den Grundlagen der ersten drei auf und behandeln spezifische Arten von Abhängigkeiten und Anomalien in den Daten. [1] Im Endeffekt muss bei jeder Modellierung einer Datenbank eine Entscheidung zwischen Einhaltung der Normalformen, welche in der Regel einen höheren Aufwand bedeutet, und der Akzeptanz von fehlenden Eigenschaften, mit dem Nutzen von einfacherer Umsetzung, getroffen werden.

3.2 BASE

Das Akronym BASE wurde im Kontrast zum ACID-Prinzip erschaffen und beschreibt Eigenschaften von Datenbanken in verteilten Systemen. Dabei ist es wichtig, zu betonen, dass die unvollständige Erfüllung von ACID meist zu BASE führt [9]. Die Definitionen der einzelnen Aspekte unterscheiden sich je nach Quelle, beinhalten jedoch alle dieselbe Grundidee. Das Konzept steht im engen Zusammenhang mit dem später vorgestellten CAP-Theorem.

3.2.1 Basically Available

Die grundsätzliche Verfügbarkeit beschreibt die Eigenschaft, dass eine Verfügbarkeit des Systems nicht automatisch die aller Daten meint. [20]

3.2.2 Soft state

Die Eigenschaft des Soft-States beschreibt, dass sich die Daten unabhängig von der Applikation verändern können. Dies steht in engem Zusammenhang mit der eventuellen Konsistenz. Eine nachträgliche Herstellung der Konsistenz kann genau zu diesen Veränderungen führen. Im Gegensatz dazu steht der Hard-State, bei welchem sich die Daten nur durch direkte Manipulation von einer Applikation verändern. [22]

3.2.3 Eventually consistent

Die Beschreibung der eventuellen Konsistenz bezieht sich nicht auf die Konsistenz der Daten innerhalb der Datenbank, sondern auf die Konsistenz über mehrere Knoten des verteilten Systems. Konkret ist damit gemeint, dass alle Instanzen denselben Datenbestand haben. Die Eventualität bezieht sich auf die zeitliche Betrachtung. Es ist nicht jederzeit sichergestellt, dass alle Instanzen dieselben Daten liefern, jedoch möglich, da nach einer gewissen Zeit der konsistente Zustand in Hinblick auf ein einzelnes Datum wiederhergestellt wird. Eine gesamte Konsistenz kann ebenfalls erreicht werden, wenn keine weiteren Änderungen geschehen, bis alle Daten vollständig synchronisiert wurden. [20]

3.3 Anforderungen bei Microservices in moderner Infrastruktur

In den moderneren Entwurfsmustern der Microservices bleiben viele der bekannten Anforderungen bestehen, jedoch erhalten sie oftmals eine andere Gewichtung. Während in monolithischen Strukturen in der Regel eine Datenbank für alle Aufgaben genutzt wurde, steigt durch die wachsende Anzahl an Services auch die Möglichkeit und Bereitschaft eine unterschiedliche Gewichtung der Kriterien vorzunehmen und dadurch mehrere Systeme auszuwählen. Neben den bereits bekannten gibt es auch neue Anforderungen, die sich sowohl aus den Services und ihren Lebenszyklen ergeben, als auch aus den typischen Merkmalen einer modernen Infrastruktur.

Aus dem Entwurfsmuster der Microservices und dem Ansatz der "Separation of concerns" [11] ergibt sich eine weitere Eigenschaft, die in der Wahl der Datenbank zu berücksichtigen ist und diese in der Regel vereinfacht. Das Ziel ist es, dass eine Datenbank nur die Informationen für eine spezielle Ressource enthält. Zusammen mit der Auftrennung der Applikation in eigene Services ergibt sich die Aufteilung der Datenbank in mehrere kleine Teile. Ein zugrundeliegendes Entwurfsmuster dafür ist der Ansatz "Database per Service". Es gibt dabei verschiedene Ausprägungen. Die erste und einfachste Variante nennt sich "Table-per-Service" und beschreibt die Zuordnung einzelner Tabellen zu verschiedenen Microservices. Dieser Ansatz birgt das Risiko, dass unbewusst Abhängigkeiten zwischen Schemata und somit zwischen den Services geschaffen werden. Eine klarere Zuordnung der Tabellen zu einem Service ist möglich, indem das Schema der Tabelle dem jeweiligen Microservice zugeordnet wird. Dieser Ansatz nennt sich entsprechend "Schema-per-Service" und verdeutlicht die Grenzen eines Microservices innerhalb der Datenbank. Die weitgehendste Trennung wird mit dem Ansatz "Database-Server-per-Service" erreicht. Entsprechend dem Namen wird bei diesem Prinzip für jeden Service eine eigene Applikation erzeugt. Das bedeutet nicht, dass bei einem Betrieb von mehreren Instanzen eines einzelnen Services mehrere Datenbanken notwendig sind. Mit einem Service sind in diesem Falle nur die Arten von Services gemeint, nicht die konkreten Instanzen. [6] Im Gegensatz dazu gibt es das Designpattern der "Shared-Database". Dabei werden genau diese Aspekte nicht umgesetzt, sondern jede Applikation kann uneingeschränkt mit allen Daten interagieren. Ein möglicher Vorteil dieses Konzepts ist die Sicherstellung von Konsistenz über die Ebene der Datenbank, anstelle der Notwendigkeit dazu eine Kommunikation zwischen den Services zu nutzen. [7] Im Einzelfall muss abgewogen werden, welches Design Pattern für die jeweiligen Anwendungsfälle überwiegt. Dabei sollte insbesondere ein Augenmerk auf die logische Zusammengehörigkeit der Daten aus mehreren Services gelegt werden.

3.3.1 Containerisation

Microservices werden in den meisten Fällen in einer Infrastruktur bereitgestellt, die basierend auf Containern die jeweilige Instanz betreibt. Häufig handelt es sich dabei um Umgebungen, die Docker oder Kubernetes nutzen. Neben dem Betrieb der Applikation innerhalb eines Containers ist es ebenfalls möglich, die Datenbanken in eigenen Containern zu betreiben und somit die Vorteile einer Container-Landschaft zu nutzen. Ein Nachteil dabei ist, dass die Datenbank dem Lebenszyklus eines Containers unterworfen wird und diese damit umgehen können muss. Insbesondere in Kombination mit den anderen Anforderungen kann diese Eigenschaft zu Konflikten führen.

