ulm university universität
uulm

Fakultät für
Ingenieurwissenschaften
und Informatik
Institut für Verteilte Systeme
Institut für Organisation und
Management von Informations-
systemen

Januar 2014

Design und Implementierung eines zuverlässigen und verfügbaren (NoSQL) Datenbanksystems

Masterarbeit an der Universität Ulm

OMI-2014-M-02

Vorgelegt von:
Christopher B. Hauser

Gutachter:
Prof. Dr. Stefan Wesner
Prof. Dr. Frank Kargl

Betreuer:
Dr. Jörg Domaschka
Dipl.-Inf. Benjamin Erb

Fassung 27. Januar 2014

Satz: PDF-LaTeX 2ε

Abstract

Die Verwendung von Datenbanksystemen für die Datenhaltung in Anwendungen ist populär, da der Implementierungsaufwand für die Manipulation und Speicherung zu persistierender Daten hinfällig wird. Die Wahl des Datenbanksystems ist im Hinblick auf Anforderungen an die Anwendung und zukünftige Veränderungen wie Wachstum für einen erfolgreichen Betrieb entscheidend. Datenbanksysteme setzen unterschiedliche Schwerpunkte im Bereich der Fehlertoleranz, der Skalierbarkeit oder der Konsistenz, werden jedoch in vorhandener Literatur überwiegend anhand des Speichertyps wie relational oder Dokumenten-orientiert kategorisiert. Eine Einteilung in Klassen von Datenbanksystemen gemäß verwendeter Konzepte der Fehlertoleranz wie Replizierung, Skalierung, Konsistenz, Konfliktmanagement und Dauerhaftigkeit bietet eine alternative Sichtweise auf Datenbanksysteme, die eine Auswahl des Datenbanksystems für eine Anwendung auf ihre Anforderungen hin erleichtert. Die Ausarbeitung definiert hierfür aus einer neuen Perspektive fünf Gruppen von Datenbanksystemen.
Die Ausarbeitung beschreibt neben der Definition der Konzepte und einer Analyse elf verbreiteter Datenbanksysteme ein Design eines fehlertoleranten Datenbanksystems. Das Design wird auf Basis des Virtual Nodes Frameworks, welches für verteilte fehlertolerante Anwendungungen verwendet werden kann, implementiert. Die Implementierung setzt dabei auf Virtual Nodes auf und modifiziert Komponenten des Frameworks, um Partitionierung des globalen Anwendungszustands und dadurch partielle Replikation zu ermöglichen.

Inhaltsverzeichnis

1. Einleitung

Die Anforderungen an Anwendungen unserer modernen technischen Infrastruktur sind sehr hoch. Durch die inzwischen fast allgegenwärtige Verknüpfung unserer Gesellschaft mit weltweit erreichbaren Anwendungen durch das Internet sind in den vergangenen Jahren neue Maßstäbe und Herausforderungen für die technische Umsetzung entstanden. Die weit verbreitete Nutzung stationärer aber auch mobiler Zugangsknoten für Nutzer resultieren in hohen Erwartungen an Erreichbarkeit und Zuverlässigkeit von Anwendungen. Betreiber von weltweit genutzten Anwendungen betreiben kommerzielle Dienste und sind finanziell oder sogar existenziell direkt mit dem Betrieb einer Anwendung verbunden. Doch auch in kleineren Dimensionen sind Software und gespeicherte Daten omnipräsent. Viele softwaregestützte Betriebsabläufe in Firmen müssen verfügbar und zuverlässig sein um die strategischen Interessen des Unternehmens verfolgen und realisieren zu können. Die Speicherung von privat, organisatorisch oder wirtschaftlich genutzten Daten muss verlustfrei und der Zugriff darauf jederzeit möglich sein.

Die skizzierten Szenarien stellen zwei Herausforderungen dar. Zum einen werden Anwendungen wegen wachsender Nutzerzahlen *immer größer*. Die wachsende Variable erfordert zum einen mitwachsende Speichervolumen aber auch ein Zuwachs an Prozessen um die Anwendung auszuführen (vgl. Abbildung 1.1). Zum anderen werden die Anwendungen *immer wichtiger*, da von kleinen bis hin zu sehr großen Unternehmen die Organisation, die Dienstleistungen oder sogar der Verkauf digital erbracht wird. Wächst ein Unternehmen, so muss seine technische Infrastruktur angemessen mit wachsen, sodass die beiden Bereiche Größe und Wichtigkeit miteinander verknüpft sind. Um die Komplexität der Implementierung neuer Anwendungen zu verringern, wird die Datenhaltung in Datenbanksysteme ausgelagert. Abbildung 1.1 stellt schematisch die Nutzer dar, die eine skalierbare Anwendung konsumieren, deren Datenhaltung auf einem ebenso skalierbaren Datenbanksystem beruht.

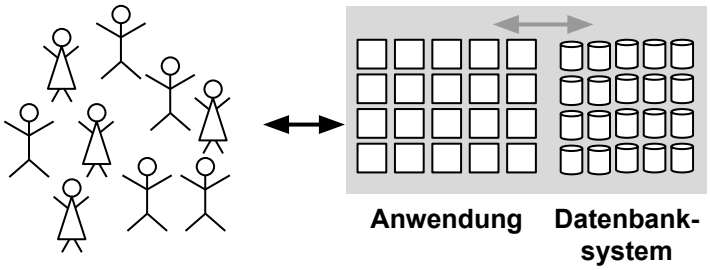

Abbildung 1.1.: Nutzer verwenden skalierbare Anwendungen die Datenbanksysteme verwenden

Damit immer größer werdende Anwendungen ihre Aufgaben erledigen können, sollen diese auch bei wachsenden Zahlen an Zugriffen *verfügbar* sein. Um z.B. keine Datenverluste zu erleiden soll eine Anwendung *zuverlässig* arbeiten. Die beiden Begriffe werden im Hinblick auf Datenbanksysteme in dieser Ausarbeitung genauer definiert.

1.1. Motivation

Die Masterarbeit besteht aus den beiden Teilen *Design* und *Implementierung*, die aufeinander aufbauen. Der erste Teil soll eine Übersicht vorhandener Konzepte der Fehlertoleranz in Datenbanksystemen bieten, während der zweite Teil ein mögliches Design eines Datenbanksystems definiert und die Implementierung mit dem Virtual Nodes Framework vorstellt.

Merkmale zur *Klassifizierung von Datenbanksystemen* in bestehender Literatur orientieren sich hauptsächlich am verwendeten Speichertyp. Datenbanksysteme werden demnach unterschieden, ob sie dem relationalen Modell oder einem Speichertyp der NoSQL Datenbanksysteme wie Spalten-orientiert angehören. Die vorliegende Ausarbeitung definiert Konzepte und daraus resultierend eine alternative Klassifizierung zur Sicherstellung von Zuverlässigkeit und Verfügbarkeit von Datenbanksystemen. Das Ergebnis kann daher zur Auswahl eines geeigneten Datenbanksystems für eine zu entwickelnde Anwendung verwendet werden. Das Verständnis von Fehlertoleranz in Datenbanksystemen ist zudem für den zweiten Teil der Ausarbeitung, dem Entwurf und der Implementierung eines eigenen Datenbanksystems notwendig.

Virtual Nodes ist primär ein Framework für die Entwicklung von fehlertoleranten verteilten Anwendungen. Ein Entwurf und eine Implementierung eines fehlertoleranten Datenbanksystems soll zeigen, wie geeignet das Framework für die Implementierung eines Datenbanksystems ist. Ein Faktor dafür ist die Menge und der Aufwand notwendiger Änderungen am Framework.

1.2. Struktur

Datenbanksysteme können als verteiltes System betrachtet werden, deren Schwerpunkt auf der Speicherung von Daten liegt. Zunächst werden in dieser Arbeit daher in Kapitel 2 grundlegende Konzepte aus dem Bereich der verteilten Systeme erläutert. Der Begriff der Fehlertoleranz wird erläutert, der die Kriterien verfügbar und zuverlässig beinhaltet. Zudem können Datenbanksysteme anhand ihrer Speichertypen kategorisiert werden, sodass relationale und NoSQL Datenbanksysteme unterteilt werden können.

Nach der Einführung in Grundlagen werden Konzepte vorgestellt, die aus existierenden Datenbanksystemen extrahiert wurden. Kapitel 3 stellt diese Konzepte zur Realisierung von Fehlertoleranz in Datenbanksystemen kategorisch vor. Die einzelnen Themen wie Skalierung oder Replizierung werden zunächst erläutert und jeweils tabellarisch aufgelistet. Anhand der Kategorisierung des Kapitels lassen sich Datenbanksysteme analysieren und gegenüberstellen.

Eine solche Analyse dient als Grundlage der Klassifizierung von Datenbanksystemen, welche sich in Kapitel 4 auf Grundlage der ermittelten Kategorien und elf betrachteten Datenbanksystemen anschließt. Die betrachteten Datenbanksysteme sind dabei aus den Bereichen der relationalen und der NoSQL Datenbanksysteme.

Der zweite Teil der Ausarbeitung beschreibt die begleitende Implementierung zur vorliegenden Arbeit. Ziel ist ein fehlertolerantes Datenbanksystem auf Grundlage des *Virtual Nodes Framework*s. Das verwendete Framework dient der Realisierung fehlertoleranter verteilter Anwendungen und

muss im Hinblick auf die Implementierung eines Datenbanksystems analysiert und angepasst werden.

Zunächst befindet sich in Kapitel 5 die Beschreibung eines Anwendungsfalls, der vom implementierten Datenbanksystem abgedeckt werden soll. Das implementierte Konzept orientiert sich an einem Konzept aus dem Design-Teil in Kapitel 4, welches im Hinblick auf den Anwendungsfall angepasst wurde. Kapitel 5 beschreibt neben des Anwendungsfalls die Umsetzung der Anforderungen und die Kernaufgaben.

Die Umsetzung des Datenbanksystems kann in Komponenten gegliedert werden, die unterschiedlich tief in das verwendete Virtual Nodes Framework eingreifen. Die notwendige Implementierung besteht aus einem Service, der klassisch auf Virtual Nodes aufsetzt und das Framework nutzt. Zwei weiteren Komponenten ersetzen die Middleware- und die Replikations-Schicht des Frameworks. Die Implementierung wird in Kapitel 6 vorgestellt.

Abschließend werden die betrachteten Themen der Ausarbeitung in Kapitel 7 diskutiert. Zum einen wird auf Datenbanksysteme und deren Konzepte und Klassifizierung im Bezug auf Fehlertoleranz eingegangen. Zum anderen wird die Implementierung diskutiert. Eine Zusammenfassung in Kapitel 8 schließt die Ausarbeitung mit einem Ausblick, in dem mögliche Erweiterungen enthalten sind, ab.

2. Grundlagen

Für die Umsetzung und das Verständnis zuverlässiger und verfügbarer Datenbanksysteme werden im Folgenden zunächst Grundlagen verteilter Systeme erläutert. Anschließend werden Grundlagen existierende Datenbanksysteme vorgestellt. Für eine spätere Implementierung auf Basis von Virtual Nodes wird zudem das Framework vorgestellt.

Die betrachteten verteilten Systeme verfolgen dabei eine sog. *Shared-Nothing Architektur*. Darunter ist zu verstehen, dass keine gemeinsamen Komponenten wie Festplatten oder RAID-Systeme verwendet werden sondern einzelne Datenknoten vollständig eigene Hard- und Software-Systeme nutzen. Die notwendige Kommunikation findet über Netzwerk-Nachrichten oder den Austausch von Dateien statt.

2.1. Verteilte Systeme

Verteilte Systeme beschreiben eine Menge von Rechnern, die gemeinsam eine Anwendung ausführen und dadurch untereinander organisiert sind. Zunächst können beteiligte Rechner in Clients und Server unterteilt werden, wobei Clients eine Anwendung konsumieren, die von Servern bereitgestellt wird. Die Kommunikation zwischen Clients und Servern findet nach einem Request-Response-Verfahren statt, wobei ein Client eine Anfrage (Request) an einen Server sendet und eine Antwort (Response) erhält. Eine Ansammlung von Servern wird als Cluster bezeichnet, wobei innerhalb eines Clusters ein Server auch als Knoten bezeichnet wird (Abb. 2.1). Die Kommunikation zwischen einem Client und einer serverseitigen Anwendung findet z.B. über entfernte Methodenaufrufe oder über Nachrichtenübertragungen statt. Besteht die serverseitige Anwendung aus einem einzigen Knoten, wird dieser Knoten die Anfrage bearbeiten. Sind mehrere Knoten vorhanden, sind weitere Mechanismen zur Synchronisierung innerhalb des Clusters notwendig (vgl. Kapitel 2.1.1 und 2.1.2).

2.1.1. Synchronisierung

Um eine Synchronisierung mehrerer Rechner innerhalb eines verteilten Systems zu erreichen ist eine einheitliche Reihenfolge von Ausführungsschritten notwendig. Beispielsweise sollen Anfragen von Clients nicht in einer willkürlichen, zufälligen Reihenfolge auf jedem Knoten eines Clusters anders sondern in der selben Reihenfolge wie die Reihnefolge des Eintreffens der Anfragen erfolgen. Um ein gemeinsames Verständnis für Reihenfolgen zu bekommen, ist der Begriff von Uhren notwendig. Als Uhr kann eine absolute Zeit durch einen lokalen Timer mit gelegentlichem Abgleich untereinander verwendet werden. Die Synchronisierung läuft

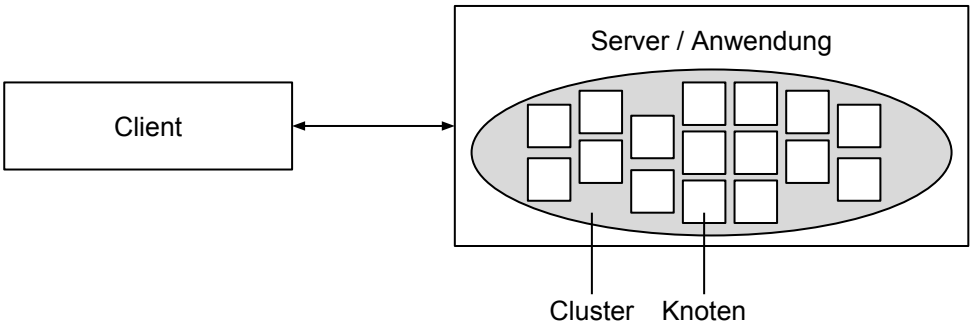

Abbildung 2.1.: Übersicht einer Anwendung auf Basis eines verteilten Systems

unabhängig von eintreffenden Anfragen und kann durch die Verwendung von logischen Uhren vereinfacht werden. Bei logischen Uhren wird für eintreffende Anfragen eine relative Zeit definiert. Dadurch kann eine relative Reihenfolge definiert werden, was in vielen Anwendungsfällen gegenüber einer absoluten Zeitangabe ausreicht. Beispielsweise bei *Lamport Uhren* wird eine Reihenfolge einzelner Ereignisse definiert, die jedoch nicht zwingend voneinander abhängen. Um diese strikte Reihenfolge zu lockern, werden *Vektor Uhren* verwendet, die zwischen Ereigbnissen, die kausal voneinander abhängig sind, Reihenfolgen definieren [27].

Neben der Synchronisierung der Reihenfolge von Anfragen oder Ereignissen ist die Synchronisierung des Anwendungszustands notwendig. Ein erfolgreich synchronisierter Anwendungszustand innerhalb eines verteilten Systems wird als *konsistent*, der Zeitraum einer Abweichung als *Inkonsistenz-Fenster* bezeichnet. Der Anwendungszustand kann zu einem beliebigen Zeitpunkt global gesammelt und als Momentaufnahme (Snapshot) festgehalten und synchronisiert werden. In diesem Fall steht die Synchronisierung in keiner Abhängigkeit zu Anfragen von Clients. Um den Anwendungszustand partiell zu synchronisieren kann über ein Abstimmungs-Verfahren während der Verarbeitung einer Anfrage der veränderte Teil-Zustand synchronisiert werden. Da ein Anwendungszustand eine beliebig komplexe Struktur darstellen kann, erfolgt eine *Serialisierung* des Zustands, in der eine sequentielle Reihenfolge des Zustands erzeugt wird. In dieser Form kann eine Speicherung oder eine Netzwerkübertragung erfolgen. Auf Empfängerseite erfolgt eine Wiederherstellung, die sog. Deserialisierung.

Eine Herausforderung in verteilten Systemen, die einen gemeinsamen Anwendungszustand verwenden, ist die die Zugriffskontrolle, um Inkonsistenzen zu vermeiden. Vorhandene Zugriffsverfahren sind pessimistisch oder optimistisch. Beim pessimistischen Ansatz wird z.B. durch wechselseitigen Ausschluss eine Kollission vorzeitig verhindert. Mit Hilfe von verteilten Transaktionen werden Anfragen gruppiert und als eine atomare Operation ausgeführt. Zu Gunsten paralleler Ausführungen werden Transaktionen serialisiert, sodass Transaktionen ohne gegenseitige Abhängigkeiten nebenläufig ausgeführt werden. Um gegenseitgen Zugriff auf eine gemeinsame Ressource zu verhindern, wird mit dem 2-Phasen-Commit-Protokoll eine Sperre für Zugriffe eingerichtet. Beim optimistischen Ansatz hingegen werden keine Sperren oder Zugriffskontrollen ausgeführt, sodass Konflikte entstehen dürfen, die jedoch erkannt und dann aufgelöst werden. Die Konfliktlösung kann bspw. zu einem Abbruch einer Anfrage oder einer Transaktion führen [27].

2.1.2. Kommunikationsmodelle

Damit einzelne Knoten eines verteilten Systems intern gemeinsam zusammen arbeiten um eine verteilte Anwendung korrekt ausführen zu können, werden Kommunikationsformen benötigt. Grundsätzlich kann dafür ein deterministisches Verhalten der Knoten verwendet werden, sodass sich einzelne Knoten implizit auf die Arbeitsweise der anderen Knoten verlassen können und keine Kommunikation zur Laufzeit erfolgt. Teilweise ist jedoch eine interne Kommunikation notwendig, damit Knoten sich explizit absprechen und austauschen können. Diese Kommunikation kann Nachrichten-basiert zwischen allen Knoten oder über zentrale Kommunikations-Knoten erfolgen. Neben der internen Kommunikation ist eine Kommunikation zwischen dem Client und der Anwendung notwendig. Hierfür werden entfernte Prozedur- bzw. Methodenaufrufe verwendet (bspw. RPC bzw. RMI) oder Nachrichten- oder Stream-basiert kommuniziert [27]. Die Kommunikation kann auf verschiedene Schichten oder Protokolle im OSI-Modell erfolgen.

Cluster-interne Kommunikation

Neben Unicast-, Broadcast- oder Multicast-Nachrichten, die an einen einzelnen oder an eine Gruppe von Knoten übertragen wird, kann unterschieden werden, welcher Knoten die Kommunikation initiiert. Gerade im Bezug auf Redundanz kann ein replizierter Knoten die Kommunikation z.B. regelmäßig initiieren, um sich möglichst konsistent mit anderen Knoten zu halten. Dieser Ansatz ist pull-basiert. Im Gegensatz dazu steht das push-basierte Verfahren, bei dem ein Knoten die Kommunikation an alle anderen Knoten über mehrere Unicasts oder einen Broadcast initiiert. Die beiden Verfahren werden als *asynchrone* bzw. *synchrone* Kommunikation bezeichnet [30]. Bei der Übertragung von Nachrichten an mehrere Knoten z.B. über IP-basierte Netzwerke kann aufgrund von unterschiedlichen Laufzeiten durch Entfernungen oder verschiedenen Auslastungen von Teilstücken des Netzwerks oder von Geräten die Reihenfolge und der Zeitpunkt des Eintreffens nicht vom Netzwerk gewährleistet werden. Eine exakt identische Reihenfolge der Verarbeitung einzelner Nachrichten auf verschiedenen Empfänger-Knoten kann mit Hilfe von logischen Uhren erfolgen. Ein solcher Nachrichtenaustausch nennt sich *Total-Ordered Broadcast*.

Als weiteres Kriterium der cluster-internen Kommunikation ist die Definition, was übertragen werden soll. Zum einen besteht die Möglichkeit, dass neue Daten als Ergebnis einer Client-Anfrage vollständig übertragen werden. Zum anderen, dass eine reine Benachrichtigung ohne den eigentlichen Inhalt übertragen wird und der empfangende Knoten den aktuellen Zustand selbst einholt. Als dritte Möglichkeit kann eine Aktualisierungsoperation versendet werden, die von jedem Knoten ausgeführt werden muss um den lokalen Anwendungszustand zu aktualisieren. Hierfür kann eine cluster-interne Operation erzeugt oder die Anfrage des Clients verwendet werden.

Client-Server Kommunikation

Nachdem ein Client mit einem Server verbunden ist, kann eine Anfrage übertragen werden. Diese Anfrage kann, wie bei e-Mails, Nachrichten-basiert erfolgen. Andererseits kann eine entfernte Methode aufgerufen werden. Dazu werden häufig auf Ebene der Anwendungsentwicklung Middleware-Systeme verwendet, die einen Methodenaufruf der Programmiersprache über ein

Netzwerk abbildet und neben des gerufenen Methodennamen und ggfs. zuständigen Objekts auch Parameter und Rückgabewerte überträgt. Ein Beispiel für eine solche Middleware ist Java RMI auf Basis der Programmiersprache Java.