3.3.2 horizontale und individuelle Skalierbarkeit

Aufgrund der Fähigkeiten der Infrastruktur Vorgänge, wie beispielsweise die horizontale Skalierung, selbstständig durchzuführen, kann das Verhältnis zwischen Datenbankservern und Applikationsservern nicht genau vorhergesehen werden. Aus diesem Grund muss die Datenbank fähig sein, eine beliebige Anzahl an gleichzeitigen Clients zu bearbeiten. Wird zusätzlich dazu die Datenbank selbst in einem Container betrieben, ist es ebenfalls notwendig, dass diese horizontal skalieren kann. Insbesondere in Kombination mit den anderen Anforderungen kann das teilweise zu Konflikten führen.

3.3.3 CAP-Theorem

Das CAP-Theorem ist eine Anforderung, die aus dem Bereich der verteilten Systeme stammt. Es beschreibt, dass es bei verteilten Systemen nur möglich ist, zwei der drei Eigenschaften zu erfüllen. Das C steht für die Konsistenz (Consistency) der Daten und meint, ähnlich wie beim BASE-Ansatz, dass eine Transaktion eine Anpassung aller Replikate des Datensatzes hervorrufen muss, beziehungsweise bei einer Abfrage dieselben Daten geliefert werden müssen, unabhängig von der angefragten Instanz. Die Verfügbarkeit (Availability) bezieht sich auf die Reaktion der Datenbank mit akzeptablen Antwortzeiten auf die Anfragen der Clients. Die letzte Eigenschaft der Ausfalltoleranz (Partition Tolerance) erfordert den stabilen Weiterbetrieb der Software bei geplanten oder ungeplanten Ausfällen oder Anpassungen einzelner Serverstrukturen. [21] Zur Beschreibung der erfüllten Eigenschaften werden diese meistens in einem Akronym zusammengefasst. Eine CA-Datenbank beschreibt in diesem Fall die Erfüllung von Consistency und Availability.

3.3.4 Automatisierung

Eine Anforderung, die sich weniger auf die Datenbanken selber und vielmehr auf die Implementation innerhalb des Services bezieht, ist das streben nach einer möglichst automatischen Provisionierung und der Bereitstellung der Datenbank in einem für die eigentliche Funktionalität nutzbaren Zustand. Eine besondere Bedeutung erhält diese Eigenschaft bei der Implementierung eines automatischen Deployments [23] innerhalb der eigenen Infrastruktur.

4 Datenbanken

Der Begriff Datenbanken meint meist das Datenbanksystem (DBS), welches sich aus zwei Komponenten zusammenstellt. Der erste Teil ist die eigentliche Datenbank (DB). Diese hat die Aufgabe, alle Daten abzuspeichern, zu organisieren und bei einer Abfrage wiederzugeben. Die zweite Komponente ist das Datenbankmanagementsystem (DBMS). Alle Clients die mit dem Datenbanksystem kommunizieren wollen, verbinden sich mit dem Datenbankmanagementsystem und geben ihre Anforderungen anhand eigener Abfragesprachen, auch "query languages" genannt, an das System weiter. "Ein Datenbankmanagementsystem (DBMS) ist eine Software zur sicheren, konsistenten und persistenten Speicherung großer Datenmengen, mit dem Ziel mehreren Benutzern (gleichzeitig) effizienten, zuverlässigen, sicheren und bequemen Zugriff auf diese Daten zu ermöglichen." [1] Wie zu Beginn vorgestellt, sind Datenbanken eine zentrale Komponente der Architektur, wenn es um die Speicherung von Daten für Microservices geht. Diese Bedeutsamkeit entstand nicht erst bei den Konzepten von Microservices, sondern bereits bei früheren Softwareprodukten, die in monolithischen Strukturen entwickelt wurden. Lange Zeit waren relationale Datenbanksysteme das Mittel der Wahl, da sie es erlauben jegliche Daten und Relationen abzuspeichern und zu validieren. Mit immer wachsenden Anforderungen entstanden im Laufe der Zeit andere Datenbanksysteme für spezielle Verwendungszwecke. Diese werden meist den nicht-relationalen Datenbanksystemen zugeordnet. Im Folgenden werden die unterschiedlichen Konzepte vorgestellt. Es wird dabei bewusst auf Eigenschaften konkreter Softwareprodukte verzichtet, da in den ersten Analyseschritten die konzeptionellen Unterschiede eine Vorauswahl zulassen.

4.1 Relationale Datenbanksysteme

Die bekannteste Art der Datenbanksysteme sind die relationalen Datenbanksysteme oder oft als RDBMS für "Relation Database Management System" abgekürzt. Ein anderer Name dafür ist SQL-Datenbank, wobei SQL die Abkürzung für Structured-Query-Language ist und damit nur den Standard zur Kommunikation mit dem Datenbankmanagementsystem beschreibt. Dadurch ist das erste Kriterium für diesen Typ bekannt. Nahezu alle Implementationen nutzen eine, untereinander mehr oder weniger kompatible, Variante von SQL. Aufgrund der langen Existenz dient SQL häufig als Inspiration für neuere Query-Languages. Die Daten innerhalb der Datenbank werden in verschiedenen Schemata beziehungsweise Katalogen gruppiert. Ein Schema dient dabei der Zusammenfassung mehrerer Tabellen zu einer meist logischen Einheit. Die Tabelle definiert Eigenschaften, wie zum Beispiel die zugrundeliegende Architektur, auch als Engine bezeichnet, oder Zugriffsrechte für unterschiedliche Benutzer. Die eigentlichen Daten werden innerhalb der Tabelle als neue Zeile angelegt, sodass jede Entität eine eigene erhält. Standardisiert werden diese Zeilen durch beliebig viele Spalten, welche auf Ebene der Tabelle definiert werden. Eine Spalte beinhaltet konkrete Informationen zu den einzelnen Bestandteilen

der Daten. Darunter fallen Aspekte, wie der Datentyp, der Umgang mit fehlenden Werten und gegebenenfalls sinnvolle Standardwerte oder automatische Anpassungen. Diese Struktur muss vor dem Anlegen der Daten hinzugefügt oder, wenn sie schon vorhanden ist, entsprechend angepasst werden. Dazu dient ebenfalls die Sprache SQL, beziehungsweise der Teilbereich DDL, die Data Definition Langage. [19] Bei der Abfrage der Daten wird gezielt ausgewählt, welche zurückgegeben werden sollen und ob diese gegebenenfalls mit anderen Daten zusammengefasst oder weiterverwendet werden müssen. Durch diese Funktionen ist es möglich, Logik in die Datenbank auszulagern und anderseits Funktionalität selbst zu erschaffen, ohne eine weitere Komponente zu erzeugen. Die bekanntesten Datenbanken sind Oracle SQL, MySQL, MariaDB oder SQLite. Anhand der zugehörigen Anwendungen zeigt sich, dass SQL von kleinsten Daten bis hin zu großen Speichern von Unternehmen durchweg genutzt werden kann und wird.

4.2 Nicht-relationale Datenbanksysteme

Der Begriff "nicht-relationale Datenbanksysteme" oder im Englischen "NoSQL" als Akronym für "Not only SQL" meint nicht einen speziellen Datenbanktyp, sondern die Sammlung der Alternativen zu klassischen, relationalen SQL-Datenbanken. Insbesondere die englische Bezeichnung macht dabei deutlich, dass die Benutzung nicht dazu führt, dass alle Konzepte der relationalen Datenbanken verändert oder ignoriert, sondern das sie um andere Funktionalitäten erweitert werden. Diese neuen Funktionalitäten sind jedoch oftmals der Grund, dass sie nicht jede bekannte Funktion der SQL-Datenbanken erfüllen können. Im Folgenden werden die gängigsten Kategorien der NoSQL-Datenbanken mit dem Fokus auf ihre jeweiligen Abgrenzungskriterien vorgestellt.

4.2.1 Spaltenbasierte Datenspeicher (column-based storage)

Das Konzept der spaltenbasierten Datenspeicher ähnelt dem der relationalen Datenbanken. Hier werden die Daten ebenfalls in Tabellen strukturiert, wobei die Datensätze jeweils eine Zeile bilden und die Attribute in Spalten und Spaltengruppen organisiert werden. Gespeichert werden diese Dateien in den meistens Systemen anhand des Schlüssels und basieren nicht auf einem zuvor errechneten Hashwert. [26] Ein Vorteil dieser Datenbanken gegenüber der SQL-Variante liegt in der Reduzierung der notwendigen I/O-Operationen auf den Datenspeichern. Die Daten einer SQL-Tabelle werden in der Regel in Blöcken anhand der jeweiligen Zeile gespeichert. Dies kann zu ineffizienter Speichernutzung führen, wenn die Datengröße in den meisten Fällen von der Größe eines Speicherblocks abweicht. Im Kontrast dazu ist der Speicher für NoSQL Tabellen anhand der Spalten organisiert. Ein Vorteil dieses Ansatzes ist, dass innerhalb eines Datenblocks alle Einträge denselben Datentyp haben und somit spezielle Kompressionsverfahren für diesen angewandt werden können. Darüber hinaus kann der Speicherplatz eines Datenblocks effizienter genutzt werden, da insbesondere bei kleinen Datentypen seltener zusätzliche Datenblöcke notwendig werden, während bei zeilenorientierter Speicherung jedes Mal ein neuer Datenblock erzeugt werden muss. [4] Einige bekannte spaltenbasierte Datenspeicher sind HBase und Cassandra, welche jeweils von Apache bereitgestellt werden. In den Cloud Umgebungen werden oft die jeweiligen Softwarelösungen der Anbieter, wie zum Beispiel Amazon Redshift für AWS [2], genutzt.

4.2.2 Schlüssel-Wert-Datenspeicher (key-value storage)

NoSQL-Datenbanken, die einen Key-Value-Storage implementieren, bilden im Endeffekt eine große Hash-Tabelle. Sie erfüllen dieselbe Funktionalität, indem sie den Hashwert des Schlüssels berechnen, und anschließend wird dieser Wert zur Speicherung der zugehörigen Werte genutzt. [26] Aus diesem Grund ist der Typ äußerst performant in Hinblick auf einzelne Lesevorgänge anhand der Schlüssel, auf der anderen Seite ist sie weniger optimiert auf Aspekte, wie der Filterung von Nicht-Schlüsseln oder der Verkettung von Daten. Außerdem sollte beachtet werden, dass ein Update der Werte zu einem auswechseln des gesamten Datensatzes führt, wodurch die Operationen bei großen Datenmengen pro Datensatz zeitintensiv werden können. Auf der anderen Seite ermöglicht das Speicherformat eine Skalierung über mehrere Knoten, da es keine Abhängigkeiten zwischen einzelnen Datensätzen gibt.

4.2.3 Zeitreihen-Datenspeicher (time-series database)

Zeitreihen Speichersystem werden häufig zur Speicherung von chronologischen Daten genutzt. Ein großes Anwendungsfeld sind Messwerte von Sensoren und IoT-Komponenten. [26] Der Schlüssel wird hierbei durch das Datum gebildet und die Werte sind die zugehörigen Informationen. Aufgrund der vordefinierten Anwendung ist hier eine gezielte Optimierung in Hinblick auf die typischen Auswertungen einer Zeitreihe oder Problemen, wie dem Empfang der Daten in falscher Reihenfolge, möglich. Durch ihre Ähnlichkeiten werden für diese Arbeit die Zeitreihen-Datenspeicher den Cloumn-Based-Datenspeichern zugeordnet.

4.2.4 Dokumentdatenspeicher (document-based storage)

Anders als die bisher vorgestellten Datenbanken sind die Daten in einer Dokumentdatenbank nicht in Tabellen organisiert, sondern innerhalb von Dokumenten mit flexiblen Schemata. Viele Datenmodelle sind nicht von Beginn an vollständig definiert, sodass diese Flexibilität die Erweiterung des Datenmodells vereinfacht. Die Dokumente sind meistens in schemabasierten Formaten, wie JSON oder XML, und damit textbasiert gespeichert. [15] ein weiterer Vorteil dieser Formate ist, dass sie am gängigsten für den Datentransfer in Web-Applikationen und insbesondere Web-APIs sind. Somit sind sie den meisten Entwicklern vertrauter, als Strukturen, die auf Tabellen basieren. Darüber hinaus erlaubt dieser Zusammenhang, dass die Datenmodelle leicht wiederverwendet werden können. Die meisten Systeme ermöglichen außerdem ein Parsing dieser Daten und somit Queries innerhalb der Dokumente.