Die Kommunikation zwischen Server und Client kann synchron oder asynchron erfolgen. Bei synchroner Kommunikation blockiert der Client nach Absenden seiner Anfrage bis zum Erhalt einer Antwort des Servers. Java RMI verwendet bspw. eine synchrone Kommunikation. Alternative dazu ist eine asynchrone Kommunikation, bei der der Client vom Server kontaktiert wird, sobald eine Antwort auf seine zuvor abgesetzte Anfrage vorliegt. Diese nicht blockierende Variante erfordert die Möglichkeit einer Kommunikationsaufnahme des Servers zum Client, welche z.B. bei Verwendung einer Network-Address-Translation (NAT) oder einer Firewall zu Problemen führen kann.

2.2. Fehlertoleranz

In verteilten Systemen ist bei zunehmender Menge an Hardware die Chance eines Hardwareausfalls größer. Die zunehmende Verwendung softwaregestützter Systeme erfordert zudem eine verlässliche Arbeitsweise der Anwendung. Die beiden in der Einleitung dieser Arbeit eingeführten Begriffe *verfügbar* und *zuverlässig* lassen sich im Begriff Fehlertoleranz zusammenfassen. Dabei ist zu verstehen, dass das Verhalten eines Systems, welches sich im Sinne der Anwendung stets korrekt verhält als fehlertolerant gilt. Der Begriff der Fehlertoleranz setzt sich aus den Teilen "Fehler" und "Toleranz" zusammen und beschreibt das Verhalten eines Systems, welches im Fall eines auftretenden Soft- oder Hardwarefehlers die Funktionsweise des Gesamtsystems erhält und dadurch tolerant in Bezug auf den Fehler erscheint. Ein fehlertolerantes, oder auch verlässliches System soll daher die Anforderungen Verfügbarkeit, Zuverlässigkeit, Sicherheit und Wartbarkeit abdecken [27], welche im Folgenden definiert werden.

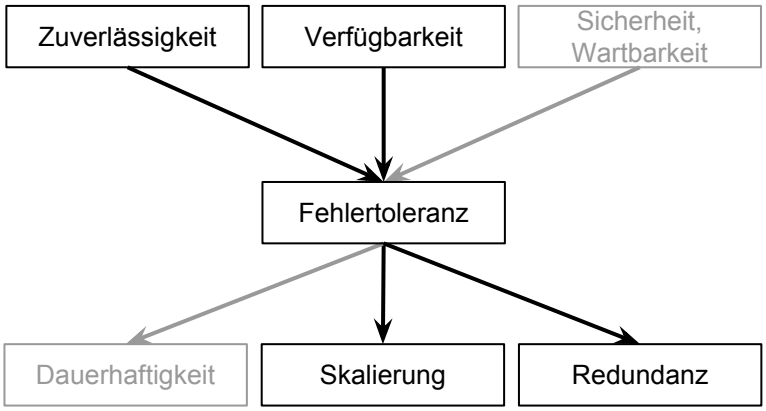

Abbildung 2.2.: Themenfeld "Fehlertoleranz" in Datenbanksystemen

Die *Verfügbarkeit* eines Systems beschreibt die Erreichbarkeit zu einem beliebigen Zeitpunkt, sodass das System in vollem Funktionsumfang jederzeit nutzbar ist. Wird ein unterschiedlich stark frequentiertes System betrachtet, muss die Verfügbarkeit auch in Lastspitzen gewährleistet sein, um als fehlertolerant zu gelten. Im Bezug auf Datenbanksysteme versteht sich die Verfügbarkeit

auch unter hoher Last, z.B. durch große gespeicherte Datenvolumen oder überdurchschnittlich viele Anfragen an das System. Anfragen sollen in einem verfügbaren Datenbanksystem auch mit entsprechend regulären Antwortzeiten beantwortet werden.

Ein *zuverlässiges* System beschreibt eine fortlaufend fehlerfreie Ausführung innerhalb eines betrachteten Zeitraums. Im Fall eines Datenbanksystems bedeutet zuverlässig, dass eine Anfrage richtig interpretiert und korrekt ausgeführt wird und nach positiver Rückmeldung auch positiv ausgeführt wurde. Bspw. soll das System nach positiver Rückmeldung auf einen Schreibzugriff die Änderungen tatsächlich geschrieben haben, sodass spätere Lesezugriffe den veränderten Datensatz erhalten. Die Begriffe Verfügbarkeit und Zuverlässigkeit stehen in engem Verhältnis, da nur durch beide Kriterien gemeinsam ein fehlertolerantes Datenbanksystem ermöglicht werden kann.

Die beiden Kriterien *Sicherheit und Wartbarkeit* stehen nicht unmittelbar im Fokus dieser Arbeit, da für ein fehlertolerantes Datenbanksystem im Forschungskontext andere Maßstäbe als für ein kritisches Produktivsystem gelten. Unter Sicherheit ist im Kontext von Fehlertoleranz definiert, dass nichts Katastrophales passieren darf, wenn ein System vorübergehend nicht korrekt funktioniert. Diese Eigenschaft ist bspw. in Flugzeugsystemen von hoher Bedeutung, hingegen für eine Datenbankanwendung für bspw. Social Media nicht primäres Ziel. Der Begriff der Wartbarkeit beschreibt, dass ein ausgefallenes System einfach zu reparieren ist, um schnellstmöglich wieder in einen produktiven Zustand zu gelangen. Dieses Kriterium ist bspw. für kommerzielle Systeme interessant, bei denen Ausfälle finanzielle Folgen haben.

Damit ein verteiltes System fehlertolerant funktioniert, müssen Vorkehrungen getroffen werden. Die Verfügbarkeit kann durch Skalierung erhöht werden, die Zuverlässigkeit durch Redundanz (vgl. Abb. 2.2). Die folgenden Kapitel gehen auf Skalierung und Redundanz genauer ein.

2.2.1. Skalierung

Skalierung ist notwendig, wenn durch Zuwachs an Daten oder Zugriffen eine Überlastung des Systems droht. Skalierung erhöht für diesen Fall die Verfügbarkeit. Skalierung enstpricht also der Erweiterung eines Systems um zusätzliche Ressourcen, um mehr Leistung zu gewinnen. Zunächst existieren drei Formen der Skalierung, die unterschiedlich komplex in Ihrer Realisierung sind [27].

Der einfachste aber beschränkteste Fall der Skalierung ist eine *vertikale Skalierung*, bei der die Hardware weiter ausgebaut wird. Im Fall eines Datenbanksystems kann eine vertikale Skalierung sinnvoll sein, wenn bspw. das Datenvolumen der Festplatte eines einzelnen Servers aufgebraucht ist und diese durch eine größere Festplatte getauscht werden kann. Diese Art der Skalierung, besonders von wenigen einzelnen Knoten, ist durch die Verfügbarkeit größerer Hardware beschränkt.

Sofern die Anwendung eines verteilten Systems eine funktionale Aufteilung in mehrere Teilanwendungen ermöglicht, kann eine *funktionale Skalierung* sinnvoll sein. Diese Form der Skalierung findet dabei weniger auf Ebene des verteilten Systems sondern mehr auf Anwendungsebene statt. Die Anwendung kann hierbei verschiedene verteilte Systeme parallel nutzen und die Verknüpfung selbst herstellen. Im Bezug auf Datenbanksysteme ist eine funktionale Skalierung

der Anwendung auf mehrere unterschiedliche Datenbanksysteme möglich. Eine Anwendung kann verschiedene Datenbanksysteme oder mehrere Instanzen eines Datenbanksystems parallel betreiben.

Sind die Möglichkeiten der vertikalen und funktionalen Skalierung beschränkt oder ausgereizt, kann eine *horizontale Skalierung* des Systems erfolgen. Dabei werden zur Skalierung des Systems beliebig zusätzliche Knoten hinzugefügt und die Aufgaben darauf verteilt. Die Ansammlung der Knoten kann als Cluster bezeichnet werden. Zwischen den Knoten findet bspw. Nachrichten-basiert Kommunikation statt, damit das Gesamtsystem als Ganzes arbeiten kann. Im Fall von Datenbanksystemen kann eine Partitionierung der Daten stattfinden, damit eine gleichmäßige Aufteilung der Daten auf zur Verfügung stehende Knoten erfolgen kann. Die horizontale Skalierung kann auch zur räumlichen Verteilung der Daten genutzt werden, um zum einen Daten räumlich näher an den Nutzer zu bringen oder zum anderen bspw. vor regionalen Naturkatastrophen geschützt zu sein.

Verteilungsstrategien

Bei horizontaler Skalierung stehen mehrere Knoten für die Anwendung zur Verfügung, die parallel genutzt werden. Entscheidend ist dabei die optimale Verteilung der Anwendung innerhalb der verfügbaren Ressourcen. Zudem sind Mechanismen erforderlich, wie ein Client einen zuständigen Server im Cluster findet und sich verbindet.

Damit Verteilung realisiert werden kann, werden eindeutige *Identifier* für Teile der Anwendung benötigt. Im Fall eines Datenbanksystems kann dieser Identifier der Primärschlüssel eines Datensatzes sein. Für die Verteilung werden zwei Zuteilungen benötigt. Zum einen muss ein Identifier einem Knoten zugeordnet werden. Zum anderen muss ein Client mit einer Anfrage an ein verteiltes System anhand des angefragten Identifiers einem Knoten zugeordnet werden. Diese Zuordnungen können statisch oder dynamisch realisiert werden. Im statischen Fall kann eine Menge von Identifiern in einzelne Bereiche aufgeteilt werden, wobei jeder Bereich einem Knoten zugewiesen wird. Diese statische Verteilung (*Ranges*) wird auf allen oder in einzelnen zentralen Organisations-Knoten gespeichert. Dynamische Verteilung errechnet während der Zuordnung anhand des Identifiers den Knoten. Diese Berechnung kann mit Hilfe einer Hashfunktion erfolgen. Ein einfaches Beispiel einer Hashfunktion ist eine Modulo-Operation mit dem Identifier und der Anzahl an verfügbaren Knoten.

Eine Hashfunktion hat den Vorteil, dass die Organisation der Verteilung nicht gespeichert wird sondern beliebig errechnet werden kann. Die Berechnung hat jedoch den Nachteil, dass eine Veränderung der Gegebenheiten wie das Hinzufügen oder Entfernen einzelner Knoten eine Veränderung der Hashfunktion zur Folge hat. Die Verteilung verliert daher an Flexibilität. Der Begriff *Consistent Hashing* beschreibt ein spezielles Verfahren mit Hilfe von Hash-Funktionen, welches dieses Problem löst [13, 18]. Consistent Hashing eignet sich besonders für die Speicherung von großen Datenvolumen in einem Cluster mit vielen Knoten.

Consistent Hashing definiert für die Partitionierung einen Wertebereich, dessen Enden aneinander schließen und dadurch einen Ring bilden (vgl. Abb. 2.3). Auf diesen Wertebereich wird aus zwei verschiedenen Richtungen über eine oder zwei verschiedene Hashfunktionen zugegriffen. Zum einen werden Client-Zugriffe auf das Cluster anhand ihres Identifiers auf eine Position im

Ring gehasht, zum anderen werden Knoten z.B. manuell oder über Hashing einer eindeutigen Kennung auf den Wertebereich verteilt. Abbildung 2.3 zeigt ein Beispiel mit einem Wertebereich des Rings von [1, 24] und drei Servern (S1, S2, S3), die gleichmäßig im Ring verteilt sind. Ein Client-Zugriff bekommt nun bspw. mit einer Modulo-Hashfunktion die Position 3 im Ring zugewiesen. Im Uhrzeigersinn der nachfolgende Knoten, im Beispiel S1, ist für diesen Zugriff zuständig.

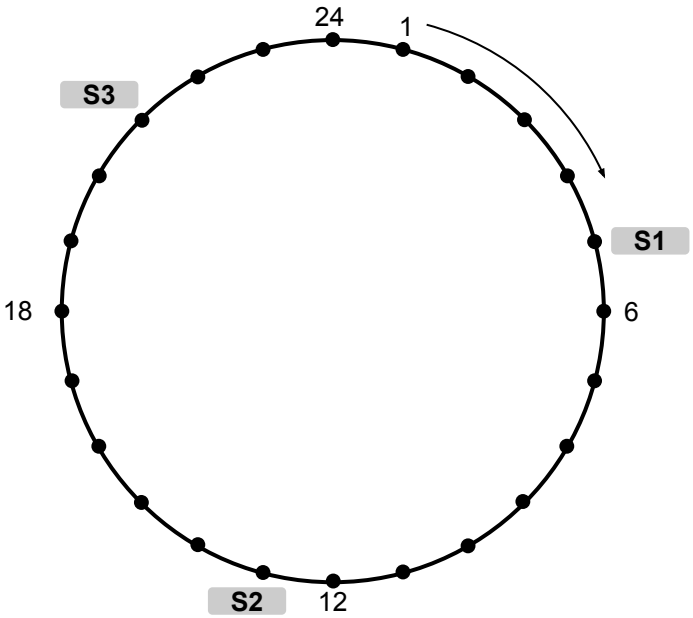

Abbildung 2.3.: Consistent-Hashing-Ring

Die Flexibilität dieses Verfahrens entsteht durch die Entkopplung der Position von Knoten zum Hashverfahren eines Identifiers auf einen Bereich. Dadurch können z.B. Knoten mit starker Hardware einen größeren Bereich abdecken als Knoten mit schwächerer Hardware. In diesem Fall muss die Verteilung der Knoten im Ring jedoch gespeichert werden und kann nicht weiter per Hashfunktion erfolgen. Consistent Hashing kann daher in diesem Fall als Mischform aus den beiden vorgestellten Verteilungsstrategien Ranges und Hashing betrachtet werden.

2.2.2. Redundanz

Redundanz wird eingesetzt, um Systeme zuverlässiger zu machen, aber auch um die Verfügbarkeit zu erhöhen. Bei redundant arbeitenden verteilten Systemen findet die Ausführung oder Speicherung mehrfach parallel statt. Die Ergebnisse können bei Bedarf verglichen werden, um Zuverlässigkeit zu gewinnen. Im Fall eines Knotenausfalls kann eine Anfrage von einem redundanten Knoten ausgeführt werden, was das System verfügbarer macht. Außerdem kann Redundanz zur Skalierung eingesetzt werden, wenn Anfragen an das System parallel abgearbeitet werden müssen um die Menge an Anfragen zu bewältigen. Gerade bei mehrfacher Speicherung eines Anwendungszustands ergeben sich jedoch zusätzliche Herausforderungen an die Synchronisierung. Ebenso müssen z.B. parallele Schreibzugriffe auf den selben Datensatz auf verschiedenen Knoten mit Konflikten umgehen können. Bei parallelen Ausführungen

müssen die Knoten den selben deterministischen Ausführungsregeln folgen, um zum selben Ergebnis zu kommen. Die folgenden Kapitel erläutern Grundlagen zu Konfliktmanagement und Konsistenzmodellen.

In redundanten, verteilten Systemen kann unterschieden werden, ob alle beteiligten Knoten exakt den gleichen Aufgaben nach gehen oder ob eine Unterscheidung nach Knoten-Typen vorhanden ist. Sind alle Knoten gleich und kommunizieren gegenseitig, wird der Aufbau als Peer-to-Peer bezeichnet. Beispielsweise kann eine Einteilung in verschiedene Gruppen wie Daten- und Organisations-Knoten im Cluster erfolgen. Die Daten-Knoten halten den Anwendungszustand bereit. Client-Zugriffe darauf oder die Verteilung cluster-intern werden von den Organisations-Knoten verwaltet.

Eine dritte Möglichkeit ist die Unterscheidung nach *Leader- und Follower-Knoten*. Hierbei sind alle Knoten identisch, jedoch folgen die Follower-Knoten dem Anwendungszustand eines Leader-Knotens. Änderungen an einem Follower-Knoten sind nicht möglich oder werden von diesem mit dem Leader-Knoten abgesprochen. Sind alle Knoten vom gleichen Typ und auch gleichberechtigt handelt es sich um eine sog. *aktive Replikation*. Im Fall des Leader/Follower-Modells handelt es sich um *passive Replikation* [14]. Die Unterscheidung kommt daher, dass bei aktiver Replikation sämtliche Knoten aktiv teilhaben müssen, während bei passiver Replikation die optional vorhandenen Follower-Knoten passiv vom Leader-Knoten hinsichtlich der Aktualisierung des Anwendungszustands gesteuert werden. Leader/Follower werden auch als Master/Slave bezeichnet.

Bei Redundanz im Fall einer Leader/Follower- bzw. Master/Slave-Architektur im Cluster existieren *Failover-Strategien*, die das Verhalten im Fall des Ausfalls eines Master-Knotens beschreiben. Fällt ein Master aus, übernimmt einer der Slaves die Rolle des Masters. Abhängig von der Aktualität des Slaves ist das verteilte System sofort oder erst nach Einspielen des Anwendungszustands z.B. aus einem Backup wieder verfügbar. Zudem ist mit einem aktuellen Slave kein oder kaum Verlust zu erzielen, während beim Einspielen eines Backups ein veralteter Anwendungszustand hergestellt wird. Als Failover-Strategien stehen *Cold-*, *Warm-* und *Hot-Standby* zur Verfügung. Cold-Standby geht davon aus, dass kein Slave den aktuellen Anwendungszustand bereit hält, sodass ein neuer Master zunächst durch Kopiervorgänge hergestellt werden muss. Warm-Standby geht von einem fast aktuellen Slave aus, der lediglich keinen vollständigen synchronen Anwendungszustand bereit hält, der jedoch z.B. durch erneutes Einspielen der vergangenen Operationen hergestellt werden kann. Hot-Standby geht von einem vollwertigen aktuellen Anwendungszustand aus, sodass ein Slave sofort als Master bereit stehen kann.

Umgang mit Konflikten

Die Synchronisierung in verteilten Systemen ist notwendig, damit parallel arbeitende Knoten einen gemeinsamen Anwendungszustand konsistent bereitstellen können. Kapitel 2.1.1 führt dazu pessimistische und optimistische Zugriffsverfahren ein. Beim pessimistischen Ansatz werden Transaktionen verwendet, deren Anfragen blockiert oder abgebrochen werden, sofern ein Konflikt droht. Das verteilte System hält eine einzige Version bereit, deren ständige Konsistenz gewahrt wird. Beim optimistischen Ansatz werden vom verteilten System mehrere Versionen bereit gestellt, die wieder zu einer Version vereinigt werden oder als Zugriffsverlauf einzeln bestehen bleiben. Transaktionen können auch beim optimistischen Ansatz verwendet werden, wobei im

Fall eines Konflikts die Transaktion abgebrochen und zurückgesetzt wird. Im Folgenden werden die beiden Verfahren aus Sicht der Client-Anfrage als *Locking* und *Versionierung* bezeichnet, wobei abgebrochene Transaktionen aufgrund verlängerter Verarbeitungszeiten dem Locking-Verfahren zugeordnet werden.

Um Konflikte bei Locking im Voraus nicht aufkommen zu lassen, kann der redundant gespeicherte Anwendungszustand auf allen Kopien gesperrt oder sogar bis auf eine Kopie gelöscht werden. Dieses Locking-Verfahren benötigt, zusätzlich zum eigentlichen Zugriff, weitere Kommunikationsverfahren, in denen sich die jeweiligen Knoten absprechen und den Wunsch einer Sperrung mitteilen. Ein mögliches Protokoll dafür ist das bereits genannte Zwei-Phasen-Commit Protokoll (2PC). Durch den zusätzlichen Nachrichtenaustausch und die Sperre auf den jeweiligen Knoten entstehen zusätzliche Latenzzeiten der Anfrage an das System. Dafür können parallele Schreibzugriffe nicht stattfinden, sondern werden so lange blockiert, bis die Sperre aufgehoben ist. Die Anfragen werden also sequentiell abgearbeitet. Das Konfliktmanagement findet im Fall von Locking während der Verarbeitung der Anfragen statt.

Gegenüber Locking existiert die Versionierung. Jeder Schreibzugriff findet lokal auf einer neuen Kopie statt und wird mit einer eindeutigen Versionskennung versehen. Finden parallele Zugriffe auf verschiedenen Kopien statt, werden die Anfragen sofort bearbeitet statt blockiert. Entstehende Konflikte können später anhand der Versionskennung erkannt und auf den jeweiligen Kopien deterministisch oder manuell gelöst werden. Neben der sofortigen Ausführung der Anfrage kann gegenüber dem Locking eine zusätzliche Kommunikation während der Ausführung des Zugriffs verzichtet werden. Dadurch können wesentlich geringere Latenzzeiten für die Anfrage entstehen, da das System intern das Konfliktmanagement unabhängig von der Anfrage realisieren kann. Hierbei sind ggfs. zusätzliche Kommunikationen notwendig, die jedoch die Verarbeitung nicht beeinflussen. Bei Versionierung kann auf jedem Knoten nur die aktuellste Version vorliegen und im Cluster verteilt nach parallelen Zugriffen verschiedene Versionen gleichzeitig existieren. Alternativ kann jeder Knoten einen vollständigen Verlauf der Versionen bereitstellen. Die Versionskennung wird durch Zeitstempel wie absolute Zeitstempel, Lamport Uhren oder Vektor Uhren realisiert.