4.2.5 Graphendatenbank (graph database)

Eine Graphendatenbank, oder teilweise “Diagramm Datenspeicher“ genannt, bezeichnet eine fundamental andere Datenstruktur als die bisher vorgestellten Systeme. Während bei den anderen Systemen die eigentlichen Daten im Vordergrund stehen, fokussiert sich dieses auf die Beziehung der Daten untereinander. Wie der Name vermuten lässt, sind die Datensätze innerhalb der Datenbank als Graph im Sinne der Informatik definiert. Jeweils ein Datum bildet dabei einen Knoten und steht meistens mit einem oder mehreren anderen Knoten in einer Relation. Diese wird als Kante dargestellt. Die Hauptaufgabe dieser Architektur stellt die

effiziente Abfrage von Daten anhand von einzelnen oder langkettigen Relationen dar. Aufgrund der unterschiedlichen Aufgaben ist insbesondere hier eine andere Abfragesprache vorhanden als SQL oder eine daran angelehnte Syntax.

4.2.6 Objektdatenspeicher (object data storage)

Objektdatenspeicher sind, anders als die bisher vorgestellten Datenbanken, nicht für die Speicherung von Datensätzen aus elementaren Datentypen oder einer Komposition dieser gedacht, sondern richten sich gezielt an die Anforderung sonstige binäre Datenmengen zu speichern. [26] In der Regel sind diese Dateien deutlich größer als elementare Datenfelder, wodurch manche Systeme einen Datensatz auf mehrere Knoten aufteilen, um die durchschnittliche Lesegeschwindigkeit zu erhöhen.

4.2.7 Hybride Applikationen

In vielen Fällen ergibt die Analyse der Anforderungen einer Anwendung, dass das Profil zu mehreren Datenbanktypen passt oder explizit Funktionalitäten aus verschiedenen Typen benötigt werden. Ein mögliches Vorgehen in diesem Fall, ist die Anbindung an mehrere verschiedene Applikationen mit den jeweiligen Funktionalitäten. Einige Softwareprodukte im Kontext der NoSQL-Datenbanken bieten die Möglichkeit, innerhalb eines Servers verschiedene NoSQL-Typen bereitzustellen. In solchen Fällen sollte explizit darauf geachtet werden, dass diese Implementation verlässlich ist und den eigenen Erwartungen entspricht.

4.3 Speichersysteme für Datenbanken

Im Allgemeinen gibt es zu nahezu jeder Kategorie von Datenbankmanagementsystemen jeweils zwei Versionen der Speicherverwaltung. Auf der einen Seite stehen die deutlich seltener genutzten In-Memory-Systeme und auf der anderen Seite die bekannten On-Disk-Storage-Systeme. Beide Systeme haben ihre individuelle Anwendungszwecke, sodass eine Koexistenz beider Varianten eine spezifisch angepasste Technologieentscheidung ermöglicht. Ergänzend zu der Möglichkeit, verschiedene Anwendungen mit den jeweiligen Speichersystemen auszurüsten, oder in begründeten Fällen beide Systeme in einer Anwendung zu implementieren, gibt es bei vielen Systemen mittlerweile die Option die Art des Speichers zu wählen. So ist es zum einen möglich, die Entscheidung über das konkrete Speichersystem unabhängig vom Datenbanksystem zu machen und zum anderen ermöglicht dies teilweise eine kombinierte Lösung beider Arten der Speicherung in einer Instanz des Datenbanksystems. Trotz dieser Möglichkeit sollte das Zusammenspiel von Software, Datenbanksystem und Datenspeicher ausreichend beachtet werden. Alleine durch die Funktionsweise und die Anwendungsbereiche, gibt es bei vielen Datenbanksystemen eine bevorzugte Art der Speicherung. Nachfolgend wird eine Einführung in die jeweiligen Speicherkonzepte sowie in die möglichen Vor- und Nachteile der verschiedenen Ansätze gegeben.

4.3.1 On-Disk-Storage Systeme

Eine der Hauptaufgaben von Datenbanken ist die Persistierung von Daten, weshalb viele die persistenten Speichermedien des jeweiligen Gerätes nutzen. Systeme, die diesen Speicherort wählen, werden aufgrund des englischen Namens Disk als On-Disk-Storage-Systeme bezeichnet. Bei Anwendungen, die mit einem Filesystem arbeiten, wird eine oder mehrere Dateien angelegt, in denen die einzelnen Einträge gespeichert werden. [18] Jede Operation führt daher, unter Vernachlässigung von Caching-Lösungen, zu einer oder oftmals zu einer Vielzahl von Input/Output-Operationen auf dem Datenträger. Aus diesem Verhalten resultiert eines der großen Performancekriterien für On-Disk-Speicher: Die Performanz des zugrundeliegenden Speichermediums oder Medienpools. Insbesondere bei großen Lesevorgängen, welche die Kapazitäten des Caches überschreiten oder bei Schreibvorgänge, die eine direkte Speicherung anfordern, ist die Lese- und Schreibgeschwindigkeit unmittelbar von der Geschwindigkeit der Datenquelle abhängig. Aufgrund der typischen Funktionalität eines Datenspeichers, Dateien an einem beliebigen freien Platz abzuspeichern, kann es vorkommen, dass ähnliche Daten, die häufig zusammen benötigt werden, trotzdem an weit entfernen Speicherorten abgelegt sind. Dieses Verhalten führt dazu, dass neben der Datenrate des Datenspeichers auch die Latenz für zufällige Anfragen eine große Rolle spielen kann. Ein weiterer Punkt ist die Eignung des Datenspeichers für eine große Anzahl von Lese- und Schreibvorgängen innerhalb des Lebenszyklus, um eine vorschnelle Alterung und somit potentielle Ausfälle zu vermeiden. Damit die benannten Einflüsse insbesondere bei zeitintensiven Schreibvorgängen abgemildert werden können, bieten viele Systeme die Möglichkeit einen Write-Cache zu nutzen. Dieser bearbeitet zusätzlich zu den integrierten Caches der Speichermedien die Aufgaben im Cache und überträgt sie erst im Anschuss daran auf den eigentlichen Speicher. Manche Systeme bieten darüber hinaus die Möglichkeit, hauptsächlich im Cache zu arbeiten und nur gelegentlich, beispielsweise nach einigen Stunden, eine Änderung auf dem eigentlichen Datenspeicher auszuführen. Bei typischen Konfigurationen des Systems, die die Daten zeitnah auf dem Medium speichern, bietet diese Art des Speicherns eine Sicherheit gegenüber dem Verlust von Daten. Bei unvorhergesehenen Ereignissen, wie einem Stromausfall, ist somit sichergestellt, dass die Daten persistiert und beim nächsten Startvorgang weiterhin vorhanden sind.