Umgang mit Konsistenz

Die Betrachtung von Konsistenz muss zunächst in zwei verschiedene Sichtweisen unterteilt werden. Die traditionelle Datensicht betrachtet *datenzentrierte Konsistenzmodelle*, bei denen die Speicherung der Daten im Mittelpunkt steht. Dem gegenüber stehen *clientzentrierte Konsistenzmodelle*, welche die Sichtweise auf einen Datensatz aus Anwendungssicht als Perspektive wählen [27]. Die datenzentrierte Sichtweise ist sehr viel strenger und muss den Zugriff auf den Datensatz kontrollieren. Die clientzentrierten Modelle akzeptieren vorübergehende Inkonsistenzen innerhalb des verteilten Systems, solange ein Client ein konsistentes Verhalten zu sehen bekommt.

Zu den wichtigsten *datenzentrierten Konsistenzmodellen* gehören in verschieden starker Konsistenzanforderung die *strenge Konsistenz*, die *sequentielle Konsistenz* bis hin zur kausalen und letzendlich schwachen Konsistenz. Strenge Konsistenz ist der Idealfall, allerdings bei verteilten Systemen aufgrund der Nebenläufigkeit auf verschiedenen Knoten nicht zwingend notwendig. Stattdessen wird meist auf die sequentielle Konsistenz ausgewichen, welche die gleiche Ausführungsreihenfolge von Schreibzugriffen gewährleistet. Die kausale Konsistenz schränkt diese

Garantie noch weiter ein, sodass nur voneinander abhängige Schreibzugriffe in gleicher Reihenfolge auftreten. Zuletzt garantiert die schwache Konsistenz nicht zu jedem Zugriff, sondern zu gesetzten Synchronisationspunkten eine sequentielle Konsistenz. Zusätzlich existieren noch datenzentrierte Konsistenzmodelle wie Eintritts- und Freigabe-Konsistenz, die vor oder nach einem Schreibzugriff die Konsistenzgarantie wahren und bspw. auf Anwendungsebene definiert werden kann.

Die *clientzentrierten Konsistenzmodelle* vereinfachen die Betrachtung der Konsistenz auf einen Datensatz gegenüber der datenzentrierten Sichtweise, da nur ein Client statt der Datensatz mit eventuell mehreren Clients betrachtet wird. Diese Sichtweise geht von einer selteneren Aktualisierung eines Datensatzes und von einer hohen Skalierbarkeitsanforderung aus, da von Nebenläufigkeit ausgegangen wird. Als grundlegendstes clientzentriertes Konsistenzmodell gilt die *Eventual Consistency* (engl., endliche Konsistenz) [29]. Bei diesem Ansatz wird von einzelnen, oft zentralen, Schreibzugriffen und überwiegend verteilten Lesezugriffen ausgegangen. Ein geänderter Datensatz wird zu irgendeinem unbestimmten späteren Zeitpunkt auf allen Replikaten einen konsistenten Zustand haben. Reicht dieses Konsistenzmodell nicht aus, können entsprechend verfeinerte Modelle Verwendung finden. Monotones Lesen beispielsweise garantiert, dass ein Client mit einem gelesenen Datensatz bei späterem erneuten Lesezugriff keine ältere Version als die eigene bekommt. Monotones Schreiben garantiert, dass die Reihenfolge der Schreibzugriffe eines Clients erhalten bleibt. Besonders interessant, wenn ein Zugriff willkürlich ein Replikat auswählen kann, ist das Modell Read-your-writes, welches garantiert, dass nach einem Schreibzugriff ein Lesezugriff auch die eigenen geschriebenen Werte enthält. Ein weiteres Problem sind sog. Lost-Updates, bei denen ein Schreibzugriff eines Clients exakt zwischen einem Lese- und einem darauf aufbauenden Schreibzugriff eines anderen Clients erfolgt und dadurch unbemerkt verloren geht. Das Modell Writes-follows-Reads löst das Problem, indem beim Schreibzugriff die Version oder der Wert des vorherigen Lesezugriffs mitübertragen und auf Aktualität überprüft wird.

Weshalb überhaupt in verteilten Systemen teilweise auf eine strenge Anforderung der Konsistenz verzichtet wird, ist mit dem *CAP-Theorem* [6] zu erklären. Dieses besagt, dass nur zwei der drei Kriterien *Consistent* (Konsistent), *Available* (Verfügbar) und *Partition tolerant* (Tolerant gegenüber Netzwerkpartitionen) von einem verteilten System vollständig realisiert werden kann (vgl. Abb. 2.4).

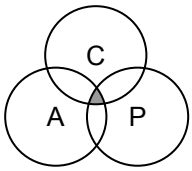

Abbildung 2.4.: CAP-Theorem: der Schnitt aller drei Kriterien ist nicht erreichbar.

Das Modell beschreibt den Schwerpunkt eines verteilten Systems im Fall einer auftretenden Netzwerkpartition. Allerdings darf das Modell nicht als eine strenge Auswahl von zwei der drei Kriterien für die Architektur eines verteilten Systems verstanden werden. Vielmehr kann durch teilweises Reduzieren eines Kriteriums ein anderes teilweise realisiert werden. So führt eine weniger strenge Konsistenz eines gegen Netzwerkausfälle tolerantes System im Fall eines Ausfalls zu erhöhter Verfügbarkeit [16]. Die Schwachstelle des CAP-Theorems liegt allerdings darin, dass

nur der Fall eines Netzwerkproblems und nicht der reguläre Anwendungsfall betrachtet wird. Das alternative *PACELC*-Modell [1] definiert hierfür diese beiden Fälle und welche Schwerpunkte jeweils gelten. Im Fall einer Partition (P) kann das verteilte System verfügbar oder konsistent sein (A/C), ansonsten (else, E) den Schwerpunkt auf kurze Latenzzeiten oder Konsistenz der Verarbeitung von Anfragen haben (L/C). Gegenüber dem CAP-Theorem wird zudem das Kriterium der Latenzzeit eingeführt, welche im Modell im Gegensatz zur Verfügbarkeit oder Konsistenz steht. Mögliche Kombinationen sind also PAEL / PAEC mit Verfügbarkeit im Fall einer Partition und sonst geringen Latenzzeiten oder Konsistenz. Andererseits PCEL / PCEC mit Wahrung der Konsistenz im Fall einer Partition.

Redundanz und Consistent Hashing

Skalierung und Redundanz sind zunächst voneinander unabhängige Techniken zur Realisierung von Fehlertoleranz. Skalierung und Redundanz kann kombiniert werden, um Vorteile beider Techniken zu nutzen. Während Skalierung eine Anwendung im verteilten System verteilt, sorgt die Redundanz für Kopien. Redundanz ohne Skalierung entspricht vollständiger Replizierung, da alle Knoten den vollständig gleichen Anwendungszustand verwenden. Im Kombination mit Skalierung ergibt sich partielle Replizierung, da jeder Knoten einen Teil des verteilten Systems oder eine Kopie eines Teils darstellt.

Consistent Hashing sorgt für Skalierung durch Hash-Funktionen und repräsentiert die Verteilung durch einen Ring (vgl. Abb. 2.3). Jeder Knoten im Ring ist für einen Teil des Systems zuständig. Das Consistent Hashing Verfahren kann um partielle Replizierung erweitert werden, in dem definierbar viele im Uhrzeigersinn nachfolgende Knoten eines Knotens diesen replizieren. Mit bspw. insgesamt zwei Replikaten würde im Beispiel der Abb. 2.3 Knoten S1 den Knoten S3, S2 replizieren. Fällt ein Knoten aus, wird automatisch aufgrund der Strategie der Partitionierung ohne zusätzlichen Aufwand ein zuständiges Replikat verwendet. Die Replikate können je nach verwendetem Verfahren aktive Replikation oder passive Replikation verwenden.

2.3. Datenbanksysteme

Nachdem bisher verteilte Systeme grundlegend erklärt wurden, beschäftigt sich dieses Kapitel mit speziellen Systemen, die verteilt auftreten können. Der Schwerpunkt liegt weniger auf der Ausführung von Prozessen sondern viel mehr auf den zu speichernden Daten. In vorherigen Kapiteln wurden verteilte Systeme vorgestellt und auf Skalierung und Redundanz eingegangen. Die entstehenden Herausforderungen wie Konflikte und Konsistenz werden im folgenden Kapitel 2.3.1 erneut betrachtet, jedoch im speziellen Bezug auf Datenbanksysteme.

Datenbanksysteme speichern die enthaltenen Daten intern sehr verschieden ab. Zudem kann der Zugriff von Clients an das Datenbanksystem unterschiedlich stattfinden. Kapitel 2.3.2 präsentiert Kategorien, die Datenbanksysteme unterschiedlichen Speichertyps zugeordnet werden können. Abschließend werden die Kategorien in Kapitel 2.3.3 im Hinblick auf ihre Schwerpunkte gegenübergestellt.

2.3.1. Konflikte und Konsistenz

In Kapitel 2.2.2 wurde Redundanz und die dadurch entstehenden Herausforderungen von Konflikten und der Konsistenz eingeführt. Ergänzend zum vorgestellten Umgang mit Konflikten kann bei Datenbanksystemen mit einzelnen Datensätzen die Versionskennung durch Timestamps, Sequenznummern oder Vektor Uhren realisieren. Timestamps sind absolute Zeitangaben und benötigen daher eine zuverlässige, synchronisierte Uhrzeit jedes teilnehmenden Knotens im verteilten System. Sequenznummern, die für jeden Datensatz bei jeder Änderung bswp. hochgezählt werden können, bei parallelen Zugriffen, möglicherweise unbemerkt in Konflikt zueinander stehen. Vektor Uhren kodieren daher in die Versionskennung bspw. die Kennung des eigenen Knotens und eine fortlaufende Sequenznummer ein. Hierbei wird versucht, eine eindeutige Versionskennung zu erzeugen. Versionsbasierte Systeme sind üblicherweise *non-blocking*, hingegen sind lock-basierte Systeme als *blocking* zu bezeichnen. Anwendungsspezifisch ist diese Eigenschaft ein Entscheidungskriterium bei der Wahl eines zu verwendenden Datenbanksystems, da die Verarbeitungszeiten von Anfragen bei non-blocking und häufig parallelen Zugriffen kürzer sind. Hingegen kann bei einer Anwendung ein auftretender Konflikt hinderlich sein, wenn hohe Konsistenzanforderungen vorliegen.

Um optimale Konsistenzanforderung zu erhalten wird in Datenbanksystemen das *ACID-Modell* eingesetzt. Es definiert, dass Anfragen an das System atomar (Atomicity), konsistent (Consistency), isoliert (Isolation) und dauerhaft (Durability) bearbeitet werden. Die vier Kriterien werden in Verwendung von Transaktionen interessant, welche mehrere Anfragen zusammenfassen und ohne Unterbrechung durch nebenläufige Anfragen in einer Abfolge atomar und isoliert abgearbeitet werden. Zudem wird in ACID-basierten Systemen als Konfliktbehandlung Locking eingesetzt, sodass durch Blockieren oder Abbrechen von Transaktionen eine strenge Konsistenz gewährleistet wird. Im Bereich der Datenbanksysteme mit persistent zu haltenden Daten ist das Kriterium Dauerhaftigkeit wichtig, da diese Eigenschaft eine garantierte persistente Speicherung der Daten zusichert. Im Fall von ACID sichert das Datenbanksystem nach erfolgreichem Abschluss einer Transaktion eine garantierte Persistierung der veränderten Daten zu.

Diesem streng konsistenten Modell steht ein Modell für sehr skalierbare und fehlertolerante Datenbanksysteme gegenüber. Das *BASE-Modell* steht für "Basically Available, Soft State, Eventual consistency". Die Schwerpunkte liegen hierbei auf Verfügbarkeit und Toleranz bei Ausfällen und reduziert die Konsistenzanforderung auf Eventual Consistency [15].

2.3.2. Kategorien von Datenbanksystemen

Datenbanksysteme unterscheiden sich in der Form der Speicherung von Daten. Eine grobe Einteilung von Datenbanksystemen kann in *relationale Datenbanksysteme* und in *nicht-relationale Datenbanksysteme* erfolgen. Zu den nicht-relationalen Datenbanksystemen lassen sich die sog. *NoSQL Datenbanksysteme* einordnen. Während bei relationalen Datenbanksystemen zur Manipulation und Abfrage die *Structured Query Language* (SQL) verwendet wird, setzen NoSQL Datenbanksysteme auf Alternativen. Während bei relationalen Datenbanksystemen der Speichertyp tabellarisch erfolgt, werden unter dem Begriff der NoSQL Datenbanksystemen viele verschiedene Speichertypen eingeordnet. Zu den Kerntypen gehören Spalten-orientierte, Graphenbasierte, Key/Value-basierte und Dokumenten-orientierte Datenbanksysteme [15, 26, 24]. Die

nachfolgenden Kapitel beschreiben je eine der genannten Speichertypen von sowohl relationalen als auch den NoSQL Datenbanksystemen. Dabei wird je auf die Speicherstruktur eingegangen und Unterschiede untereinander herausgearbeitet.

Relationale Datenbanksysteme

Relationale Datenbanksysteme sind die am weitesten verbreitetsten. Seit der Entwicklung im Jahr 1970 hat das relationale Datenbankmodell einen Siegeszug gegenüber alternativen Modellen begonnen und ist etabliert. Im Hintergrund steht ein Modell mathematisch wohldefinierter Relationen und Operationen auf enthaltene Daten.

Die Speicherung von Daten in relationalen Datenbanken findet in fest definierten Schemata statt. Die Organisationsstruktur sieht zunächst Datenbanken vor, in denen sich Tabellen mit fest definierten Spalten befinden. Spalten verfügen über einen Datentyp und einen Namen. Eingetragene Daten liegen als Zeilen oder sog. Tupel einer Tabelle vor. Um komplexere Strukturen abzubilden, können Tabellen untereinander Datenbankübergreifend gegenseitig referenziert werden. Üblicherweise hat eine Tabelle einen Primärschlüssel, dessen Wert einer Zeile in einer externen Tabelle als Fremdschlüssel in einer Spalte eingetragen wird.
Die Persistierung der Tabellen ist vom Datenbanksystem abhängig und findet bspw. durch baumartige Strukturen statt. Um Anfragen auf Tabellen zu optimieren, können auf ein oder mehrere Spalten sog. indizes gelegt werden. Sie bieten eine redundante, separat aufbereitete Speicherung der Spalteninhalte an, auf denen schneller Datensätze anhand eines Wertes gefunden werden können.

Der Datenzugriff in relationale Datenbanksysteme findet in der Regel über die *Structured Query Language* (SQL) statt, die durch Standards einheitlich definiert wird. Umgekehrt ist SQL mit vollem Funktionsumfang an ein relationales Modell gebunden. Ein Anwendungsentwickler definiert einen Zugriff als Zeichenkette, die zur Laufzeit seiner Anwendung an das Datenbanksystem übertragen wird. Das System selbst kann die SQL-Anfrage optimieren und ausführen. SQL ist z.B. für Schreib- oder Lesezugriffe aber auch für Schemaänderungen definiert. Lesezugriffe über mehrere Tabellen, die z.B. über Fremdschlüssel schematisch miteinander verbunden sind, können in SQL über JOINs verbunden werden, sodass das ausführende Datenbanksystem als Resultat Tupel mit Spalten aus möglicherweise beiden Tabellen haben. Das Resultat eines Lesezugriffs kann üblicherweise als ein zweidimensionales Konstrukt betrachtet werden, welches das Datenbanksystem nach Ausführung der SQL-Anfrage an die aufrufende Anwendung meist synchron zurück gibt.

Eine Folge von Zugriffen mit SQL kann bei relationalen Datenbanksystemen häufig als Transaktion zusammengefasst werden. Die Transaktion wird vom Datenbanksystem als eine atomare Anfrage isoliert ausgeführt. Dadurch wird gewährleistet, dass z.B. die Abfolge eines Lese- und daraus resultierenden Schreibzugriffs nicht von einem zwischenzeitlich eintreffenden Schreibzugriff beeinträchtigt wird und ein ungewolltes Ergebnis eintritt.
Zudem nutzen relationale Datenbanken üblicherweise Locks auf Zeilen oder ganzen Tabellen, um konkurrierende Zugriffe abzufangen. Die entstehende Verzögerung eines Zugriffs auf einen gelockten Eintrag hängt vom entsprechenden Zugriff ab, der den Lock verursacht, und kann erheblich sein.

Spalten-orientierte Datenbanksysteme

Während bei relationalen Datenbanksystemen zeilenweise gespeichert wird, findet die Anordnung bei Spalten-orientierten Datenbanksystemen in Spalten statt. Die Einteilung erfolgt ebenso in Tabellen und Spalten, jedoch mit Schwerpunkten auf Spalten statt auf Zeilen. Auf den ersten Blick aus Anwendersicht sehen die beiden Modelle sehr ähnlich aus, unterscheiden sich aber sehr deutlich. Spalten-orientierte Datenbanksysteme verzichten üblicherweise auf fest definierte Schemata, sodass Zeilen innerhalb einer Tabelle nicht immer die selben Spalten beinhalten müssen. Eine Tabelle ist daher weniger als eine strikte Vorgabe von Zeilen und Spalten zu betrachten, sondern als eine Ansammlung von semantisch zusammengehörigen Zeilen mit variablen Spalten. Dennoch existieren Spalten-orientierte Datenbanksysteme, die einfaches SQL zur Datenmanipulation nachempfunden haben [15].

Durch die Aufteilung eines Datensatzes anhand seiner Spalten, liegen die gespeicherten Daten weniger untereinander verknüpft vor. Zudem sind Felder innerhalb einer Spalte nach bspw. dem Primärschlüssel eines Datensatzes sortiert. Diese beiden Kriterien ermöglichen eine einfache Aufteilung und dadurch Partitionierung der gespeicherten Daten. Gleichzeitig steht jedoch der Anwendung eine tabellarische Struktur zur Verfügung. Spalten-orientierte Datenbanksysteme sind eine gute Alternative zu relationalen Datenbanksystemen, wenn auf die tabellarische Arbeitsweise nicht vollständig verzichtet werden soll, jedoch eine hohe horizontale Skalierbarkeit erforderlich ist.

Besonders geeignet ist die Spalten-orientierte Speicherung, wenn Lese- oder Schreibzugriffe nicht auf komplette Datensätze sondern auf eine Teilmenge der Felder einiger Datensätze erfolgt. Dadurch, dass intern aufeinander folgende Daten verschiedener Datensätze der selben Spalte unmittelbar benachbart gespeichert werden, kann der Zugriff vom Datenbanksystem schneller abgearbeitet werden als bei relationalen Datenbanksystemen. Ein weiterer Vorteil ergibt sich aus dem Verzicht von festen Schemata. Während bei relationalen Tabellen eine Veränderung der Spalten auf sämtliche enthaltene Zeilen erfolgen muss, kann die Veränderung im Spalten-orientierten Modell nur bei neu hinzukommenden oder aktualisierten Datensätzen erfolgen. Besonders bei sehr vielen Datensätzen kann diese Unterscheidung ein kritisches Kriterium sein, da Schemaänderungen bei relationalen Datenbanksystemen meist einen Lock auf die gesamte Tabelle bis zum Abschluss der Änderung zur Folge haben, während Anfragen blockiert werden.

Graphen-basierte Datenbanksysteme

Bei Graphen-basierten Datenbanksystemen besteht das Datenbanksystem aus einem Graphen. Die enthaltenen Daten befinden sich dabei in den Knoten, Kanten oder in Attributen. Dieser Speichertyp unterscheidet sich daher grundlegend von anderen. Zugriffe auf die Daten entsprechen dem Navigieren durch den Graphen, sodass dieser Speichertyp besonders für eng miteinander verknüpfte Daten Anwendung findet. Der entstehende Graph ist jedoch schwer zu skalieren, da eine Partitionierung möglicherweise die Daten korrupiert. Zudem ist dieser Speichertyp exotisch, weshalb im Folgenden der Arbeit Graphen-basierte Datenbanken nur oberflächlich vorkommen werden.

Die Graphen-basierte Speicherung findet daher überwiegend in überschaubar großen Datenvolumen statt, deren Datensätze untereinander eng miteinander verknüpft sind. Durch die navigierende Zugriffsart ist eine Abfrage einfacher realisierbar als z.B. mit verketteten relationalen Tabellen. Während bei SQL bspw. vor Ausführung einer Anfrage bekannt sein muss, wie viele Schritte einer verknüpften Datenspeicherung durch entsprechende JOIN-Operationen getätigt werden müssen, kann bei Graphen-basierten Datenbanksystemen die Abfrage dynamisch entlang der vorliegenden Daten erfolgen.

Key/Value-basierte Datenbanksysteme

Die Datenbanksysteme, derer Speicherung einfacher Key/Value-Paare entspricht, ist die älteste Form von Datenbanksystemen. Sie wurden aufgrund zu einfacher Strukturen von den relationalen Datenbanksystemen verdrängt, gelangen jedoch durch die Zunahme an Datenmengen wieder an Bedeutung. Die Speicherung eines Datensatzes findet über einen eindeutigen Identifier statt. Die Datenbanksysteme dieses Speichertyps verwenden häufig sog. Buckets, in denen Key/Value-Paare gespeichert werden. Durch diese Organisation können ähnliche Werte-Paare in einem Bucket gruppiert und gesammelt werden, die aus Sicht der Anwendung semantisch zusammen gehören. Der gespeicherte Wert kann von einer einfachen Zeichenkette bis hin zu größeren Binärdaten für bspw. Videodaten genutzt werden. Der Zugriff auf gespeicherte Daten findet hauptsächlich über den Key als Identifier statt und wird über eigene Netzwerkprotokolle oder z.B. über REST und dem HTTP-Protokoll getätigt.

Diese einfach gehaltene Speicherung ermöglicht die Umsetzung performanter Anwendungen mit hohen Anforderungen an geringe Abfragezeiten. In Kombination mit geringeren Speichergrößen findet die Speicherung der Key/Value-Paare häufig RAM-basiert statt, um die Zugriffszeiten gegenüber der Speicherung auf Festplatte zu verkürzen. Ein Nachteil dieser Speicherform sind zu einfache oder fehlende Filter- oder Suchmöglichkeiten der Datensätze, da der gespeicherte Wert beliebigen Typs sein kann und ggfs. vom Datenbanksystem nicht interpretiert wird. Da überwiegend über einfache Get- und Set-Operationen zugegriffen wird, können Key/Value-basierte Datenbanksysteme optimal zur Zwischenspeicherung während eines Programmablaufs statt zur endgültigen Persistierung verwendet werden. Key/Value-basierte Datenbanksysteme sind eng mit den nachfolgenden Dokumenten-orientierten Datenbanksystemen verknüpft.

Dokumenten-orientierte Datenbanksysteme

Dokumenten-orientierte Datenbanksysteme sind im Wesentlichen erweiterte Key/Value-basierte Datenbanksysteme. Die gespeicherten Werte haben neben einem Identifier einen Inhalt, der einem fest definierten strukturierten Beschreibungsformat wie z.B. der JavaScript Object Notation (JSON) folgt. Eine Definition, welche Felder im Dokument erforderlich sind, ist unüblich, sodass kein festes Schema definiert wird. Teilweise können durch Regeln, die auf ein Dokument bei Schreibzugriffen angewendet werden, schemaähnliche Verhaltensweisen wie die Existenz und Gültigkeit einzelner Felder erzielt werden. Ein Dokument wird, ähnlich wie bei einer Key/Value Datenhaltung, über einen Identifier eindeutig gekennzeichnet. Optional sind Indizierungen über Inhalte der Dokumente möglich, um in einer Menge von Dokumenten performant Lesezugriffe zu ermöglichen. Eine Menge von gleichartigen Dokumenten kann in Datenbanken

oder Collections geordnet werden. Lesezugriffe finden üblicherweise über den Identifier, über einen angelegten Index oder über das sog. Map/Reduce-Verfahren[12] statt. Die Speichertypen Key/Value-basiert und Dokumenten-orientiert sind so ähnlich, dass existierende Datenbanksysteme teilweise in der Literatur wechselnd zugeordnet oder von den Entwicklern selbst in beide Kategorien eingeordnet werden. Ein wesentlicher Unterschied ist die Vergabe fester Beschreibungsformat für die enthaltenen Werte. Bei Key/Value-basierten Datenbanksystemen ist das Format beliebig, sodass das Datenbanksystem keine Interpretationen der enthaltenen Daten tätigen kann. Die Strukturierbarkeit eines Dokuments wird vom Datenbanksystem durch die Definition des Beschreibungsformats gewährleistet.

Besonders JSON-basierte Dokumenten-orientierte Datenbanksysteme sind bei Anwendungen mit webbasierten Benutzerschnittstellen beliebt. Durch die im Browser verwendete Programmiersprache JavaScript lassen sich mit JSON formatierte Daten optimal weiterverarbeiten. Die zunehmende Verwendung von JavaScript auf Serverseite mittels Node.js vervollständigt die Verwendung von JavaScript in allen Bereichen einer Anwendung. Gegenüber relationalen Datenbanksystemen fällt die bidirektionale Wandlung der Daten zwischen Anwendung und Datenbanksystem durch die Verwendung eines einheitlichen Formats weg. Zudem werden Datensätze in Dokumentenform nicht wie bei relationalen Datenbanksystemen normalisiert. Die Normalisierung zerlegt einen semantisch zusammengehörigen Datensatz in einzelne Datensätze, die logisch trennbar sind und bei separater Speicherung Duplikate vermeidet. Dieser Vorgang hat zur Folge, dass zusammengehörige Daten vom Datenbanksystem bei einer Anfrage zusammen gesucht werden müssen. Bei Dokumenten-orientierten Datenbanksystemen werden auf Kosten von Duplikaten Daten nicht zerlegt und sind dadurch wesentlich schneller abrufbar.

2.3.3. Gegenüberstellung der Kategorien

Zunächst unterscheiden sich die vorgestellten Datenbanksysteme im Speichertyp und in den Abfragetechniken. Die Schwerpunkte liegen allerdings, abhängig vom betrachteten Datenbanksystem, teilweise sehr weit auseinander. Allgemein, während relationale Datenbanksysteme durch Transaktionen und Locks eine sehr strenge Konsistenz der Daten gewährleisten, zielen die meisten der NoSQL Datenbanksysteme auf eine bessere Skalierbarkeit und kürzere Verarbeitungszeiten von Datenzugriffen. Die unterschiedlichen verwendeten Modelle wie bei relationalen Datenbanken meist ACID oder bei NoSQL Datenbanken meist BASE beschreiben eben diese Schwerpunkte. Beide Modelle haben ihre Anwendungsbereiche, abhängig vom verwendeten Schwerpunkt, der beispielsweise anhand des CAP-Theorems mit CA für ein ACID-System oder AP für ein BASE-System definierbar ist. Anwendungsabhängig ist die Skalierbarkeit alleiniges Kriterium, wenn Zugriffszahlen oder Speichermengen mit relationalen Modellen schwerer und mit einem NoSQL-Ansatz leichter bewältigbar sind.

Im Bezug auf zu speichernde Datenmengen sind Unterschiede bemerkbar. Während sich beispielsweise Spalten-orientierte Datenbanksysteme erst ab einem zu speichernden Datenvolumen von mehr als einigen Gigabyte empfehlen [7], sind relationale Datenbanksysteme auch in kleineren Anwendungsfällen sinnvoll. Umgekehrt kann eine relationale Datenbank ab einem größeren Volumen schwieriger ausreichend performant im Hinblick auf Anfragen und Datenvolumen skaliert werden. Grund für die schwierigere Skalierbarkeit von relationalen oder auch Graphen-basierten Datenbanksystemen ist die hohe Herausforderung der Partitionierung, wenn eine vertikale oder

funktionale Skalierung nicht ausreicht. Einzelne, voneinander unabhängige Datentupel wie in einem Key/Value-basierten System sind hingegen z.B. anhand des Keys sehr einfach verteilbar und dadurch partitionierbar. Zudem besteht eine Datenbankanfrage in SQL aus Verknüpfungen mit Bedingungen auf Basis mehrerer Tabellen. Diese SQL-Anfragen verteilt auszuführen ist nicht performant oder überhaupt nicht realisierbar, wenn Tabellen verteilt und zerstreut vorliegen.

2.4. Virtual Nodes

Virtual Nodes [14] ist ein Framework zur Implementierung fehlertoleranter, verteilter Anwendungen. Das auf Java-basierte Komponenten-orientierte Framework ist durch die Dissertation von Jörg Domaschka entstanden und deckt z.B. den Bereich der Replizierung eines verteilten Systems ab. Abbildung 2.5 stellt eine Beispielanwendung mit einem Client und einem Cluster bestehend aus zwei Servern dar. Die Service Implementierung kennt eine client- und eine serverseite, die jeweils Virtual Nodes verwendet. Die Anwendung wird von einer Java Lautzeitumgebung (Runtime Environment, RE) ausgeführt, die auf einem Betriebssystem aufsetzt. In diesem Kapitel werden die Grundlagen von Virtual Nodes erklärt, um in Kapitel 6 die Implementierung eines Datenbanksystems auf dessen Basis zu erläutern.

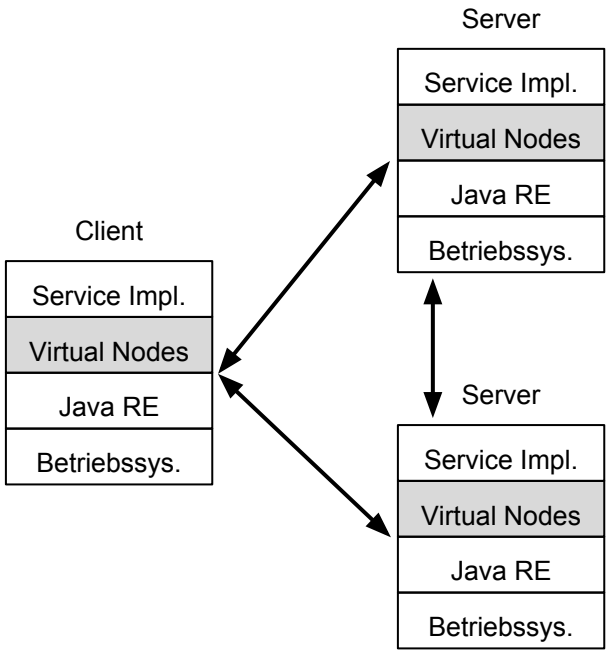

Abbildung 2.5.: Einordnung von Virtual Nodes in verteilten Systemen

Virtual Nodes definiert zunächst Server- und Client-Knoten. Server sind Teil des Clusters, welches auf Basis von Virtual Nodes den implementierten Service ausführt. Clients hingegen sind Knoten, die den Service in Verwendung von Virtual Nodes nutzen. Das Framework setzt zur Gruppenkommunikation aller beteiligter Knoten jGroups[1] ein, welches notwendige Grundfunktionen für den Aufbau eines netzwerkbasierten Clusters wie bspw. Hinzufügen oder Entfernen von

[1] http://www.jgroups.org/

Knoten oder eine zuverlässige Nachrichtenübertragung bietet. Virtual Nodes abstrahiert diese Funktionalität durch den sogenannten *MemberManager*, der als geordnete Liste sogenannte Views mit Knoten im Cluster bereitstellt. Über *Listener* können Komponenten innerhalb von Virtual Nodes über Ereignisse des MemberManagers informiert werden, um bspw. entsprechend das Cluster anzupassen. Die Abstraktion der Bewegungen im Cluster wird durch das *JoinProtocol* als Komponente in Virtual Nodes implementiert. Intern findet hier z.B. im Fall eines neuen Knotens die Initialisierung der Übertragung des aktuellen Zustands der Anwendung statt.

Abbildung 2.6 stellt eine vereinfachte Form der schematischen Aufteilung der wichtigsten Komponenten von Virtual Nodes dar. Im Hintergrund befindet sich das Framework selbst, welches Aufgaben wie dem beschriebenen JoinProtocol oder dem Entgegennehmen von Client-Anfragen (dargestellter Pfeil) erfüllt. Die erste Komponente ist die Replikation. Sie ist für die Verteilung des Zustands und die Ausführung von Client-Anfragen zur Laufzeit der Anwendung verantwortlich. Optional kann diese Komponente eine Persistenz-Komponente beinhalten. Schließlich kommuniziert die Replikation über die Middleware mit dem implementierten Service. Als Middleware kann ein RMI-Interface gegenüber der Service-Implementierung genutzt werden. Für die Entwicklung einer Anwendung auf Basis des Frameworks muss daher kein tiefer Einblick in Virtual Nodes, sondern lediglich Kenntnis zur Nutzung von RMI vorhanden sein.

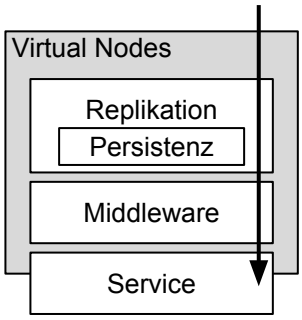

Abbildung 2.6.: Komponenten der Software-Architektur eines serverseitigen Knotens

Bevor ein Client an ein Cluster eine Anfrage auf Basis von Virtual Nodes stellen kann, muss stellvertretend ein geeigneter Server-Knoten gewählt werden. Die Strategie der Wahl eines Server-Knotens im Cluster kann vor der Ausführung des Clients konfiguriert werden. Einige der zur Verfügung stehenden Selektoren sind nur in Kombination mit einem der beiden zur Verfügung stehenden Replikationsimplementierungen sinnvoll (vgl. Kapitel 2.4.2). Zur Auswahl stehen der FairSelector, welcher gleichmäßig zufällig einen Server-Knoten aussucht, zu dem der Client verbunden wird. Der LastReplicaSelector verbindet den Client zum letzten verwendeten Replikat.

Im Folgenden wird die Realisierung von Skalierung und Replizierung im Virtual Nodes Framework erläutert, um fehlertolerante Anwendungen zu ermöglichen. Abschließend wird das Programmiermodell vorgestellt, welches die Implementierung des Frameworks in Komponenten aufteilt.

2.4.1. Skalierung

Im Folgenden wird das Hinzufügen eines neuen Knotens genauer betrachtet. Dieser verbindet sich zunächst als Client mit der Gruppenkommunikation, um eine aktuelle View des MemberManagers von Virtual Nodes zu erhalten. Anschließend versendet der neue Knoten per Broadcast eine GET-Nachricht, worauf der älteste und erste Knoten der View mit einer SET-Nachricht an den neuen Knoten antwortet. Diese SET-Nachricht enthält den Zustand der Anwendung, der beim Eintreffen der GET-Nachricht beim ältesten Knoten gesammelt und serialisiert wurde. Nachdem der neue Knoten den erhaltenen Zustand eingespielt hat, versendet dieser per Broadcast eine GOT-Nachricht wodurch alle vorhandenen Knoten über die Existenz des neuen, nun vollwertigen Knoten Bescheid wissen. Hierdurch wird auch die View des MemberManagers aktualisiert. Dieser Ablauf wird in Abbildung 2.7 als UML-Diagramm graphisch repräsentiert.

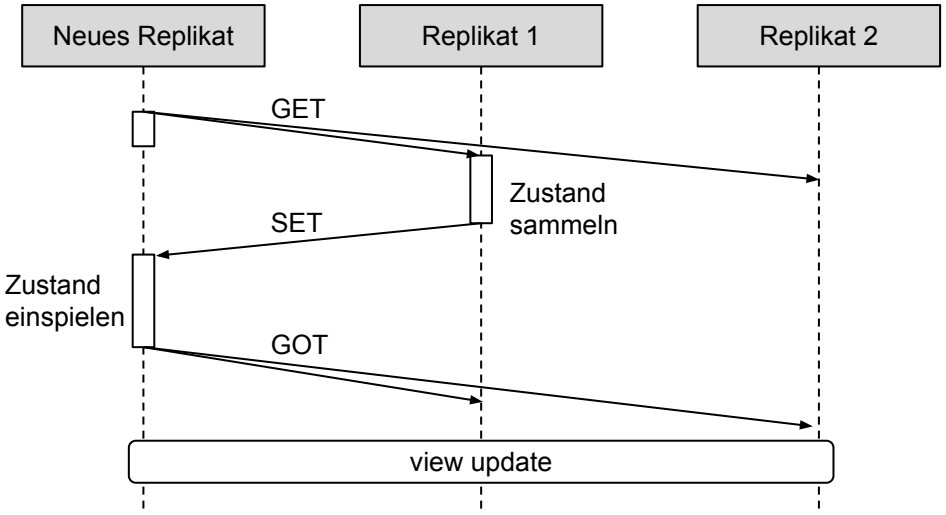

Abbildung 2.7.: Ablauf beim Hinzufügen eines neuen Knotens

Beim Sammeln des Zustands können einzelne Komponenten ihren Zustand beitragen. So kann anwendungsabhängig z.B. die Replikations-Schicht selbstständig beim Eintreffen einer GET-Nachricht am lokalen Knoten ihren eigenen Zustand serialisieren, der gemeinsam mit Zuständen anderer Komponenten wie dem MemberManager oder der Middleware den mit der SET-Nachricht übertragenenen Zustand bildet. Dabei ist es abhängig von der Implementierung der austauschbaren Komponenten, welche Informationen als Zustand definiert werden.

2.4.2. Replikation

Virtual Nodes definiert für die Implementierung der zentralen Komponente, der Replikation, aktive oder passive Replizierung. Bei der aktiven Replikation werden alle Knoten des Clusters als exakt gleich betrachtet und synchron repliziert, während bei der passiven Replikation ein Knoten als sog. Leader die Führungsrolle der Verarbeitung von Client-Anfragen übernimmt und die übrigen diesem folgen. Bei aktiver Replikation wird eine Anfrage also auf allen Knoten gleichermaßen ausgeführt, was im Fall eines Datenbanksystems eine strenge Konsistenz gewährleisten kann. Die passive Replikation von Virtual Nodes kann verschieden konfiguriert werden.

So kann die Client-Anfrage sofort oder nach einer einstellbaren Anzahl an anfallenden Anfragen nach erfolgreicher Verarbeitung auf dem Leader an alle Knoten verteilt werden. Der Zustand des Leaders kann entweder nach einem zeitlichen Intervall oder nach einer einstellbaren Anzahl an verarbeiteten Anfragen verteilt werden. Durch diese Konfigurationsmöglichkeiten kann im Bezug auf ein Datenbanksystem ein clientzentriertes Konsistenzmodell wie bspw. eventually consistent umgesetzt werden.

Da die spätere Implementierung dieser Ausarbeitung auf die passive Replikation aufbaut, wird im Folgenden genauer auf deren Implementierungsdetails eingegangen. Die passive Replikation hat die in Abbildung 2.6 dargestellte Subkomponente Persistenz. Die Komponente der Replikation ist für die Unterscheidung in Leader- und Follower-Knoten sowie die Ausführung von Client-Anfragen zuständig. Der Follower antwortet beim Eintreffen einer Client-Anfrage, die nicht als *Read-Only* von der Service-Implementierung deklariert wurde, mit der Aufforderung ein anderes Replikat zu wählen, da die Client-Anfrage nur vom Leader ausgeführt werden darf. Ein Scheduler koordiniert die Abarbeitung der Anfragen, sodass die Reihenfolge des Eintreffens mit der Reihenfolge der Abarbeitung übereinstimmt. Zudem akzeptiert der Scheduler nur das Ausführen einer einzigen Anfrage zur selben Zeit. Die Replikation ist am MemberManager registriert und definiert im Fall des Ausfalls des Leader-Knotens deterministisch einen neuen Leader.

Bei der passiven Replikation ist die Replikations-Komponente also für die korrekte Ausführung der Anfragen zuständig, während die Subkomponente, die Persistenz, für die Speicherung und Verteilung des Zustands und der Anfragen gemäß ihrer Konfiguration im Cluster zuständig ist. Die Persistenz kann dabei über zwei mögliche Verbindungswege ausgetauscht werden. Zum einen kann der Zustand in eine Datei geschrieben werden, auf die alle Knoten des Clusters Zugriff haben. Zum anderen kann die Verteilung über *PersistenceMessage*s über die Gruppenkommunikation und daher über das Netzwerk erfolgen. Abbildung 2.8 zeigt mit einer netzwerkbasierten Beispiel-Konfiguration den Ablauf einer Anfrage des Clients (C) an ein Cluster mit einem Leader- (S1) und zwei Follower-Knoten (S2, S3). Der Client wendet sich im Beispiel direkt an den Leader (1), der die Anfrage ausführt (2), den neuen Zustand an alle Knoten im Cluster verteilt (3) und anschließend das Ergebnis der ausgeführten Anfrage an den Client zurück sendet (4). Die Kommunikationsform ist also eine synchrone, push-basierte Übertragung, die eine strenge Konsistenz garantiert.

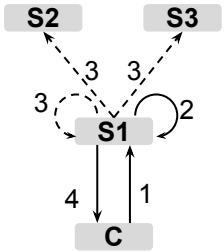

Abbildung 2.8.: Knoten-Kommunikation bei passiver Replizierung (Bsp. Konfiguration)

Für die Wahl eines Server-Knotens vor Verbindungsaufbau des Clients stehen im Fall einer passiven Replikation die beiden Selektoren FairNonLeaderSelector (verbindet zufällig zu einem Follower-Knoten) und LeaderSelector (verbindet mit dem Leader-Knoten) zur Verfügung.