4.3.2 In-Memory-Storage Systeme

Die Alternative zu den bekannten On-Disk-Datenspeichern sind In-Memory-Systeme. Dem Namen entsprechend nutzen diese nicht die Datenträger des Systems, sondern den Speicher, der zur Laufzeit bereitgestellt wird. Das Verlassen auf den Speicher des Systems bringt sowohl Vorteile als auch Risiken mit sich. Der größte Vorteil bei der Verwendung des Arbeitsspeichers für alle Daten ist die erheblich schnellere Zugriffszeit, wodurch eine der beiden großen Performancekriterien eines In-Memory-Speichers eliminiert wird. [16] Der zweite Aspekt in der Performance lässt sich durch die Verwendung des RAMs lösen. Die Datenrate eines modernen RAM-Riegels liegt in der Regel in der Größenordnung von etwa 20 GB/s. [8] Dem gegenüber stehen NVMe-Speicher mit maximal 3,5 bis 7 GB/s oder SATA-SSDs mit sogar nur 0.6 GB/s. [12] Die Vorteile, die diese Systeme mit sich bringen, müssen mit dem Nachteil abgewogen werden, der aus der Beschaffenheit des RAM entsteht. Der RAM ist ein flüchtiger Speicher, wodurch von einem potentiellen Datenverlust ausgegangen werden muss. Bei jedem Ausfall der Spannungsversorgung des Speichers gehen alle darauf enthaltenen Informationen verloren.

Die fehlende Garantie über die Beständigkeit zeigt in einem anderen, später ausführlicher betrachteten Aspekt, jedoch ebenfalls einen weiteren Vorteil. Es ist dadurch möglich, mehrere Instanzen parallel zu betreiben, ohne das diese eine gemeinsame Datengrundlage benötigen. Um das Problem der fehlenden Persistierung zu beheben, benutzen einige Systeme das Konzept von Replica-Sets. Bei diesen Systemen gibt es einen oder mehrere Primärknoten, die die Anfragen bedienen. Im Hintergrund werden die Daten an verschiedenen weiteren Knoten, die selbst eine In-Memory- oder On-Disk-Speicherlösung verwenden, weitergeleitet. Im Falle eines Ausfalls können die Dateien direkt aus den anderen In-Memory-Systemen bereitgestellt werden oder mittels des On-Disk-Systems eine neue In-Memory-Instanz mit den alten Zuständen wiederherstellen. [16] Ein typischer Anwendungsfall für diese Technologie ist die Bereitstellung eines Caches für semi-permanente Daten. Insbesondere im Kontext von Micorservices können dadurch unnötige Requests vermieden werden, da zuvor erzeugte Daten erneut verwendet werden. Insbesondere der Verlust der Daten ist hier kein großes Problem, da diese jederzeit neu generiert werden können. Sollen die Daten trotzdem gesichert werden, ist meist ein Replica-Set mit On-Disk-Storage die bevorzugte Wahl. [3] Ein anderer Anwendungsfall ist der Umgang mit Live-Daten. Unabhängig vom konkreten Fall geht es dabei um eine möglichst schnelle Interaktion mit den Vorgängen. In den meisten Systemen ist hier dennoch eine Persistierung erwünscht, sodass im Hintergrund Mechanismen dafür benötigt werden. [3]

5 Auswahl- und Entscheidungskriterien

Im Folgenden soll zu vereinzelten Aspekten der Anforderungsanalyse eine genauere theoretische Betrachtung in Hinblick auf die jeweiligen Stärken und Schwächen der Datenbanksysteme durchgeführt werden. Alle Untersuchungen wurden dabei auf konzeptioneller Ebene ausgeführt, um in Phasen des Software-Designs den Typ der Datenbank festlegen zu können, ohne sich mit konkreten Applikationen beschäftigen zu müssen. Innerhalb dieser Betrachtung werden die Objektdatenbanken nicht evaluiert, da diese sich an eine andere Art von Daten richten und somit nicht direkt vergleichbar sind. Ebenfalls wird das Konzept hybrider Datenbanksysteme in dieser Ausarbeitung nicht betrachtet, da sie im Allgemeinen eine Verbindung der Eigenschaften der jeweiligen Varianten darstellen und die Bewertung somit selbstständig gebildet werden kann. Die konkrete Auswahl der Implementierung muss jeweils individuell durchgeführt werden, da sie in dem Spannungsfeld zwischen optimaler Funktionalität und vorhanden Kompetenzen verordnet werden muss. Ein Grund für das Auslassen dieses Themenbereichs ist der hohe Einfluss des Profils und der Präferenzen der Entwicklerteams bei dieser Fragestellung. Das Ergebnis dieser Betrachtung ist keinesfalls als konkreter und allumfassender Leitfaden zu sehen, er ist vielmehr eine Inspiration für mögliche Vergleiche und sollte an die individuellen Bedürfnisse der eigenen Projekte angepasst werden.

5.1 Kriterien

Die ausgewählten Kriterien bilden eine Mischung aus reinen Aspekten der Datenbankmodelle, Eigenschaften für die Software und Anforderungen, die in der Infrastruktur begründet sind. Die Bewertung findet unter Annahme einiger Grundsätze innerhalb des Systems statt. Die gesamten Anwendungen werden in einer hybriden Umgebung betrieben, die aus einem lokalen Rechenzentrum und einer Cloud-Struktur besteht. In dieser wird die Laufzeitumgebung durch verschiedene Kubernetes Cluster bereitgestellt, welche unterschiedliche Kunden bedienen sollen. Die Microservices wurden in Anlehnung an das Konzept der getrennten Zuständigkeiten (Single Responsibility Principle) erstellt und bündeln demnach mehrere Funktionalitäten, die sich alle auf dieselben Datensätze beziehen.

5.1.1 ACID

Die Anforderungen des ACID-Prinzips waren maßgeblich für die Entwicklung von relationalen Datenbanken, weshalb sie die Kriterien dieser meistens problemlos erfüllen. Die Erfüllung der ACID-Kriterien ist in verschiedenen NoSQL-Varianten unterschiedlich umgesetzt. Ein Großteil der frühen Systeme kann ACID nicht erfüllen, da meist die Konsistenz mit der Bedeutung aus dem ACID-Prinzip aufgrund der verteilten Systeme nicht umgesetzt werden konnte. Mittlerweile gibt es Systeme, die ACID-Kompatibel sind. Am Beispiel der dokumentenbasierten

Datenbanken zeigt sich die Weiterentwicklung deutlich. Zuerst wurde die Konsistenz nur innerhalb einzelner Dokumente erfüllt, während sie in neueren Implementierungen auch mehrere Dokumente umfassen kann. [17] Weiteren Einfluss auf die Dauerhaftigkeit der Daten hat das verwendete Speichersystem, bei welchen beide Systeme die Werte zur Laufzeit der Software speichern. Bei einem Neustart der Instanz gehen die Werte einer In-Memory-Datenbank jedoch verloren, während sie bei On-Disk-Speichern erhalten bleiben.

5.1.2 Normalformen

Die Ziele der Normalformen, bei welchen insbesondere die Deduplizierung der Daten zu erwähnen ist, lassen sich in den meisten Fällen nur bedingt auf die Datenbanken für Microservices übertragen. Vor allem bei dem Konzept "Database per Service" ist eine Duplizierung von Werten nicht völlig vermeidbar. Die Definitionen der Normalform lässt sich nur auf ein einzelnes Schema beziehen, sodass diese nicht direkt weiterverwendbar sind.

5.1.3 BASE

Im Vergleich zu ACID als Standard für relationale Datenbanken bildet das BASE-Prinzip die Grundlage in vielen Implementierungen verschiedener NoSQL-Datenbanken. Bei der Betrachtung der Erfüllung dieser Kriterien ist zu beachten, dass es sich eher um Einschränkungen handelt als um direkte Anforderungen. So zeigt sich zum Beispiel bei den relationalen Datenbanken, dass sie ohne die entsprechenden Einschränkungen auskommen. Basically-Available ist nicht gegeben, da die Datenbank immer mit allen Daten verfügbar ist. Die Eventual-Consistency wird ebenfalls durch die garantierte Konsistenz der Daten auf jeglichen Replikaten umgangen. Aus diesem Grund ist auch ein möglicher Soft-State nicht gegeben, da der Zustand sich nur durch die Transaktionen ändern kann. Anders sieht es bei den NoSQL-Datenbanken aus. Nahezu alle Implementierungen sind auf verteilte Systeme ausgelegt und führen somit zu Basically-Avaliable. Die anderen beiden Kriterien hängen stark vom Umgang der Implementierung mit Transaktionen ab. Sollte die Implementierung nicht garantieren, dass Änderungen sofort auf alle Replikate ausgeführt werden, resultiert daraus ein möglicher Soft-State. Durch die nachträgliche Synchronisation wird auch die Konsistenz möglicherweise erst später hergestellt, sodass nur Eventual-Consistency gegeben ist.

5.1.4 CAP-Theorem

Das CAP-Theorem wird in den meisten Fällen in Hinblick auf NoSQL-Datenbanken betrachtet. Bei einer Betrachtung der relationalen Datenbanken zeigt sich, dass diese je nach Konfiguration entweder CP oder CA erfüllen. Dadurch wird der Fokus auf die Konsistenz der Daten gelegt. Spalten-basierte Datenbanken priorisieren oftmals die Skalierbarkeit und damit die Ausfallsicherheit gegenüber der Konsistenz, jedoch gibt es in dem meisten Fällen zusätzlich die Möglichkeit die Konsistenz als zentralen Faktor zu definieren. Eine ähnliche Priorisierung nehmen Key-Value-Datenbank vor, sodass diese ebenfalls eher auf AP-Szenarien spezialisiert sind. Die Dokumenten-basierten Datenbanken richten sich in dem meisten Fällen nach Verfügbarkeit und Ausfallsicherheit. Bei Graphdatenbanken wird häufig ein Master-Slave-System eingesetzt. Dieses System nutzen ebenfalls die relationale Datenbanken, sodass die Effekte in Hinblick

auf das CAP-Theorem ähnlich sind und die Ausfallsicherheit weniger priorisiert wird. Diese Erfüllung von AP und damit der Verlust von Concistency führt zu der zuvor erwähnten Einschränkungen in Hinblick auf ACID.

5.1.5 Single Responsibility & Discrete Boundaries

Das Konzept der Single-Responsibility und der diskreten Grenzen lässt sich mit jeder der in dieser Arbeit betrachteten Datenbanksysteme erfüllen. Die Schwierigkeit liegt dabei in der Modellierung der Daten, da von Beginn an beachtet werden muss, dass diese nicht vermischt werden dürfen. Der einfachste Ansatz für diese Vorhaben ist in jedem System das Konzept des Datenbankservers pro Service, bei welchem durch technische Grenzen logische Grenzen aufgelöst werden. Ist der Ansatz einer zentralen Datenbank erwünscht, müssen Konzepte angepasst werden. Die Funktion eines Schemas oder einer Tabelle ist in manchen Systemen, wie unter anderem einer Objektdatenbank, die nur aus Buckets und Daten besteht, nicht im klassischen Sinne vorhanden. In diesem Fall muss eine Alternative mit den konkreten Möglichkeiten des Systems gefunden werden. Eine Besonderheit in diesem Themenfeld ist die Graphendatenbank. Während diese ebenfalls ermöglicht, für den Service eigenständig betrieben zu werden, ist es dort nicht zwingend sinnvoll. Eine der Hauptaufgaben und Ziele dieser Strukturen ist die Vernetzung von Daten. Insbesondere im Sinne der Auswertung kann es sinnvoll sein den Graphen über die Grenzen eines Service hinweg zu definieren. Bei der Vernetzung ist zudem eine Differenzierung zur Trennung von Zuständigkeiten notwendig. Diese bezieht sich meist auf die Arten von Knoten und Kanten, anstelle auf die konkrete Struktur.