2.4.3. Programmiermodell

Virtual Nodes ist Komponenten-orientiert aufgebaut, damit einzelne Bestandteile über Konfigurationen beim Starten des Systems oder während der Laufzeit anwendungsspezifisch geladen werden können. Die Komponenten stehen untereinander in Abhängigkeit, welche vom Framework vor der Ausführung der jeweiligen Komponenten aufgelöst wird. Eine Komponente definiert hierfür einen *Initialiser*, der als *Activator* der Komponente gilt. Dieser Initialiser definiert, welche Eigenschaften die Komponente zur Verfügung stellt und welche Eigenschaften anderer Komponenten benötigt werden. Wird eine Komponente geladen, kann im Initialiser zudem der Ladeprozess beeinflusst werden, in dem gezielt Sub-Komponenten geladen werden. Welche Komponenten das sind, hängt von der Komponente ab, die wiederum die Information statisch beinhalten oder durch Konfigurationsdateien erhalten kann. Nachdem eine Komponente geladen ist, muss diese gestartet werden. Hierfür werden im Initialiser benötigte Eigenschaften anderer Komponenten gebunden und lokale Handler an Ereignissen anderer Komponenten übergeben werden. Anschließend startet der Initialiser eigenständig die Komponente, in dem zunächst ein *Context* erzeugt wird, auf Grundlage dessen die Hauptklasse der Komponente instanziiert wird. Zu möglichen Typen von Handlern, die registriert werden können, gehören View Updates oder Anfragen zur Serialisierung des Zustands der Komponente.

Komponenten erzeugen durch das verkettete Starten in den Initialisern eine baumartige Struktur. Zu Beginn wird der Initialiser geladen, dessen Klassenname in einer Konfigurations-Konstanten *DEFAULT INITIALISER* steht. Üblicherweise ist das der *BasicStragetyInitialiser*, der selbst unter anderem den *ReplicationInitialiser*, den *GroupcomInitialiser* oder den *JoinInitialiser* startet. Der ReplicationInitialiser läd abhängig von der gewählten Konfiguration die passive oder die aktive Replikation. Der GroupcomInitialiser startet die Gruppenkommunikation auf Basis von jGroups und der JoinInitialiser startet das JoinProtocol, welches die Knoten in Virtual Nodes und deren Aktivitäten im Cluster verwaltet.

2.5. Zusammenfassung

Das Grundlagenkapitel befasst sich mit Themenfeldern, die im Folgenden der Ausarbeitung verwendet werden. Zunächst steht ein *verteiltes System* im Mittelpunkt, welches aus vielen Rechnern besteht und eine Anwendung gemeinsam ausführen soll. Damit serverseitig Rechner den selben Anwendungszustand bereit stellen und dadurch das Gesamtsystem konsistent ist, muss eine Synchronisierung erfolgen. Zwischen den beteiligten Knoten erfolgt zudem Kommunikation, die in Client-Server- oder Cluster-interne Kommunikation unterschieden werden kann.

Das Thema *Fehlertoleranz* aus dem Bereich der verteilten Systeme wurde definiert und die dafür notwendigen Techniken Skalierung und Redundanz vorgestellt. Fehlertoleranz wird in dieser Arbeit in Zuverlässigkeit und Verfügbarkeit unterteilt. Skalierung sorgt für die Verteilung der serverseitigen Anwendung auf mehrere in einem Cluster verfügbaren Knoten. Dadurch wird die Verfügbarkeit gewährleistet, da das System entsprechend den Anforderungen der Anwendung wachsen kann. Redundanz erzeugt primär Zuverlässigkeit, in dem parallele Ausführungen getätigt und die Ergebnisse miteinander verglichen werden. Im Bezug auf Ausfälle aufgrund von Hard- oder Software-Fehlern erhöht die Redundanz auch die Verfügbarkeit. Redundanz

birgt jedoch auch Risiken von Inkonsistenzen durch Nebenläufigkeiten und den Zugriff auf gemeinsame Daten. Entsprechend wurden Möglichkeiten zum Umgang mit Konflikten und zum Umgang mit Konsistenz vorgestellt. Bei Konflikten kann eine optimistische oder pessimistische Zugriffskontrolle stattfinden, die durch Locking oder Versionierung realisiert wird. Im Kontext von Konsistenz wurden das CAP-Theorem und das PACELC-Modell vorgestellt.

Neben verteilten Systemen und Fehlertoleranz wurden Grundlagen zu *Datenbanksystemen* erläutert. Sie stellen eine spezielle Form von verteilten Systemen dar und beschäftigen sich daher auch mit Skalierung, Redundanz, Konsistenz und Konflikten. Datenbanksysteme können anhand ihrer Speichertypen kategorisiert werden. Neben relationalen Datenbanksystemen wurden einige Speichertypen der NoSQL Datenbanksystemen vorgestellt. Besonders relationale Datenbanksysteme verwenden zur Wahrung einer strengen Konsistenz das ACID-Modell. Diesem Modell steht das BASE-Modell mit hohen Skalierbarkeitsanforderungen und Eventual Consistency gegenüber.

Zum Abschluss des Grundlagenkapitels wird ein Framework vorgestellt, auf Basis dessen eine Anwendung für ein verteiltes System implementiert werden kann. Das Virtual Nodes Framework soll fehlertolerante Anwendungen bspw. durch die Verwendung von Redundanz ermöglichen.

3. Konzepte der Fehlertoleranz in Datenbanksystemen

Nachdem im vorherigen Grundlagenkapitel Fehlertoleranz und verteilte Systeme vorgestellt wurden, beschäftigt sich das vorliegende Kapitel mit theoretischen Konzepten, mit denen Fehlertoleranz in Datenbanksystemen realisiert werden kann. Die Konzepte gliedern sich in Replizierung, Konfliktmanagement, Konsistenz, Partitionierung und Dauerhaftigkeit. Jedes dieser Themen enthält Kriterien mit einer Menge an möglichen Auswahlen, was sich am Ende jedes Themas tabellarisch anreiht. Die Recherche der Konzepte, Kriterien und ihre Auswahlmöglichkeiten im Hinblick auf Fehlertoleranz in Datenbanksystemen sind Ergebnisse dieser Arbeit. In Kapitel 4 werden elf Datenbanksysteme praktisch anhand der im vorliegenden Kapitel beschriebenen theoretischen Konzepten miteinander verglichen.

3.1. Replizierung

In Kapitel 2.2.2 wurde die Verwendung von Redundanz eingeführt. Um Redundanz zu erreichen, muss eine Replizierung der Daten eines Datenbanksystems erfolgen. Die Kommunikation basiert dabei auf in Kapitel 2.1.2 vorgestellten Grundlagen und wird hier im Bezug auf Datenbanksysteme genauer analysiert. Die Replizierung wird im Folgenden auf Knoten innerhalb eines Clusters bezogen. Eine Cluster-Replizierung findet in einigen Datenbanksystemen zusätzlich Anwendung und verhält sich auf einem übergeordneten Abstraktionslevel ähnlich wie die Cluster-interne Replizierung einzelner Knoten. Replizierung ganzer Cluster wird verwendet, um Daten geografisch näher an Nutzer zu bringen oder um vor regionalen Katastrophen geschützt zu sein.

Die Replizierung wird im Folgenden in Architektur und Aktualisierung unterteilt. Unter Architektur ist die Anordnung von Knoten oder das Verhalten im Fall eines Knotenausfalls des Gesamtsystems zu verstehen. Als zweiter Punkt Aktualisierung der Replizierung wird die Kommunikation der Knoten untereinander verstanden [31, 30, 32].

3.1.1. Architekturen

Die Architektur eines Datenbanksystems kann zunächst allgemein in *Single-Master* oder *Multi-Master* eingeteilt werden. Single-Master bedeutet, dass innerhalb des verteilten Systems ein einziger Knoten als primärer, für die Replizierung ausschlaggebender Knoten gilt. Überwiegend sind nur auf diesem Knoten Schreibzugriffe zugelassen, die gemäß dem erläuterten Leader/Follower- oder auch Primärkopie-Verfahren an die anderen Knoten übertragen werden. Multi-Master ent-

spricht dem Verfahren der aktiven Replikation, definiert jedoch allgemeiner die Existenz mehrerer Master-Knoten innerhalb des Systems, auf die parallel auf den selben Datensatz schreibend zugegriffen werden darf.

Bei einer Single-Master Architektur ist im Fall des Ausfalls des Masters für ein fehlertolerantes System die Reaktion des Systems interessant. In Kapitel 2.2.2 wurden Failover-Strategien im Rahmen von Redundanz in verteilten Systemen vorgestellt, die von *Cold-Standby* über *Warm-Standby* zu *Hot-Standby* gehen und auf Datenbanksysteme übertragbar sind. Dabei sind Kriterien wie Aktualität des Replikats und Verluste nach einem Ausfall des Masters interessant. Zusätzlich zur Failover-Strategie kann die Wahl des Masters einer Architektur kategorisiert werden. Im Fall eines Ausfalls kann manuell z.B. durch Eingriffe eines Adminstrators, oder automatisch vom System durch bspw. Abstimmungsverfahren der übrigen Knoten ein neuer Knoten als Master bestimmt werden. Denkbar ist auch eine statische Konfiguration der verfügbaren Knoten mit bspw. einer definierten Reihenfolge.

Eine weitere Unterscheidung bei der Replizierung von Datenbanksystemen ist die Granularität der Verteilung. Eine *vollständige Replizierung* findet statt, wenn jeder Knoten die kompletten Daten des Gesamtsystems vorliegen hat. Dem gegenüber steht die *partielle Replizierung*, bei der ein Knoten nur einen Teil der gesamten Daten im System vorliegen hat. Die partielle Replizierung hängt mit der Partitionierung der Daten zusammen, weshalb im späteren Kapitel zu Partitionierung darauf erneut Bezug genommen wird. Allgemein findet bei Single-Master Systemen meist eine vollständige Replizierung und bei partitionierten, Multi-Master Systemen eine partielle Replizierung statt.

3.1.2. Aktualisierung

Kapitel 2.1.2 führt Kommunikationsformen in verteilten Systemen, wie asynchrone/synchrone oder Push/Pull-basierte Nachrichtenübertragung ein. Bei Datenbanksystemen kann die Aktualisierung der Knoten für die Replizierung ebenso *synchron* oder *asynchron* erfolgen. Bei synchroner Aktualisierung wird während der Verarbeitung einer Anfrage an das Datenbanksystem die Replizierung sofort an alle Knoten ausgeführt, sodass die Anfrage erst nach erfolgter Replizierung beendet wird. Bei asynchroner Aktualisierung findet die Replizierung nicht während sondern unabhängig von der Anfrage statt. Dadurch kann die Anfrage schneller beantwortet werden, auf Kosten einer Bestätigung und Absicherung durch die Replizierung. Knoten können Informationen über *Push* an andere übertragen, oder aber replizierte Knoten holen über *Poll* die Informationen selbst ab. Die Wahl der Informationsbeschaffung hängt eng mit der Synchronität der Aktualisierung zusammen. Synchrone Aktualisierung wird über Push realisiert, eine Mischform kann als *Poll&Hold* realisiert werden, bei der asynchron aktualisiert wird, jedoch die Verbindung anschließend bestehen bleibt und darüber synchron aktualisiert wird.

Der Inhalt einer Aktualisierungsnachricht der Replizierung kann die Client-Anfrage an das System oder dessen ausgeführtes Ergebnis enthalten, so dass also *Operationen* oder die *neuen Daten* übertragen werden. Die im Kapitel 2.1.2 vorgestellte dritte Option, eine Benachrichtigung über das Vorhandensein neuer Daten zu versenden, findet in den betrachteten Datenbanksystemen keine Verwendung. Zusätzlich kann die Granularität der Nachrichten für die Aktualisierung definiert werden. Eine Aktualisierungsnachricht kann z.B. nach einer definierbaren Anzahl an

abgeschlossenen Operationen, nach einer Anzahl an Transaktion oder nach einem definierbaren zeitlichen Intervall erfolgen. Bspw. kann nach jeder (Anzahl = 1) Transaktion oder Operation eine Aktualisierungsnachricht erzeugt werden.

3.1.3. Zusammenfassung

Die vorgestellten Kriterien und ihre möglichen Werte werden in Tabelle 3.1 zusammengefasst. Die aufgelisteten Kriterien lassen sich auf vorhandene Datenbanksysteme anwenden und auf einen oder teilweise mehrere Werte aus dem Wertebereich zuordnen.

Kategorie		Wertebereich
	Cluster-Replizierung	möglich, nicht möglich
Architektur		
	Allgemein	Single-Master, Multi-Master
	Master-Failover	Cold-Standby, Warm-Standby, Hot-Standby
	Master-Wahl	automatisch, manuell, konfiguriert
	Verteilung Granularität	partielle Replizierung, vollständige Replizierung
Aktualisierung		
	Synchronität	synchron, asynchron
	Informationsbeschaffung	Pull, Push, Poll&Hold
	Informationsaustausch	Operationen, Neue Daten
	Nachrichten Granularität	je Operation, je Transaktion

Tabelle 3.1.: Zusammenfassung: Kriterien und Wertebereiche der Replizierung

3.2. Konfliktmanagement

Wie im Grundlagenkapitel 2 erläutert sind bei Redundanzen Konflikte möglich, die unterschiedlich vom Datenbanksystem gehandhabt werden können. Wie in Kapitel 2.2.2 eingeführt, existieren zwei grundlegende Strategien zur Behandlung von Konflikten: Locking und Versionierung. Locking kann z.B. mit Hilfe von Transaktionen erfolgen, die auch verteilt möglich sind. Dafür wird ein Protokoll wie das 2PC benötigt. In replizierten Datenbanksystemen mit einem einzigen Master-Knoten sind Transaktionen und Locking meist nur auf dem Master- und nicht auf den Slave-Knoten möglich. Bei Multi-Master Systemen mit Locking sind jedoch verteilte Transaktionen notwendig.

Bei Versionierung hingegen sind Konflikte zugelassen, müssen jedoch entsprechend erkannt und gelöst werden. In Datenbanksystemen wird die Versionskennung wie in Kapitel 2.3.1 erläutert durch Timestamps, Sequenznummern oder Vektor Uhren realisiert. Die Konflikterkennung findet anhand dieser entweder synchron bei einem Zugriff auf einen Datensatz oder durch einen asynchron laufenden Prozess im Hintergrund des Datenbanksystems statt. Bei Überprüfung während des Lese- oder Schreibzugriffs werden alle replizierten Knoten abgefragt und die Daten miteinander verglichen. Ein asynchroner Prozess kann bspw. nach einem zeitlichen

Intervall alle zuletzt geänderten Datensätze abfragen und vergleichen. Auf der einen Seite bietet diese Variante während des Zugriffs eine höhere Konsistenz, auf der anderen Seite ist jedoch im Bezug auf die Verarbeitungszeit der Anfrage ein asynchroner Job zu bevorzugen. Ist ein Konflikt erkannt worden, kann dieser vom Datenbanksystem eigenständig gelöst oder an die Anwendung zur Konfliktlösung weitergereicht werden. Bei asynchronen Prozessen werden im Fall der Konfliktlösung auf Anwendungsebene die Datensätze entsprechend markiert. Im synchronen Fall wird meist der Konflikt direkt an die Anwendung weitergereicht. Das Verfahren der Konfliktlösung ist allerdings üblicherweise konfigurierbar.

Ein spezielles Verfahren bei synchroner Konflikterkennung ist das Abstimmungs- oder *Quoren*-Verfahren. Hierbei werden alle replizierten Knoten (N Stück) bei einem Zugriff auf einen Datensatz angefragt, der anfragende Client bekommt jedoch bereits nach einer einstellbaren Anzahl an geantworteten Knoten Rückmeldung. Die Einstellung ist üblicherweise für Schreib- und Lesezugriffe (W, R) separat wählbar, sodass konfigurationsabhängig eine schwache (z.B. N=3, W=R=1) bis eine sehr starke Konsistenz (z.B. N=3, W=R=N) unter Abwägung der Notwendigkeit und Verarbeitungszeit erreicht werden kann. Müssen nicht alle replizierten Knoten antworten, wird zudem ein Ausfall einzelner Knoten toleriert.

Tabelle 3.2 stellt eine tabellarische Übersicht der Kriterien und ihrer Möglichkeiten zum Thema Konfliktmanagement in Datenbanksystemen dar.

Kategorie		Wertebereich
	Strategie	Locking, Versionierung
	Umsetzung	Voting/Quoren, 2PC
	Transaktionen	auf Master, verteilt, keine
Versionierung		
	Versionskennung	Timestamp, Sequenznummern, Vektor Uhren
	Konflikt Erkennung	asynchroner Job, Lese-/Schreibzugriff
	Konflitk Lösung	Datenbanksystem, Anwendung

Tabelle 3.2.: Zusammenfassung: Kriterien und Wertebereiche des Konfliktmanagements

3.3. Konsistenz

Das Grundlagenkapitel 2.2.2 und 2.3.1 erläutern Konsistenz, beschreiben verschiedene Konsistenzmodelle und übergeordnet Modelle wie ACID oder BASE. Im Bezug auf Datenbanksysteme muss die Konsistenz aus zusätzlichen Blickwinkeln betrachtet werden, welche abhängig von der gewählten Architektur der Replizierung sind. Neben der datenzentrierten und der clientzentrierten Sichtweise können die Knoten unterschiedlich betrachtet werden. So ist die Konsistenz in dem Knoten, zu dem ein Client aktuell verbunden ist anders als bei anderweitigen replizierten Knoten im Cluster. Für den aktuellen Knoten kann ein passendes datenzentriertes und ein clientzentriertes Konsistenzmodell gefunden werden. Für andere Knoten ist aufgrund der Entfernung nur eine clientzentrierte Sichtweise der Konsistenzmodelle sinnvoll.

Zudem ist die Architektur der Replizierung von Bedeutung, ob also ein einzelner Master und Slaves oder mehrere Master vorhanden sind. Oft wird in einer Single-Master Architektur im Master das ACID-Modell und in den Slaves sowohl auch bei Multi-Master Architekturen das BASE-Modell verwendet. Die Verwendung von ACID in einer Multi-Master Architektur erfordert einen hohen Organisationsaufwand und wird selten vollständig verwendet.

Tabelle 3.3 enthält Kriterien aus dem Bereich der Konsistenz, die auf Datenbanksysteme angewendet werden können. Neben den hier genannten kann der Schwerpunkt des betrachteten Datenbanksystems im CAP-Theorem oder im PECELC-Modell ausgedrückt werden.

Kategorie		Wertebereich
	Modellbezeichnung	ACID, BASE, Master ACID Slaves BASE
	CAP-Modell	CA, CP, AP
	PECELC-Modell	PAEL, PCEL, PAEC, PCEC
Aktiver Knoten		
	Datensicht	Strenge K., Sequentielle K.
	Clientsicht	Strenge K., Sequentielle K., Eventual Consistency
Sonstige Knoten		
	Clientsicht	Eventually Consistent, Read-Your-Writes, Monotonic Read

Tabelle 3.3.: Zusammenfassung: Kriterien und Wertebereiche der Konsistenz

3.4. Partitionierung

Um eine optimale horizontale Skalierung zu erreichen, ist eine Verteilung der Daten im System erforderlich, was zu einer Partitionierung der gespeicherten Daten führt. Die Komplexität dieser Aufteilung hängt vom gewählten Speichertyp des Datenbanksystems ab. Eng miteinander verknüpfte, komplexe Datensätze sind üblicherweise schwieriger zu partitionieren als voneinander unabhängige primitivere gespeicherte Informationen. Als Beispiel ist eine Graphen-basierte Datenbank sehr viel schwerer zu partitionieren als ein Key/Value-basiertes Datenbanksystem mit voneinander unabhängigen Datensatz-Paaren. Im Folgenden werden Charakteristiken einer Partitionierung von Datenbanksystemen wie die Dynamik, die verwendete Strategie und dem Zugriff eines Clients erläutert.

Unter Dynamik ist im Zusammenhang mit Partitionierung zu verstehen, ob sich die Daten automatisch oder manuell auf verschiedenen vorhandenen Knoten verteilen. Die Entscheidung der Dynamik hängt maßgeblich von der verwalteten Datenmenge und der Häufigkeit einer Veränderung ab. Eine manuelle Partitionierung findet meist über Konfigurationen statt und ist daher sehr gut für Datenmengen geeignet, die stabil sind. Automatische Skalierung ist besonders für stark schwankende Datenmengen interessant, da vom Datenbanksystem eigenständig bei Bedarf skaliert werden kann. So können automatisch beispielsweise auch neue Knoten hinzu oder abgeschaltet werden, um die Last mit optimalem Resourcenverbrauch zu bewältigen.