5.1.6 Containerfähigkeit

Aufgrund der angenommenen Infrastruktur eines Kubernetes-Cluster, sollte die Datenbank innerhalb dieses funktionsfähig sein. Um das zu erreichen, ist der Betrieb der Datenbank in einem Container notwendig. Einen großen Einfluss auf die Umsetzbarkeit dieses Systems hat das verwendete Speichersystem. Im Falle eines On-Disk-Speichers muss für die entsprechenden Instanzen möglicherweise ein geteilter, aber in jedem Fall ein persistenter Speicherplatz bereitgestellt werden, welcher aufgrund der vielen Lese- und Schreibvorgänge möglichst performant sein muss. Ein weiterer Aspekt ist, dass bei diesen Datenbanken ein geordnetes Beenden wichtig ist, um valide Datensätze zu garantieren. Im Falle eines In-Memory-Speicherns wird dies vereinfacht. In aller Regel besitzen diese Systeme keine Möglichkeiten die Daten über die Laufzeit einer Instanz hinaus zu speichern. Dadurch kann auf diese Speicherplätze verzichtet und von einem leeren oder zumindest vordefinierten Datenbestand ausgegangen werden. Dieser ist ebenfalls beim Start eines Containers vorhanden. Aus diesem Grund lässt sich festhalten, dass In-Memory-Systeme in Hinblick auf Container einen leichten Vorteil gegenüber den On-Disk-Systemen besitzen. Die konkreten Datenbanksysteme haben einen geringeren Einfluss auf diese Eigenschaften, da mittlerweile nahezu alle Hersteller Container für die Datenbanken bereitstellen und diese entsprechend an den Lebenszyklus dieser angepasst sind.

5.1.7 Skalierbarkeit

Wie zuvor beschrieben besitzen, viele Datenbanksysteme mittlerweile vorbereitete Container, jedoch sind diese nicht unbedingt für die Skalierung geeignet. Die Eigenschaft ist eng an das CAP-Theorem gekoppelt. Je spezifischer die Systeme auf die Verfügbarkeit und Ausfallsicherheit ausgerichtet sind, desto besser lassen sie sich skalieren. Diese Eigenschaften werden bei den meisten Systemen durch die Erweiterung der Instanzen, also der Definition von horizontaler Skalierbarkeit, erreicht. Auf der anderen Seite führt dies häufig zu einem Verlust der Konsistenz der Daten über mehrere Instanzen hinweg.

6 Fazit

Zum Abschluss dieser Seminararbeit “Datenbanken für Microservices”‘ lassen sich einige Schlüsselerkenntnisse festhalten. Die Evolution von Softwarearchitekturen zu Microservices hat die Anforderungen an Datenbanken teilweise verändert. In modernen Infrastrukturen müssen Datenbanken den Prinzipien von Microservices entsprechen, indem sie eigenständig, skalierbar, containerfähig und automatisierbar sind. Die Wahl der richtigen Datenbanklösung für Microservices ist dabei entscheidend. Sie sollte sorgfältig anhand vielfältiger Anforderungen und Kriterien des konkreten Projektes getroffen werden. Je nach den spezifischen Anforderungen können relationale oder nicht-relationale Datenbanksysteme sowie verschiedene Speichersysteme in Betracht gezogen werden. Es ist wichtig, zu betonen, dass es keine Einheitslösung gibt, die für alle Microservice-Projekte geeignet ist. Eine weitere Erkenntnis der Untersuchung ist, dass nahezu alle Datenbankarten jede standardmäßige Aufgabe erfüllen können, da die meisten das ACID-Prinzip umsetzen. Einzig das Speichersystem hat einen größeren Einfluss auf die Erfüllung dieses Prinzips.

Zusammenfassend zeigt diese Arbeit, dass die richtige Datenbankauswahl einen erheblichen Einfluss auf die Leistung, Skalierbarkeit und Zuverlässigkeit von Microservices hat. In den meisten Fällen zeigt sich die Qualität der Entscheidung nicht in den ersten Entwicklungsphasen, sondern erst im späteren Betrieb oder bei Weiterentwicklungen. Unternehmen und Entwickler sollten sich deshalb bewusst sein, dass die Datenbankstrategie ein integraler Bestandteil der Gesamtarchitektur ist und eine gründliche Planung erfordert.

6.1 Einsatz hybrider und heterogener Systeme

Eine Möglichkeit, die Fähigkeiten mehrerer NoSQL-Varianten zu nutzen, ist die Verwendung der vorgestellten hybriden Systeme. Ein Nachteil dieser Misch-Lösungen ist, dass diese in der Regel in Hinblick auf eine einzelne Technologie weniger optimiert sind. Die Vereinigung mehrerer Strukturen in einer Software ermöglicht es, bei gemischten Anforderungen einer Anwendung für jede dieser eine optimale Datenbank auszuwählen, ohne dabei verschiedene Systeme betreuen zu müssen. Die parallele Verwendung eigenständiger Systeme ist ein alternativer Ansatz, um die Auswahl der Variante anhand von Teilen der Anforderungen zu ermöglichen. Aus operativer Sicht sollte dabei unbedingt bedacht werden, dass dies sowohl den Wartungsaufwand als auch den Ressourcenverbrauch erhöht. Unter Berücksichtigung der daraus entstehenden Kosten, sollte bei der Projektplanung eine Abwägung der Vorteile durch bessere Optimierung durchgeführt werden.

7 Weiterführende Aufgaben und Forschungsfragen

Neben den in dieser Arbeit analysierten Aspekten gibt es viele weitere Design Patterns, die in Hinblick auf die Auswahl des Datenbanksystems und den späteren Betrieb der Software berücksichtigt werden können. Eine Vielzahl dieser ergibt sich aus den Entwurfsmustern die bei der Entwicklung des Microservices eingesetzt werden, da sie Anforderungen an den Datenspeicher implizieren können. Ebenfalls untersucht werden könnten konkrete Unterschiede der verschiedenen Deployment-Strategien für das Zusammenspiel von Microservices und Datenbanken. Ein anderer Aspekt, der in Bezug auf die NoSQL-Datenbanken auftauchen kann, sind notwendige Veränderungen in den Datenmodellen. Eine veränderte Zuordnung der Daten anhand ihrer Microservices führt gegebenenfalls zu anderen Strukturen in der Speicherung und Modellierung. Insbesondere bietet sich in diesem Kontext die Graphendatenbank mit der Fokussierung auf Beziehungen an. Im Sinne der Anwendung der in dieser Ausarbeitung gesammelten Erkenntnisse, bietet sich an eine Untersuchung von verschiedenen Implementierungen aus den jeweiligen Kategorien der Datenbanken durchzuführen. Ein dabei hervorzuhebendes Themenfeld könnten die hybriden Datenbanken sein, da diese gegebenenfalls in anderen Aspekten als isolierte Systeme geprüft werden müssen. Für die Analyse bietet sich eine Kombination aus theoretischer Arbeit, Modellierungen, operativen Erkenntnissen und einem Vergleich der Belastbarkeit der Anwendung an.