Die Aufteilung der Daten in Partitionen oder in sogenannte *Shards* kann auf verschiedene Weise erfolgen. Die beiden üblichen Varianten sind *Regions/Ranges* oder *Hashing*. Gemeinsam haben beide, dass alle Datensätze zunächst ein Feld haben müssen, z.B. einen eindeutigen Primärschlüssel, um den Datensatz zu identifizieren. Bei Regions werden jedem Knoten Wertebereiche der vergebenen Primärschlüssel automatisch oder manuell zugeteilt. Beim Zugriff auf einen Datensatz über den Primärschlüssel kann der zuständige Knoten anhand der Wertebereiche ermittelt werden. Damit die Einteilung in Bereiche funktioniert, müssen die Datensätze entsprechend des gewählten Feldes sortiert vorliegen. Ein Knoten enthält ab einem bestimmten Startwert alle Elemente bis zu einem Endwert. Für eine gleichmäßige Aufteilung der Daten auf zur Verfügung stehende Knoten ist eine entsprechende Einteilung der Start- und Endwerte notwendig. Bei Veränderungen der Werte kann leicht ein Ungleichgewicht entstehen, sodass eine gelegentliche Neuverteilung der Bereiche erforderlich wird. Für eine gleichmäßige Streuung der Daten ist die Strategie des Hashings zu bevorzugen, da die Einteilung in Bereiche weg fällt und statt dessen über eine Hash-Funktion Datensätze auf vorhandene Knoten zugeteilt werden. Eine Hash-Funktion kann beispielsweise eine Moduloberechnung eines numerischen Primärschlüssels und der Anzahl verfügbarer Knoten sein, um als Ergebnis die Nummer des zuständigen Knotens zu erhalten. Benachbarte Datensätze, die bei Regions im selben Wertebereich liegen würden, werden durch die Hash-Funktion verteilt, was eine gleichmäßigere Verteilung der Daten auf die Knoten bedeutet. Durch Hinzufügen oder Wegfall eines Knoten wird jedoch im Fall von Regions oder Hashing die Zuteilung ungleichmäßig oder sogar durch Veränderung der Hash-Funktion ungültig, weshalb alle Datensätze neu verteilt werden müssten. Daher setzen vorhandene Datenbanksysteme statt "einfachem" Hashing das sogenannte *Consistent Hashing* zur Partitionierung ein. Schematisch werden zunächst die Primärschlüssel auf einen festgelegten unveränderlichen Wertebereich über eine Hash-Funktion verteilt. Auf dem Wertebereich sind verfügbare Knoten verteilt.

Kategorie	Wertebereich
Dynamik	Konfiguration, automatisch
Strategie	Regions, Consistent Hashing
Zugangsknoten	zentraler Knoten, beliebiger Knoten, Direktzugriff
Zuordnung	clientseitig, serverproxy, clientproxy, routing
Rückskalierung	nicht möglich, automatisch, manuell

Tabelle 3.4.: Zusammenfassung: Kriterien und Wertebereiche der Partitionierung

Durch die Verteilung der Datensätze auf verschiedene Knoten entsteht die Herausforderung, dass kein einfacher Zugriff eines Clients auf einen Knoten im Cluster alle Zugriffe implizit bearbeiten kann. Ein Client hat die wesentlichen Möglichkeiten, über einen Direktzugriff den richtigen Knoten anzufragen. Andernfalls kann ein zentraler Knoten als Ansprechpartner für Clients zur Verfügung stehen, oder der Client wählt beliebig z.B. zufällig einen Knoten für seine Anfrage aus, der ggfs. die Anfrage weiterleitet. Im Fall einer Single-Master Architektur mit *vollständiger Replizierung* und daher ohne Verteilung der Datensätze wird der Direktzugriff verwendet, da jeder Knoten die vollständigen Daten vorliegen hat. Im Fall einer Verteilung der Daten ist ein Direktzugriff nur möglich, wenn der Client Informationen über die Partitionierung hat, anhand derer er eigenständig die Anfrage eines Datensatzes einem Knoten zuordnet. Steht diese Information nur Cluster-intern zur Verfügung, kann ein serverseitiger Proxy auf einem separaten zentralen Knoten oder zusätzlich auf jedem Knoten Zugriffe entgegen nehmen und stellvertretend

für den Client im Cluster verarbeiten lassen. Der Proxy kann hierbei die Verantwortung während der gesamten Verarbeitungsdauer übernehmen oder lediglich als Routingmechanismus dienen, sodass die zuständigen Knoten den Client für Resultate direkt kontaktieren. Ein Spezialfall ist ein clientseitiger Proxy bspw. in einer Middleware der Komponente, die eine Anwendung mit dem Datenbanksystem verbindet. Partitionierung in Kombination mit Replizierung führt zu partieller Replikation. Da ein Knoten nur seine Partition und damit nicht die vollständigen Daten vorliegen hat, kann ein Replikat des Knotens auch nur diesen Teil der Daten replizieren.

Tabelle 3.4 stellt die Kategorien aus dem Bereich der Partitionierung übersichtlich dar, anhand derer Datenbanksysteme mit Unterstützung für Partitionierung eingeordnet werden können. Ein zusätzliches Kriterium ist die Möglichkeit einer Rückskalierung des Clusters, sodass eine Verkleinerung des Clusters bei Abnahme an Zugriffen oder Daten erfolgen kann.

3.5. Dauerhaftigkeit

Je nach Anwendungsgebiet eines Datenbanksystems ist die Dauerhaftigkeit, also die garantierte Speicherung auch im Fall eines Fehlers im System, unterschiedlich wichtig. Entsprechend bieten Datenbanksysteme verschiedene Garantien für die Persistierung von ändernden Zugriffen auf Datensätze. Die Garantie reicht von schwach bis streng, wobei unter schwach die Bezeichnung *Eventual persistent* fällt. Eventual persistent bedeutet, dass ein Schreibzugriff nicht sofort sondern asynchron zu einem späteren Zeitpunkt persistiert wird. Fällt zwischenzeitlich das System aus, können Datenverluste auftreten. Der Gegensatz dazu ist eine strenge Dauerhaftigkeit, die im ACID-Modell Anwendung findet. Um auch im schlimmsten Fall eines Ausfalls keine Verluste zu haben wird sogenanntes *Journaling* oder *Write-Ahead-Logging* eingesetzt. Schreibzugriffe werden hier vor oder während der Ausführung zuverlässig in eine Logdatei geschrieben. Logdateien haben die Eigenschaft, immer am Ende erweitert zu werden, weshalb das Protokollieren des Schreibzugriffs schnell erfolgen kann. Gerade im Bereich von relationalen Datenbanksystemen kann ein Schreibzugriff aufgrund der intern auf Lesezugriffe optimierte Datenstruktur erheblichen Aufwand bedeuten. Bei einfacheren Datentypen kann eine strenge Dauerhaftigkeit auch durch *In-Place-Updates* unmittelbar in der Datenstruktur erfolgen, ohne Protokolldateien. Gerade bei replizierten Knoten kann auch über ein sog. *Quorum* definiert werden, wieviele Knoten einen Schreibzugriff erfolgreich dauerhaft persistiert haben müssen, bevor ein Client-Zugriff als erfolgreich gilt. Allgemein ist eine strenge Dauerhaftigkeit nur in Kombination mit einem dauerhaften Medium wie einer Festplatte (Disk) sinnvoll. Teilweise setzen Datenbanksysteme aufgrund von Latenzzeiten nur auf RAM-basierte Lösungen, wenn z.B. eine Persistenz anwendungsbedingt nicht notwendig ist oder anderweitig erfolgt. Eine interessante und performante Kombination ist die Verwendung von Journaling mit einer asynchronen Persistierung. Tabelle 3.5 stellt die drei Kategorien der Dauerhaftigkeit und mögliche Werte übersichtlich dar.

Kategorie	Wertebereich
Speichermedium	Disk, RAM
Journaling	ja, nein
Garantie	Streng, Quorum, eventual persistent

Tabelle 3.5.: Zusammenfassung: Kriterien und Wertebereiche der Dauerhaftigkeit

3.6. Zusammenfassung

Das dritte Kapitel der Ausarbeitung stellt Konzepte zur Realisierung von fehlertoleranten Datenbanksystemen vor. Dabei werden Grundlagen und Konzepte aus dem Grundlagenkapitel aufgegriffen und in Kombination im Hinblick auf ein Datenbanksystem verarbeitet. Die vorgestellten Konzepte Replizierung, Konfliktmanagement, Konsistenz, Partitionierung und Dauerhaftigkeit können mit Hilfe der jeweils vorhandenen tabellarischen Übersicht der Unterscheidungskriterien auf ein Datenbanksystem angewendet werden um dieses im Hinblick auf Fehlertoleranz zu analysieren. Eine Analyse elf ausgewählter Datenbanksysteme schließt sich im nachfolgenden Kapitel an.

4. Klassifizierung von fehlertoleranten Datenbanksystemen

Datenbanksysteme als erweiterte, spezielle verteilte Systeme wurden im vorausgehenden Kapitel 2 im Hinblick auf verschiedene Speichertypen beschrieben. Während in Kapitel 2 Grundlagen zum Umgang mit Fehlertoleranz in verteilten und in Datenbanksystemen erläutert wurden, definiert Kapitel 3 Konzepte der Fehlertoleranz in Datenbanksystemen. Diese Konzepte dienen der Analyse vorhandener Datenbanksysteme. Teil dieser Arbeit ist eine entsprechende Analyse von elf betrachteten Datenbanksystemen aus den verschiedenen Kategorien aus Kapitel 2.3.2. Das Ergebnis ist tabellarisch im Anhang A enthalten und wird im aktuellen Kapitel ausgewertet. Die folgende praktisch orientierte Analyse beschreibt eine Klassifizierung von Produkten.

Die tabellarische Analyse betrachtet jedes der elf Datenbanksysteme separat. Jedoch ist eine Ähnlichkeit durch die Verwendung ähnlicher Ansätze zwischen den Datenbanksystemen erkennbar. Die Datenbanksysteme lassen sich daher in fünf Gruppen einteilen. Zunächst werden im Folgenden diese Gruppen vorgestellt. Anschließend reihen sich detailiertere Ausführungen zu jeder Gruppe aneinander, wobei die betrachteten Datenbanksysteme namentlich zugeordnet werden. In jeder Gruppe werden die Konzepte Replizierung, Skalierung und Dauerhaftigkeit betrachtet, wobei Konfliktmanagement und Konsistenz der Replizierung untergeordnet ist.

4.1. Gruppierung der Datenbanksysteme

Die fünf Gruppen der Datenbanksysteme beschreiben die markantesten Merkmale der eingeordneten Datenbanksysteme. Ein Merkmal ist die Eigenschaft der Architektur der Replizierung, also ob Single-Master oder Multi-Master verwendet wird. Ein weiteres Merkmal ist die Möglichkeit der Skalierung, die vom Datenbanksystem bereitgestellt wird. Drei der fünf Gruppen lassen sich anhand dieser beiden Merkmale beschreiben. Die zwei übrigen Gruppen beschreiben spezielle Anwendungsfälle. Im Folgenden werden die Gruppen vorgestellt und kurz beschrieben. Eine detailliertere Beschreibung befindet sich in den nachfolgenden Kapiteln.

SML: Single-Master und Lese-Skalierung beinhaltet Datenbanksysteme, deren primäres Ziel auf strenger Konsistenz liegt. Um Ausfallsicher zu sein werden Daten eines Masters auf Slaves repliziert. Überwiegen Lese- gegenüber Schreibzugriffen können die Slaves zur Lese-Skalierung verwendet werden.

SMP: Single-Master und Partitionierung verwendet wie Gruppe SML Single-Master zur Wahrung der Konsistenz, bietet jedoch die Möglichkeit einer Partitionierung um das Datenbanksystem horizontal skalieren zu können.

MMC: Multi-Master und Consistent Hashing beinhaltet Datenbanksysteme mit hohen Anforderungen an die Skalierbarkeit des Datenvolumen bis hin zu Tera- oder Petabyte. Consistent Hashing wird als Verfahren zur Partitionierung und Replizierung verwendet, wobei alle Replikate Master-Knoten sein können.

MMO: Multi-Master Offline-By-Default entspricht einem Spezialfall, in dem alle Knoten gleichberechtigt sind, jedoch keine ständige Verbindung zueinander haben. Die Regelmäßigkeit einer Kommunikation der Knoten muss nicht gewährleistet werden, was bei anderen Gruppen notwendig ist.

RCL: Relationale Cluster ist der zweite Spezialfall, wobei in dieser Gruppe Datenbanksysteme eingeordnet sind, die dem relationalen Modell folgen und dennoch horizontal skalierbar sind.

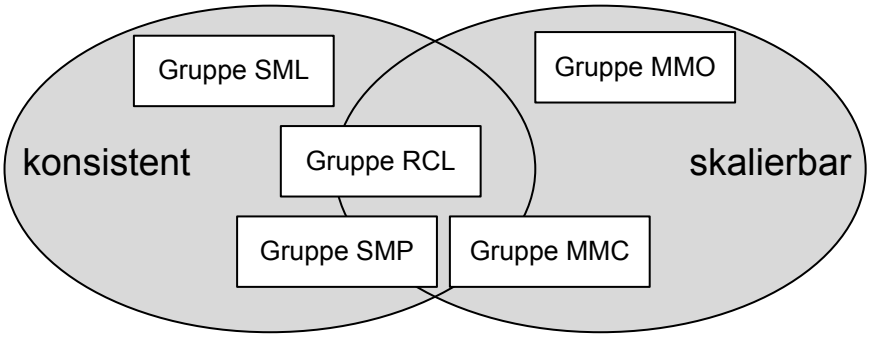

Abbildung 4.1.: Einordnung der Gruppen zwischen Konsistenz und Skalierbarkeit

Die fünf vorgestellten Gruppen können in den Bereichen zwischen konsistent und skalierbar eingeordnet werden. Abbildung 4.1 stellt die Positionen der Gruppen graphisch dar. In den nachfolgenden Kapiteln 4.2 bis 4.6 werden die fünf Gruppen detailliert mit Bezug auf die betrachteten Produkte beschrieben.

4.2. SML: Single-Master und Lese-Skalierung

In die erste Gruppe von Datenbanksystemen lassen sich relationale Systeme wie *MySQL*[1] [23] oder *PostgreSQL*[2] [28] einteilen. Sie haben aufgrund des verwendeten ACID-Modells eine hohe Garantie der Konsistenz von gespeicherten Daten. Eine Partitionierung ist nicht vorgesehen, weshalb diese Systeme bei kleineren Datenvolumen bis hin zu einigen Gigabyte verwendet werden, die von einem einzelnen Knoten verwaltet werden. Sie bieten durch Transaktionen und die Verwendung von SQL eine sehr robuste und zuverlässige Datenhaltung und werden z.B. in webbasierten Anwendungen wie mittelständische e-commerce-Systeme verwendet. Neben den genannten relationalen Datenbanksystemen lassen sich in diese erste Gruppe auch Graphen-basierte Datenbanken wie *Neo4J*[3] [21] zuordnen. Durch die interne Speicherstruktur

[1] http://www.mysql.de
[2] http://www.postgresql.org/
[3] http://www.neo4j.org/

der eng miteinander verknüpften Daten ist eine Partitionierung noch schwieriger möglich als bei relationalen Modellen, weshalb auf horizontale Skalierung mit Partitionierung auf Ebene des Datenbanksystems verzichtet wird.

Replizierung

In dieser Gruppe wird also als Architektur der Replizierung aus Kapitel 3.1 das Prinzip des Single-Master angewendet. Um gegen Ausfälle des Master-Knotens des Datenbanksystems geschützt zu sein, werden Backups mit verschiedenen Failover-Strategien unterstützt, sodass eine vollständige Replizierung des Datenbanksystems mit z.B. Hot Standby zur Absicherung eines Ausfalls oder als Backup zur Datensicherung genutzt werden kann. Die Aktualisierung der Replizierung erfolgt asynchron, sodass die Slave-Knoten mit Pull für Cold- und Warm-Standby oder Pull&Hold für Hot-Standby aktuelle Daten beliebig oft abrufen können. Die Übertragung verläuft über eine Log-Datei, die vom Master geschrieben und von den Slaves ausgelesen wird. Der Master kann in diese Datei entweder die Anfrage des Clients an ihn oder das berechnete Ergebnis der Anfrage abspeichern, sodass als Inhalt entweder die Operation oder die neuen Daten repliziert werden. Um garantiert Hot Standby zu unterstützen kann PostgreSQL so konfiguriert werden, dass Schreibzugriffe beim Master nur möglich sind, wenn mindestens ein Slave per Poll&Hold verbunden ist.

Fällt ein Master-Knoten aus, existiert bei den betrachteten Datenbanksystemen kein automatischer Wechsel eines Slaves zum neuen Master. Dieser Vorgang muss also, sofern dieser Wechsel gewünscht wird, manuell oder von einer externen Anwendung getätigt werden. Die Konfiguration von Master und Slave aber auch der Anwendung im Bezug auf die Verbindung zum Datenbanksystem erfolgt in diesen Systemen statisch. Ein automatischer Wechsel ist dadurch üblicherweise nicht möglich, da hierfür eine zusätzliche Organisation notwendig ist, welche von den Datenbanksystemen dieser Gruppe nicht bereitgestellt wird. Ein derartiger Mechanismus kann auf Anwendungsebene nachgerüstet werden.

Skalierung

Abhängig vom Anwendungsfall werden die replizierten Knoten nicht nur als Backups und zu Failover-Zwecken genutzt, sondern auch zur Lese-Skalierung. Da ACID und damit strenge Konsistenz nur im Master-Knoten gewährleistet wird, sind die Slaves jedoch *eventually consistent* und verfolgen vielmehr das BASE-Modell. Ist eine schwächere Anforderung der Konsistenz zu akzeptieren, ist es möglich, dass eine Anwendung Schreibzugriffe auf dem Master-Knoten und Lesezugriffe auf einem der Slaves ausführt und dadurch das Gesamtsystem skaliert. Da bei vielen wechselseitigen Schreib- und Lesezugriffen die Anwendung Probleme der Read-Your-Writes-Konsistenz bekommen kann, wird diese Lese-Skalierung meist nur eingesetzt, wenn Schreib- und Lesezugriffe unabhängig voneinander sind und Lesezugriffe häufiger auftreten als Schreibzugriffe. Treten hauptsächlich Schreibzugriffe auf, kann es zudem auftreten, dass die Slaves mit Einspielen des Replizierungs-Logs beschäftigt sind, sodass eine Lese-Skalierung aufgrund langer Latenzzeiten auf dem Master und auf den Slaves keinen Mehrwert in Bezug auf verkürzte Verarbeitungszeiten bedeutet.

Dauerhaftigkeit

Die vorgestellten Datenbanksysteme verfolgen im Master-Knoten das ACID-Modell und garantieren durch die Verwendung von Journaling eine strenge Dauerhaftigkeit. Die Persistierung findet auf einem dauerhaften Speichermedium wie einer Festplatte statt. Um Schreibzugriffe schneller abarbeiten zu können, können diese nicht sofort auf die Festplatte, sondern erst im RAM erfolgen, werden jedoch in einem Write-Ahead-Log für den Fall eines Absturzes protokolliert. MySQL bietet bei der Definition einer Tabelle zudem die Wahl einer sog. *Storage Engine* an. In Abhängigkeit der gewählten Storage Engine wird die Persistierung unterschiedlich getätigt oder kann im Ausnahmefall der Storage Engine MEMORY nur RAM-basiert erfolgen.

Zusammenfassung

Die in dieser Gruppe betrachteten Datenbanksysteme verwenden also Single-Master mit ACID im Master und BASE in den Slaves. Für eine Einordnung gemäß des CAP-Theorems liegen die Schwerpunkte auf Konsistenz und Verfügbarkeit (CA), im PACELC-Modell also PCEC was immer konsistent bedeutet. Im Fall einer auftretenden Netzwerkpartition bleibt das Datenbanksystem also konsistent und verfügbar sofern dies durch die Netzwerkpartition möglich ist. Das Datenbanksystem ist nicht tolerant gegenüber Netzwerkpartitionen, da z.B. beim Wegfall des Master-Knotens keine weiteren Schreibzugriffe getätigt werden können. Um eine strenge Konsistenz zu gewährleisten werden am Master die Datensätze gelockt und parallel laufende Anfragen blockiert. Im Hinblick darauf ist eine Lese-Skalierung mit Lesezugriffen auf einen Slave bei vielen Locks am Master sinnvoll.

4.3. SMP: Single-Master und Partitionierung

Die zweite Gruppe der Datenbanksysteme legt ebenfalls den Schwerpunkt auf Konsistenz. Allerdings bieten die hier betrachteten Datenbanksysteme die Möglichkeit einer Partitionierung. Die drei eingeordneten Datenbanksysteme sind aus verschiedenen Speichertypen aus dem Bereich der NoSQL Datenbanksysteme. *HBase*[4] [3, 17] ist Spalten-orientiert, *Redis*[5] ist [8] ein Key/Value-basiertes Datenbanksystem und *MongoDB*[6] [20] ist Dokumenten-orientiert. Alle drei Datenbanksysteme bieten aufgrund der nur leichten Verknüpfungen innerhalb der Daten die Möglichkeit, diese auf verschiedene Knoten zu verteilen. Um die Konsistenz zu wahren, ist jedoch für jede Partition bzw. Shard nur ein Knoten als Master-Knoten zuständig und für Backups oder Lese-Skalierung möglicherweise mehrere Slave-Knoten vorhanden. Während Redis und MongoDB auch für den Einsatz eines einzelnen Knotens entworfen und die Replizierung und Partitionierung eine Erweiterung darstellt, ist HBase auf ein Cluster mit mindestens fünf Knoten und Daten im Terabyte-Bereich ausgelegt. HBase basiert auf einer Veröffentlichung von Google, in der die intern verwendete Datenbank BigTable [7] beschrieben wird. HBase benötigt, analog zu BigTable, ein verteiltes Dateisystem, welches bei HBase HDFS[7] ist. MongoDB nutzt ebenso im

[4] http://hbase.apache.org/
[5] http://redis.io/
[6] http://www.mongodb.org/
[7] http://hadoop.apache.org/

Bereich der Partitionierung Ideen aus BigTable, was im Bezug auf die Organisation der einzelnen Knoten der Skalierung zu sehen ist, verwendet jedoch kein verteiltes Dateisystem.

Replizierung

Die Replizierung ist bei allen drei vorgestellten Datenbanksystemen asynchron, wodurch die Slaves eventually consistent sind. Bei HBase schreibt der Master für jeden angemeldeten Slave ein separates Logfile, in denen die Operationen gesammelt werden. Da HBase auf ein gemeinsames verteiltes Dateisystem setzt, findet die Replizierung nicht auf Knoten sondern implizit auf Cluster-Ebene statt. Überschreitet die Logfile eine gewisse Größe, wird sie vom Master an den Slave per Push übertragen. Bei MongoDB schreibt der Master-Knoten sämtliche Operationen in einen sog. Oplog, der von den Slaves an unterschiedlicher Stelle im Dokument fortlaufend gelesen wird, also über Poll. Bei Redis muss der Slave erst beim Master einen neuen Dump anfordern, der speziell für diesen Slave vom Master erzeugt und übertragen wird. Optional kann der Slave die Verbindung aufrecht erhalten um neue, anfallende Operationen direkt zu bekommen. Das Verfahren bei Redis ist also Poll&Hold auf Basis von neuen Daten per Dump komplett und nachfolgend je Operation inkrementell. Bei HBase und MongoDB werden Operationen übertragen. Aufgrund der Single-Master Architekturen der drei Datenbanksysteme sind wegen der Anforderung an die Konsistenz Schreibzugriffe nur auf dem Master möglich. Bei allen drei Datenbanksystemen wird über Locking sicher gestellt, dass keine Konflikte auftreten können. Bei MongoDB ist das Locking durch ein sog. Update-in-Place implizit vorhanden, wobei ein Datensatz atomar gelesen, verändert und zurückgeschrieben wird. Im Fall einer Netzwerkpartition kann es auftreten, dass zwei Master zur selben Zeit in zwei verschiedenen Subnetzen gewählt werden, die beim Lösen des Netzwerkfehlers möglicherweise in Konflikt zueinander stehen. MongoDB empfiehlt hiergegen eine ungerade Anzahl an Knoten, damit im Fall einer Zweiteilung des Netzwerks nur in einem Teil ein Master eindeutig gewählt wird und im anderen keine eindeutige Einigung entstehen kann. HBase bietet für den Fall eines Konflikts zusätzlich zum Locking eine Versionierung der Datensätze. Die Konfliktlösung verläuft über einen asynchronen Prozess, der anhand eines Timestamps die Versionskennung von Datensätzen prüft.

Die betrachteten Datenbanksysteme HBase, Redis und MongoDB sind alle dem BASE-Modell zugeordnet, obwohl HBase sehr weite Teile des ACID-Modells abdeckt. Im CAP-Theorem liegen die Schwerpunkte bei HBase und MongoDB auf CP, also konsistent und tolerant gegen Netzwerkpartitionen. Im Fall einer Netzwerkpartition wird zur Wahrung der Konsistenz die Verfügbarkeit des Datenbanksystems aufgegeben. Redis hingegen ist mit CA konsistent, verfügbar und ähnlich den Datenbanksystemen aus Gruppe SML nicht tolerant gegenüber Netzwerkpartitionen.

Skalierung

Neben einer strengen Konsistenz im Master-Knoten verfügen die Datenbanksysteme der zweiten Gruppe über die Möglichkeit einer Partitionierung. Bei Redis ist die Partitionierung nicht im Datenbanksystem vorhanden sondern Aufgabe der Anwendung, die aufgrund der einfachen Speicherstruktur von Key/Value-Paaren bspw. mit einem Hashing-Verfahren verteilen kann. Der Client greift dann direkt auf den richtigen Knoten im Cluster zu, da die Knoten untereinander nichts über die Partitionierung wissen. HBase und MongoDB verwenden einen Ansatz zur

Partitionierung, der fest im Datenbanksystem verankert ist. Dabei werden die Daten in Bereiche eingeteilt, weshalb eine Sortierung der Datensätze erforderlich ist (vgl. Kapitel 3.4). Die Einteilung der Daten in Bereiche sowie die Zuteilung der Bereiche auf vorhandene Knoten muss organisiert und gespeichert werden. Im Fall von HBase und MongoDB finden hier spezielle Knoten-Typen Anwendung, die parallel zu den Daten-Knoten, welche die tatsächliche Speicherung der Daten tätigen, arbeiten. Diese Organisations-Knoten können, um optimale Skalierbarkeit zu unterstützen, ebenso partitioniert werden. Bei HBase findet die Organisation in einer Tabelle namens META statt, welche wie jede andere Tabelle des Spalten-orientierten Datenbanksystems partitioniert und repliziert werden kann. Eine Tabelle ROOT, welche auf einem sog. NameNode gespeichert ist, dient als zentraler Einstiegspunkt der Partitionierung, in dem die ROOT-Tabelle auf die erste Stufe der META-Tabellen referenziert. Auf diesem NameNode, wird zudem ein Dienst ausgeführt, welcher die Daten-Knoten, sog. Region-Server, überwacht und im Fall eines Ausfalls einen Ersatz für den Knoten organisiert. Ein Client kopiert vor einer Anfrage an das Datenbanksystem zunächst die ROOT-Tabelle des zentralen NameNodes in einen lokalen Speicher, sowie notwendige META-Tabellen, um die Anfrage an das Datenbanksystem lokal auf die zuständigen Region-Server auflösen zu können. Die Region-Server werden dann vom Client direkt kontaktiert und gezielt abgefragt, als Einstiegspunkt dient dabei der zentrale NameNode. Bei MongoDB findet eine ähnliche Aufteilung statt, wobei die Organisations-Knoten sog. Config-Server und die Daten-Knoten sog. Shard-Server sind. Statt einem zentralen NameNode, der die Daten-Knoten überwacht, wird die Ausfallkontrolle einzelner Knoten durch sog. Heart-Beats gelöst, die als einfache Nachrichten regelmäßig zwischen sämtlichen Knoten übertragen werden. Damit Client-Anfragen nicht lokal sondern vom Datenbanksystem aufgelöst werden, hat MongoDB zusätzlich sog. Routing-Server, an welche ein Client seine Anfrage zentral schicken kann.

Um optimale Ausfallsicherheit der Datenbanksysteme zu gewährleisten, werden sämtliche Knoten nach der Single-Master Architektur repliziert. Gemeinsam mit einer Partitionierung ergeben sich tatsächlich mehrere Master-Knoten im Cluster (Multi-Master), allerdings ist für einen Datensatz nur ein einzelner Master zuständig. Die nachfolgende Gruppe mit Multi-Master Datenbanksystemen fasst diesen Punkt nochmals auf.

Dauerhaftigkeit

Die Dauerhaftigkeit der betrachteten Datenbanksysteme in dieser zweiten Gruppe wird verschieden betrachtet. Redis ist primär ein RAM-basiertes Datenbanksystem also ohne Persistenz. Optional kann allerdings nach einer definierbaren Anzahl an Schreiboperationen oder nach einem zeitlichen Invervall die Daten im RAM auf eine Festplatte persistiert werden. Um gegen Datenverluste im Fall eines Ausfalls zwischen einer Operation und dem Persistiervorgang geschützt zu sein, kann optional ein Write-Ahead-Log aktiviert werden, das sämtliche Schreiboperationen protokolliert. Redis zielt auf eine möglichst kurze Verarbeitungszeit von Anfragen im Master-Knoten, weshalb der Persistiervorgang bspw. in einem der Slaves getätigt werden kann, während der Master nur RAM-basiert arbeitet. Die Garantie an die Dauerhaftigkeit ist bei Redis also abhängig von der gewählten Konfiguration. Bei den beiden anderen Datenbanksystemen ist die Garantie streng. MongoDB und HBase protokollieren sämtliche Schreiboperationen in einem Write-Ahead-Log, wobei dieser ohnehin für die Replizierung notwendig ist. MongoDB speichert sämtliche Daten auf die lokale Festplatte, während HBase optional auch rein RAM-basiert arbeiten kann. Generell findet die Speicherung bei HBase im verteilten Dateisystem HDFS statt, sodass

die Fehlertoleranz von HBase stark von der Fehlertoleranz des Dateisystems abhängt. Durch die Verwendung eines gemeinsamen Dateisystems kann eine strenge Dauerhaftigkeit auch im Fall eines Knotenausfalls gewährleistet werden. HBase kann trotz des gemeinsamen Dateisystems als Shared-Nothing bezeichnet werden, da einzelne Knoten nur auf ihre Zuständigkeiten im Dateisystem zugreifen. Dateien wie die Write-Ahead-Logs werden bspw. gemeinsam genutzt.

Zusammenfassung

Die zweite Gruppe SMP beinhaltet drei NoSQL Datenbanksysteme unterschiedlichen Speichertyps. Gemeinsam haben sie, dass Partitionierung zur horizontalen Skalierung möglich ist. Die Verteilung der Daten im Cluster ist zwar dynamisch, muss jedoch organisiert und festgehalten werden. Dafür sind bei MongoDB und HBase spezielle Knoten und Tabellen vorhanden. Redis unterstützt die Partitionierung, bietet jedoch selbst keine Verteilung an, sodass die Organisation außerhalb des Datenbanksystems erfolgen muss. Trotz horizontaler Skalierbarkeit haben die betrachteten Datenbanksysteme hohe Anforderungen an die Konsistenz der Daten. Zur Replizierung wird daher die Single-Master Architektur verwendet, analog zur ersten Gruppe SML. Ebenso ähnlich ist die garantierte Dauerhaftigkeit, sofern eine Persistierung verwendet wird.

4.4. MMC: Multi-Master und Consistent Hashing

In der dritten Gruppe verwenden die betrachteten Datenbanksysteme gemäß Kapitel 3 Multi-Master und zur Partitionierung der Daten das Consistent Hashing Verfahren. Die drei Datenbanksysteme *Cassandra*[8] [11, 19, 4], *Riak*[9] [5] und *Couchbase*[10] [10, 9] verwenden mehr oder weniger Konzepte aus Amazons intern genutzer Datenbank Dynamo [13], weshalb besonders im Bereich der Replizierung und Partitionierung große Ähnlichkeiten vorhanden sind. Alle drei sind NoSQL Datenbanksysteme, jedoch ist Cassandra ähnlich wie HBase ein Spaltenorientiertes Datenbanksystem, Riak ist ein Key/Value-basiertes und Couchbase eine Mischform aus Key/Value-basiertem und Dokumenten-orientiertem Datenbanksystem. Couchbase ist ein Zusammenschluss der Datenbanksysteme Membase und CouchDB, weshalb der Speichertyp übergreifenend beide Formen unterstützt. Im Folgenden wird die Cluster-interne Replizierung betrachtet, allerdings können die vorgestellten Datenbanksysteme zusätzlich das komplette Cluster replizieren.

Replizierung und Skalierung

Die Replizierung und Partitionierung findet auf Basis von Consistent Hashing statt und wird daher auch hier gemeinsam betrachtet. Cassandra und Riak übertragen abhängig vom gewählten Quorum einer Schreibanfrage synchron oder asynchron an alle oder Teile der Replikate. Bei Cassandra werden die Operationen und bei Riak die neuen Daten übertragen. So kann eine

[8] http://cassandra.apache.org/
[9] http://basho.com/riak/
[10] http://www.couchbase.com/

vollständig asynchrone (W=1, N beliebig), eine vollständig synchrone (W=3, N=3) oder eine Mischform auftreten (z.B. W=2, N=3), wenn entsprechend keine, alle oder eine Teilmenge der vorhandenen Replikat-Knoten aktualisiert werden sollen. Couchbase bietet keine Möglichkeit einer Angabe und arbeitet immer mit einer asynchronen Replizierung (W=1, N beliebig), wobei die neuen Daten in einer sog. Replication Queue abgelegt und von einem asynchronen Prozess verteilt werden. Couchbase ist also als Master-Failover Warm Standby, die anderen beiden abhängig von der gewählten Konfiguration Warm oder sogar Hot Standby.

Bei Cassandra und Riak sind Schreibzugriffe auf einem beliebigen der Replikat-Knoten erlaubt, während Couchbase nur Schreibzugriffe auf dem ersten Knoten im Ring und optional Lesezugriffe auf den nachfolgenden Replikat-Knoten erlaubt. Bei Cassandra und Riak können also Konflikte auftreten, welche mit Versionierung gehandhabt werden. Bei Cassandra wird die Versionskennung über einen absoluten Timestamp, bei Riak über eine Vektor Uhr realisiert. Bei Couchbase treten Konflikte nur im Fall einer Netzwerkpartition auf, welche über eine fortlaufende Sequenznummer im Datensatz vom System anhand der Aktualität gelöst werden. Im Fall einer gleichen Sequenznummer entscheidet Couchbase anhand des Inhalts deterministisch. Cassandra löst entstandene Konflikte selbstständig bei einem Client-Zugriffe auf den im Konflikt stehenden Datensatz auf. Riak ist konfigurierbar, sodass auch die Anwendung Konflikte lösen kann, die durch Zugriffe oder durch einen asynchron laufenden Prozess erkannt werden.
Durch die Einschränkung der Schreibzugriffe bei Couchbase ist die Einordnung in die dritte Gruppe nur teilweise richtig, da Couchbase tatsächlich pro Datensatz Single-Master und nicht Multi-Master ist. Da allerdings im Fall einer Netzwerkpartition automatisch Schreibzugriffe auf Replikat-Knoten zugelassen werden, die Vergabe der Master/Slave-Rolle also durch das Consistent Hashing Verfahren automatisch erfolgt, ist die Einteilung in die Gruppe MMC akzeptabel.

Die vorgestellten Datenbanksysteme sind extrem skalierbar, verfügbar und verzichten teilweise auf Konsistenz, vorallem bei der Aktualität der Daten durch asynchrone Replizierung. Die Modellbezeichnung ist also BASE, wobei die Schwerpunkte im CAP-Theorem bei Cassandra und Riak durch Angabe von Quoren zwischen AP und CP liegt. Couchbase ist vorallem aufgrund der Single-Master Architektur pro Datensatz sehr konsistent, jedoch durch das Consistent Hashing Verfahren auch gegen Ausfälle tolerant. Die Wahl im CAP-Theorem fällt hierbei vorallem im Normalbetrieb auf CP. Im PACELC-Modell sind Cassandra und Riak PAEL, also verfügbar bei Netzwerkpartitionen und sonst ausgerichtet auf geringe Latenzzeiten.

Die Wahl eines Clients, welcher Knoten im Cluster initial kontaktiert wird, ist bei den drei betrachteten Datenbanksystemen prinzipiell beliebig, da jeder Knoten als serverseitiger Proxy für die Clientanfrage agieren kann. Bei Cassandra wird beim erstmaligen Kontakt zum Cluster eine Liste der Knoten heruntergeladen, sodass der Treiber auf clientseite optional zur Verbesserung der Verarbeitungszeiten eine Vorauswahl des Knotens für nachfolgende Anfragen treffen kann. Bei Couchbase ist ebenso ein serverseitiger Proxy vorhanden, allerdings wird die Vorauswahl auf Clientseite empfohlen.

Dauerhaftigkeit

Die Dauerhaftigkeit von Schreibanfragen wird bei Cassandra sehr streng garantiert, in dem ein Write-Ahead-Log Änderungen protokolliert. Bei Couchbase laufen die zu persistierenden Änderungen asynchron zur Anfrage, indem ein eigener Prozess eine Queue abarbeitet und keine strenge Garantie sondern *eventual persistent* gewährleistet wird. Bei Riak kann auf Konfigurations- oder Zugriffs-Ebene ein weiteres Quorum angegeben werden, wie viele Replikate die Schreibänderung zuverlässig persistiert haben sollen. Hierdurch ist von eventual persistent bis strenger Dauerhaftigkeit anwendungsspezifisch die Garantie flexibel einstellbar.

Zusammenfassung

Die drei beschriebenen Datenbanksysteme der dritten Gruppe MMC bauen auf Basis des Consistent Hashing Verfahrens auf. Die verwendeten Speichertypen unterscheiden sich dennoch, sodass besonders das Spalten-orientierte Datenbanksystem Cassandra von den beiden anderen Datenbanksystemen Riak und Couchbase abweicht. Die Partitionierung der Daten auf Basis von Consistent Hashing ermöglicht den Betrieb großer Cluster mit einer großen Anzahl an Knoten. Cassandra wird in Rechenzentren mit tausenden von Knoten betrieben, in denen die persönlichen Nachrichten der Social-Media-Plattform Facebook gespeichert werden. Besonders in diesen Größenordnungen muss das verteilte System mit Hardwareausfällen rechnen und entsprechend durch Replizierung dagegen vorbereitet sein. In Kombination mit Consistent Hashing können auf sämtlichen Replikaten Schreibzugriffe oder nur auf dem ersten Replikat im Ring gestattet werden. Cassandra und Riak sind daher tatsächlich Multi-Master für jede Partition, während Couchbase zur Wahrung einer strengeren Konsistenz pro Partition Single-Master mit Hot Standby Replikaten ist.

4.5. MMO: Multi-Master Offline-By-Default

Während die vorherig betrachteten Gruppen eine regelmäßige oder sogar ständige Verbindung zwischen den Knoten für eine konsistente Replizierung im Normalbetrieb voraussetzt, setzt *CouchDB*[11] [2] als Dokumenten-orientiertes Datenbanksystem keine Notwendigkeit dafür voraus. In einem Anwendungsfall von CouchDB sind die gleichberechtigten Master-Knoten nur unregelmäßig vorübergehend miteinander in Kontakt. Die Software des Datenbanksystems ist zudem schlank, was eine Installation auf Geräten wie Smartphones gestattet, welche ihre Daten z.B. mit einer entfernten Instanz auf einem Server synchronisiert. CouchDB kann dennoch auch mit einer ständigen Verbindung zwischen den Knoten genutzt werden. Die Bezeichnung “Offline-By-Default” beschreibt die Erwartung der Knoten des Datenbanksystems, dass grundsätzlich keine Verbindung zwischen den CouchDB-Instanzen besteht.

[11] http://couchdb.apache.org/

Replizierung und Skalierung

Die Replizierung kann manuell z.B. von der Anwendung angestoßen oder vom Datenbanksystem kontinuierlich erfolgen. Dabei kann uni- oder bidirektional zu einem entfernten anderen Knoten asynchron repliziert werden. Aus Sicht des initiierenden Knotens läuft die Kommunikation Push-basiert für die eine, jedoch Pull-basiert für die andere Richtung der Replizierung. Die Aktualisierung der Knoten verläuft also asynchron mit Pull und Push, wobei je Operation die neuen Daten übertragen werden. Abhängig davon, ob uni- oder bidirektional repliziert wird, sind alle Knoten Master-Knoten oder nur einzelne Knoten Master- und andere nur Slave-Knoten. Das Datenbanksystem unterstützt bei vollständiger bidirektionaler Replizierung allgemein die Architektur Multi-Master.

CouchDB unterstützt auf Ebene des Datenbanksystems keine Möglichkeit der Partitionierung. Die Replizierung erfolgt daher vollständig, wobei eine Auswahl einzelner Datenbanken, die repliziert werden sollen, konfiguriert werden können. Auf allen Master-Knoten sind Schreibzugriffe gestattet, die zu einem unbestimmten Zeitpunkt zwischen den Knoten ausgetauscht werden. Dadurch bietet CouchDB als Master Failover Cold Standby bis Warm Standby bei Verwendung einer kontinuierlichen Replizierung, die jedoch asynchron zu Schreibzugriffen stattfindet und daher kein Hot Standby garantiert wird. Da keiner der Master-Knoten eine speziellere Rolle zugewiesen bekommt, ist eine Vergabe einer Failover Stategie bei CouchDB nicht sinnvoll. Der Schwerpunkt der Anwendung liegt auf dem Begriff “Offline-By-Default” und nutzt die Replizierung weniger zur Lastverteilung oder Ausfallsicherheit sondern mehr zur Verfügbarkeit und Zugriff auf gespeicherte Daten an unterschiedlichen Geräten, Orten oder Infrastrukturen.

Da bei einer Multi-Master Architektur Konflikte auftreten können, deren Wahrscheinlichkeit bei eventuell unregelmäßiger, seltener Konnektivität zunehmen, bietet CouchDB eine Versionierung mit Verlauf vergangener Versionen der Datensätze an. Jeder Datensatz wird bei einem Schreibzugriff um eine Version erweitert, wobei die Versionskennung mit Vektor Uhren realisiert wird. Bei einer Replizierung werden alle Versionen eines Datensatzes komplett in eine oder beide Richtungen übertragen. Konflikte werden deterministisch vom Datenbanksystem gelöst und mit einem Hinweis gekennzeichnet. Die Anwendung kann anhand des Hinweises entscheiden, ob der Konflikt anders gelöst werden soll und z.B. zu einer anderen Version des Datensatzes wechseln. Der asynchrone Job, der die Replizierung ausführt und bei manueller Replizierung von der Anwendung gestartet werden kann, sorgt für die Ermittlung von Konflikten während der Replizierung.