7.1 Entwicklung unternehmensspezifischer Leitfäden

Für die Nutzung der gesammelten Erkenntnisse bietet sich für Unternehmen eine Erstellung von Leitfäden und Entscheidungshilfen an. Ein erster Leitfaden sollte dabei zur Auswahl einer Variante von Datenbanken dienen. Der Fokus dieses Dokuments sollte, ähnlich zu der vorliegenden Ausarbeitung, auf den Kategorien sowie allgemeinen Daten zu den Systemen liegen. Darüber hinaus sollten ebenfalls spezifische Anforderungen der Firma berücksichtigt werden. Mögliche Aspekte sind dabei die Erfüllung von Auflagen zur Zertifizierung einer Software oder eine besondere Laufzeitumgebung. Nutzbar wäre eine solche Bearbeitung insbesondere im Rahmen der Planungen und Architekturentscheidungen. In Kombination mit Erkenntnissen des vorherigen Abschnitts und Erfahrungswerten des Unternehmens kann eine erweiterte Übersicht mit Fokus auf konkrete Softwareprojekte erstellt werden. Dieser Leitfaden ist insbesondere für die Entwicklerteams eine Unterstützung, da diese nicht mehr recherchieren müssen, um diese Anforderungen umsetzen zu können, sondern hätten direkt eine vorgefertigte Auswahl. Für eine effiziente Nutzung dieser Dokumente ist es notwendig, die Inhalte als lebendig und im ständigen Wandel anzusehen, um die neusten Entwicklungen abzubilden.

A Literaturverzeichnis

[1] Alexaner Voß. Datenbanken, 2022.

[2] Inc. Amazon Web Services. AWS | Amazon Data Warehouse Redshift – Datenverwaltung. URL: https://aws.amazon.com/de/redshift/.

[3] Amazon web Services, Inc. What Is an In-Memory Database? URL: https://aws.amazon.com/nosql/in-memory/.

[4] Amazon web Services, Inc. Columnar storage - Amazon Redshift, August 2021. URL: https://docs.aws.amazon.com/redshift/latest/dg/c_columnar_storage_disk_mem_mgmnt.html.

[5] Bob Reselman. 5 design principles for microservices, January 2022. Section: Kubernetes. URL: https://developers.redhat.com/articles/2022/01/11/5-design-principles-microservices.

[6] Chris Richardson. Microservices Pattern: Database per service. URL: http://microservices.io/patterns/data/database-per-service.html.

[7] Chris Richardson. Microservices Pattern: Shared database. URL: http://microservices.io/patterns/data/shared-database.html.

[8] Crucial. RAM Memory Speeds & Compatibility. URL: https://www.crucial.com/support/memory-speeds-compatability.

[9] Armando Fox, Steven D Gribble, Yatin Chawathe, Eric A Brewer, and Paul Gauthier. Cluster-Based Scalable Network Services. URL: https://people.eecs.berkeley.edu/~brewer/cs262b/TACC.pdf.

[10] HandWiki. ACID (Computer Science). URL: https://encyclopedia.pub/entry/35885.

[11] Joseph Ingeno. Separation of Concerns (SoC) - Software Architect's Handbook. ISBN: 9781788624060. URL: https://www.oreilly.com/library/view/software-architects-handbook/9781788624060/8ff905c2-217a-47f0-85c2-789296d42e8d.xhtml.

[12] Kingston. NVMe vs SATA: What is the difference? - Kingston Technology. URL: https://www.kingston.com/en/blog/pc-performance/nvme-vs-sata.

[13] Mark Richards. 4. Microservices Architecture Pattern - Software Architecture Patterns. URL: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch04.html.

[14] Matthew A. Titmus. 1. What Is a “Cloud Native” Application? - Cloud Native Go. ISBN: 9781492076339. URL: https://www.oreilly.com/library/view/cloud-native-go/9781492076322/ch01.html.

[15] MongoDB. Document Database - NoSQL. URL: https://www.mongodb.com/document-databases.

[16] MongoDB. In-Memory Databases Explained. URL: https://www.mongodb.com/databases/in-memory-database.

[17] MongoDB. What Does ACID Compliance Mean? | An Introduction. URL: https://www.mongodb.com/databases/acid-compliance.

[18] MySQL. MySQL 8.0 Reference Manual :: 15.6.3.2 File-Per-Table Tablespaces. URL: https://dev.mysql.com/doc/refman/8.0/en/innodb-file-per-table-tablespaces.html.

[19] Oracle. What is a relational database? URL: https://www.oracle.com/database/what-is-a-relational-database/.

[20] Prof. Dr. Faeskorn-Woyke. Datenbanken Online Lexikon | Datenbanken / BASE. URL: https://wikis.gm.fh-koeln.de/Datenbanken/BASE.

[21] Prof. Dr. Faeskorn-Woyke. Datenbanken Online Lexikon | Datenbanken / CAP-Theorem. URL: https://wikis.gm.fh-koeln.de/Datenbanken/CAP.

[22] Prof. Dr. Faeskorn-Woyke. Datenbanken Online Lexikon | Datenbanken / Soft State. URL: https://wikis.gm.fh-koeln.de/Datenbanken/SoftState.

[23] Red Hat. What is deployment automation? URL: https://www.redhat.com/en/topics/automation/what-is-deployment-automation.

[24] Refactoring Guru. History of patterns. URL: https://refactoring.guru/design-patterns/history.

[25] Susan J. Fowler. 4. Scalability and Performance - Production-Ready Microservices. ISBN: 9781491965924. URL: https://www.oreilly.com/library/view/production-ready-microservices/9781491965962/ch04.html.

[26] Zoiner Tejada. Nicht relationale Daten und NoSQL - Azure Architecture Center. URL: https://learn.microsoft.com/en-us/azure/architecture/data-guide/big-data/non-relational-data.