Dauerhaftigkeit

Die Dauerhaftigkeit wird bei CouchDB streng garantiert. Durch die Erweiterung um Versionen bei jedem Schreibzugriff werden die Daten als *append-only* persistiert. Auch Löschoperationen werden als neue Version am Ende eines B-Baums auf Festplatte gespeichert. In Kombination mit Versionierung ist dadurch ein effizienter Zugriff auf Datensätze und dessen Versionen möglich. Zusätzliches Journaling ist nicht notwendig und wird nicht unterstützt, da Daten sofort append-only geschrieben werden. Beim Speichern muss zuvor ein Datensatz von der Anwendung gelesen werden und die Versionskennung mit übertragen werden. Dadurch können sog. Lost-Updates anderer paralleler Schreibzugriffe entdeckt und verhindert werden.

Zusammenfassung

CouchDB als Dokumenten-orientiertes Datenbanksystem bietet durch seine Eigenheit, die von der Produktbeschreibung als "Offline-By-Default" bezeichnet wird, sehr viele Anwendungsmöglichkeiten. Das Datenbanksystem lässt sich dadurch als Cluster betreiben, welches durch die append-only-Speicherung und die Verwendung von B-Bäumen effizient für Zugriffe nutzen lässt. Andererseits können Daten zwischen verschiedenen Orten oder Geräten synchron gehalten werden. In diesen Fällen ist das Datenbanksystem weniger in Rechenzentren sondern bspw. in Geräten von Endverbrauchern zu finden. Allerdings eignet sich die Verwendung von CouchDB nur für ein Datenvolumen, welches auf den entsprechenden Geräten gespeichert werden kann. Die fehlende Möglichkeit einer Partitionierung der Daten erlaubt keine horizontale Skalierung des Datenbanksystems.

4.6. RCL: Relationale Cluster

Die fünfte Gruppe umfasst das betrachtete Datenbanksystem *MySQL Cluster*[12] [22], welches das bereits vorgestellte Datenbanksystem MySQL um horizontale Skalierung erweitert. Das Prinzip bei relationalen Clustern ist übergreifend auf andere Implementierungen wie Erweiterungen von PostgreSQL das selbe. So werden verschiedene Knotentypen definiert, die Daten speichern, SQL-Anfragen ausführen oder die Partitionierung organisieren. Bei MySQL Cluster werden die Daten auf Tabellen-Ebene partitioniert. Dabei wird anhand des Primärschlüssels und einer Hash-Funktion verteilt und die Verteilung von sog. *Management Knoten* festgehalten.

Im Fall einer Anfrage eines Clients in SQL wird von einem sog. *Application Knoten* die SQL-Anfrage ausgeführt und anhand der Verteilung über die Management Knoten sämtliche zur Verarbeitung notwendigen Daten von den *Data Knoten* auf den *Application Knoten* kopiert. Dieser Vorgang hat zur Folge, dass SQL uneingeschränkt genutzt werden kann, da auch Joins mehrerer Tabellen oder Transaktionen auf dem Application Knoten ausführbar sind. Im Fall von Transaktionen werden die Datensätze auf den entsprechenden Daten Knoten gesperrt, da z.B. von parallel arbeitenden Application Knoten auf die selben Daten wegen einer ähnlichen oder gleichen SQL-Anfrage zugegriffen werden könnte. Der Kopiervorgang der Daten beeinflusst die Verarbeitungszeit einer Anfrage an das Datenbanksystem entsprechend. Um ausfallsicher zu sein, werden sämtliche Daten Knoten synchron als Multi-Master repliziert, was über die Organisations Knoten und die Application Knoten verwaltet wird. Schreibzugriffe sind also auf allen Replikat Knoten zulässig, jedoch aufgrund von verteiltem Locking nicht gleichzeitig möglich. MySQL Cluster bietet hierdurch eine strenge Konsistenz, auf Kosten der Latenzzeiten von möglicherweise parallel ausführbarer aber gegenseitig blockierender Anfragen.

MySQL Cluster bietet eine verteilte relationale Datenbank, die vollständig dem ACID-Modell entspricht. Im CAP-Theorem liegt der Schwerpunkt auf CP, im PACELC-Modell auf PCEC. Auch die Garantie der Dauerhaftigkeit ist streng. Zugriffe von Clients auf das Cluster erfolgen zentral über einen der Application Nodes, von denen einer vom Client gewählt werden kann.

[12] http://www.mysql.com/products/cluster/

Neben MySQL Cluster bietet eine Erweiterung von PostgreSQL einen sehr ähnlichen Ansatz an, der das relationale Datenbanksystem PostgreSQL clusterfähig macht. Durch die Anforderungen an horizontale Skalierung finden aktuell viele Veränderungen, Erweiterungen und Forschungen im Gebiet der relationalen, ACID-konformen Datenbanksystemen statt, die nach Lösungen für eine performante Partitionierung suchen [25]. Eine neue Kategorie von Datenbanksystemen, denen die vergleichsweise neuen Datenbanksysteme NuoDB oder VoltDB zugeordnet sind, nennt sich *NewSQL*. Diese Datenbanksysteme sind aufgrund des ähnlichen Ansatzes zu MySQL Cluster in dieser fünften Gruppe RCL zuzuordnen.

4.7. Zusammenfassung

Die fünf vorgestellten Gruppen von betrachteten Datenbanksystemen bieten für viele verschiedene Anwendungsszenarien Lösungen. Die Schwerpunkte im Bezug auf Latenzzeit, Konsistenz, Skalierung oder Redundanz liegen dabei teilweise weit auseinander. Einführend stellte die Abbildung 4.1 eine Einordnung der fünf Gruppen in die beiden Bereiche Konsistenz und Skalierbarkeit dar. Wie Wahl des geeigneten Datenbanksystems für eine Anwendung hängt von diversen Faktoren ab. Die Skalierbarkeit ist ein Faktor, der vorallem im Hinblick auf eine potentiell wachsende Anwendung beachtet werden sollte. Die Wahl eines schwer skalierbaren Datenbanksystems kann die Verfügbarkeit der Anwendung negativ beeinträchtigen. Auf der anderen Seite ist der Faktor der Konsistenz zu berücksichtigen. So hat eine Banking-Anwendung deutlich strengere Anforderungen als ein Blogging-Portal für Texte oder Fotos. Spezielle Anwendungsfälle wie die Synchronisierung auf mobile Endgeräte, eine streng garantierte Dauerhaftigkeit oder die Notwendigkeit einer Versionierung von Datensätzen sind weitere Faktoren, die aus dem vorliegenden Kapitel gezogen werden können.

Allgemein bieten die fünf Gruppen durch die Hauptmerkmale eine gute Übersicht der derzeit am häufigsten verbreitetsten Datenbanksysteme am Markt. Die Einteilung der Gruppen SML, SMP und MMC wird anhand der Eigenschaft der Replizierung Single-Master oder Multi-Master getätigt. Zusätzlich beschrieben die Gruppen die Möglichkeit der horizontalen Skalierung, die Lese-Skalierung, Partitionierung in einer Form wie Regions oder Consistent Hashing verwenden. Die beiden übrigen Gruppen beschreiben Spezialfälle, da MMO von keiner ständigen Verbindung zwischen den Knoten ausgeht und RCL relationale Datenbanken clusterfähig macht. Im Detail unterscheiden sich die Gruppen der Datenbanksysteme zusätzlich in Themen wie Konsistenz oder Dauerhaftigkeit. Die fünf Gruppen stellen eine Übersicht der vorhandenen Klassen an Datenbanksystemen im Hinblick auf Fehlertoleranz dar.

5. Design eines fehlertoleranten Datenbanksystems

Der bisherige Teil der Ausarbeitung beschäftigte sich mit Grundlagen und dem *Design*-Teil im Bezug auf zuverlässige und verfügbare Datenbanksysteme. Dort wurden allgemein nach Grundlagen Konzepte zur Realisierung von Fehlertoleranz in Datenbanksystemen vorgestellt und auf elf vorhandene Datenbanksysteme angewendet. Der nachfolgende zweite Teil der Ausarbeitung beschreibt in den Kapiteln 5 und 6 eine eigene Umsetzung einer *Implementierung* eines solchen Datenbanksystems. Zunächst wird in diesem Kapitel 5 ein Entwurf vorgestellt, dessen Umsetzung mit Virtual Nodes im nachfolgenden Kapitel 6 erläutert wird. Der Entwurf des eigenen Datenbanksystems soll einen Anwendungsfall abdecken, der zunächst beschrieben wird. Nachfolgend werden daraus resultierende Anforderungen vorgestellt. Abschließend werden Kernaufgaben des zu implementierenden Datenbanksystems erläutert.

5.1. Anwendungsfall

Als Anwendungsfall wird eine e-commerce-Anwendung betrachtet, die über einen Onlineshop große Mengen an Artikel an Endverbraucher verkauft. Abbildung 5.1 stellt schematisch eine grobe Aufteilung einer solchen Anwendung dar. Neben der Anwendung selbst ist die Datenhaltung von großer Bedeutung. Für einen Onlineshop sind die Entitäten Kundenstamm, Artikelstamm und Bestellungen von Bedeutung, die dauerhaft zuverlässig hinterlegt sein müssen. Außerdem ist der Warenkorb während eines Einkaufs, der auch unterbrochen und von einem anderen Ort vom Anwender fortgesetzt werden kann, von Bedeutung.

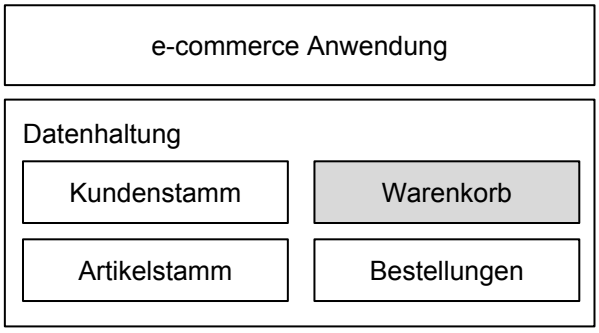

Abbildung 5.1.: Aufteilung einer e-commerce Anwendung

Das resultierende Datenbanksystem dieser Arbeit soll diesen Warenkorb in einer großen e-commerce Anwendung abbilden können. Dabei ist die zu speichernde Information sehr einfach, da eine Verknüpfung zwischen einer Kundennummer und mehreren Artikelnummern mit einer Stückzahl ausreicht. Dafür muss der Warenkorb sehr schnell verfügbar sein, um Wartezeiten für den Kunden und dadurch mit höherem Risiko den Abbruch seiner Bestellung zu vermeiden. Das Datenbanksystem muss also *performant* sein und Anfragen mit möglichst geringer Verarbeitungsdauer beantworten. Desweiteren darf der Warenkorb unter keinen Umständen ausfallen, da hierdurch in einem hoch frequentierten Onlineshop ein erheblicher finanzieller Schaden entstehen kann. Die Anforderung an die *Verfügbarkeit* ist also entsprechend hoch. Wird als Onlineshop eine weltweit betriebene e-commerce Anwendung der Größe von Amazon betrachtet, muss der Warenkorb und das dahinter stehende Datenbanksystem zudem hoch skalierbar sein. Eine vertikale Skalierung reicht hierbei nicht aus, da die Anforderung die Grenzen der möglichen Hardware überschreitet. Funktionale Skalierung ist, wie in Abbildung 5.1 dargestellt, bereits erfolgt sodass die Anwendung nicht weiter zerlegt werden kann. Das Datenbanksystem muss also *horizontal skalieren* und die zu speichernden Daten partitionieren.

Die Grundlage dieses Anwendungsfalls ist an die Veröffentlichung von Amazons Datenbanksystem Dynamo [13] angelehnt, welche für den im Onlineshop Amazon den Warenkorb der Kunden bereit hält. Nachfolgend werden die Anforderungen des Anwendungsfalls beschrieben und eine technische Umsetzung vorgestellt. Grundsätzlich soll das System performant, verfügbar und skalierbar sein.

5.2. Anforderungen

Dem Anwendungsfall sind die drei Anforderungen performant, skalierbar und verfügbar zu entnehmen. Das Datenbanksystem soll Anfragen mit geringer Verarbeitungszeit verarbeiten können, damit die darauf aufbauende Anwendung *performant* ist. Die zu speichernde Information kann sehr einfach gehalten werden und ist, da nur Nummern miteinander Verknüpft werden sollen, einzeln vom Speichervolumen gering. Das Datenbanksystem soll daher als NoSQL Datenbanksystem mit dem Speichertyp *Key/Value-basiert* implementiert werden. Auf einen Datensatz soll mit den Operatoren *get* und *store* lesend bzw. schreibend zugegriffen werden. Zusätzliche Operationen wie komplexere Abfragen, Suchen, Indizierungen oder Ähnliches sind für den beschriebenen Anwendungsfall nicht notwendig. Der Key der Datensätze muss so von der Anwendung gewählt werden, dass z.B. über die Kundennummer und einer fortlaufenden Nummer die Artikel im Warenkorb gefunden werden können. Ein Datensatz mit der Kundennummer als Key kann die Anzahl an gespeicherten Artikeln im Warenkorb speichern, damit die Keys der Artikel im Warenkorb generiert werden können. Alternativ können sämtliche Artikelnummern als Liste in einem Datensatz gespeichert werden, um den Abruf des Warenkorbs weiter zu optimieren.

Um geringe Latenzzeiten zu erzielen soll das Datenbanksystem rein *RAM-basiert* arbeiten. Eine dauerhafte Persistenzschicht im Datenbanksystem ist dadurch nicht notwendig und verringert Verarbeitungszeiten von Schreib- oder Lesezugriffen einer vergleichsweise langsamen Festplatte. Aufgrund des geringen Speichervolumens pro Datensatz ist zudem der RAM mehrerer Server in einem Cluster des Datenbanksystems im Bezug auf die Kapazität ausreichend. Die Implementierung des Datenbanksystems soll in Java erfolgen, da das verwendete Framework Virtual Nodes in Java vorliegt. Datensätze werden als einfache Java-Objekte der Anwendung realisiert, was die

Flexibilität des Datenbanksystems erhöht und aus dem Key/Valuebasierten Datenbanksystem eine einfache Form in Richtung eines Dokumenten-orientierten Datenbanksystems macht. Neben der Performanz sind die beiden umfangreicheren Kriterien skalierbar und verfügbar.

Skalierbar

Damit das Datenbanksystem horizontal *skalierbar* wird, müssen die zu speichernden Daten partitionierbar sein. Die Wahl eines Speichertyps, bei dem die Datensätze untereinander nicht verknüpft sind, macht das Verteilen der Daten auf mehrere Knoten in einem Cluster anhand des Keys einfach möglich. Da auch die Skalierung performant sein muss, soll das Consistent Hashing Verfahren verwendet werden. Damit ist das resultierende Cluster des Datenbanksystems automatisch hoch aber auch abwärts skalierbar. Sowohl die Datensätzen als auch die Knoten im Cluster müssen auf den Ring im Consistent Hashing Verfahren verteilt werden. Dazu kann eine Hashing-Funktion wie MD5 verwendet werden, die den Key oder die Kennung des Knotens in einen 128-Bit-Hashwert transferiert, dessen erste 64-Bit einer Position im Ring entsprechen. Für die Verteilung der Knoten im Ring könnte ein beliebiger verbesserter Algorithmus erweitert werden, der anhand der Belegung im Ring die Positionen der Knoten anordnet um die Lastverteilung im Cluster zu verbessern.

Tritt nun ein neuer Knoten dem Datenbanksystem bei, wird über die Hash-Funktion dessen Platz im Ring bestimmt. Dadurch verlagert sich die Zuständigkeit eines Wertebereichs im Ring von einem Knoten auf den neuen, sodass der neue Knoten die Daten des Wertebereichs vom bislang zuständigen Knoten kopieren muss. Möchte ein Knoten gezielt den Ring verlassen, muss dieser Vorgang umgekehrt geschehen, sodass die Daten auf den ersten Knoten im Ring gegen den Uhrzeigersinn verschoben werden.

Verfügbar

Schließlich muss das Datenbanksystem *verfügbar* sein. Dazu zählt die Lastverteilung der horizontalen Skalierung, jedoch muss eine Redundanz der Daten für Ausfallsicherheit sorgen. In Kombination mit Consistent Hashing können die nachfolgenden Knoten im Ring Kopien der Daten eines Knotens bereit halten. Die Anzahl an Replikaten soll konfigurierbar sein und muss vom Datenbanksystem bei zu wenigen vorhandenen Knoten im Cluster entsprechend reduziert werden. Damit das Gesamtsystem performant bleibt, soll die Replizierung asynchron zu Client-Anfragen erfolgen. Diese Eigenschaft definiert, dass die Replikate eventually consistent sind und im Fall eines Knotenausfalls noch nicht replizierte Daten verloren gehen. Für den Anwendungsfall eines Warenkorbs ist diese Eigenschaft zu Gunsten der Performance tolerierbar. Damit durch die asynchrone Replizierung keine Konflikte und dadurch ein inkonsistenter Zustand eintritt, soll pro Datensatz die Single-Master Architektur verwendet werden. Im Ring ist der erste Knoten dabei der Master, auf dem Schreibzugriffe gestattet sind. Änderungen einer Operation werden von diesem Knoten verarbeitet und die neuen Daten des Keys an die im Ring nachfolgenden Knoten repliziert, die als Slaves bereit stehen. Fällt der Master-Knoten aus, kann der erste nachfolgende Slave die Master-Rolle übernehmen und ein weiterer Knoten nachfolgend der Folge von Replikats-Knoten im Ring zusätzlich Slave werden. Durch die asynchrone Replizierung kann Warm Standby gewährleistet werden. Lesezugriffe sind theoretisch an Slave-Knoten

möglich, sollen aber zur Wahrung der Konsistenz nicht möglich sein. Die Slaves dienen daher nicht als Lese-Skalierung sondern zum Erhalt des Systems bei Knotenausfällen. Im Fall einer Netzwerkpartition könnten nach dem beschriebenen Modell mehrere Master-Knoten für einen Datensatz entstehen. Beim Vereinen des Clusters darf nur ein Master-Knoten weiter existieren, sodass einer seinen Zustand vollständig oder teilweise verwerfen muss. Das Datenbanksystem ist also PAEC, was Verfügbarkeit im Fall einer Partition und sonst Konsistenz bedeutet. Im CAP-Theorem entspricht diese Eigenschaft AP, da im Fall einer Partition das System Verfügbar ist, jedoch beim Lösen der Netzwerkpartition die Konsistenz wieder hergestellt wird.

Zusammenfassung

Das vorgestellte Entwurfskonzept im Hinblick auf die Anforderungen skalierbar und verfügbar mit performanter Verarbeitung resultiert also in einem NoSQL Key/Value-basierten Datenbanksystem, welches RAM-basiert arbeitet. Zur Partitionierung und Replizierung soll Consistent Hashing verwendet werden, wobei insgesamt zwar als Architektur Multi-Master, für jeden Datensatz separat betrachtet jedoch Single-Master verwendet wird. Die Grundlage des Entwurfskonzeptes ist eine Mischung aus bestehenden Datenbanksystemen wie Couchbase oder Riak der dritten Gruppe MMC aus Kapitel 4 und ist damit ähnlich zu Amazon Dynamo.

5.3. Kernaufgaben

Nachdem die Anforderungen an das Datenbanksystem und die zu verwendenden Konzepte beschrieben wurden, werden im Folgenden die Kernaufgaben zusammengefasst. Die Aufgaben des Datenbanksystems lassen sich in vier Teile gliedern und werden im Folgenden vorgestellt. Die ersten beiden Aufgaben beschreiben das Verhalten der Knoten im Cluster, wie ein Knoten neu oder nach einem Ausfall hinzu kommt und wie andere Knoten auf diese Veränderungen reagieren. Die beiden letzten Aufgaben definieren, wie Anfragen von Clients ausgeführt und anschließend repliziert werden sollen.

5.3.1. Hochfahren eines neuen Replikats

Die erste Kernaufgabe ist das Hinzufügen eines neuen Knotens zum Cluster. Ebenso tritt diese Aufgabe ein, wenn ein Knoten nach bspw. einem Absturz wieder dem Cluster beitreten möchte. Zunächst muss sich der neue Knoten mit der bestehenden Kommunikation des Clusters verbinden. Die bestehenden Knoten erhalten darüber die Information, dass ein neuer Knoten hinzukommen möchte. Der neue Knoten benötigt zunächst eine Kopie des aktuellen Consistent Hashing Rings, den er von einem der bestehenden Knoten erhält. Der Ring kann von allen Knoten oder gezielt von einem Knoten angefragt werden. Um Ausfallsicher zu sein ist eine Anfrage an alle Knoten sinnvoll, wobei die Antwort vom ersten antwortenden Knoten verarbeitet wird. Anhand dem deterministischen Hashing-Verfahren und dem bestehenden Ring kann der neue Knoten zunächst seine eigene neue Position ermitteln.

Anhand der neuen Position im Ring kann der Knoten berechnen, von welchen bestehenden Knoten er Daten erfragen muss. Diesen sog. Range an Daten kopiert sich der neue Knoten vom entsprechend zuständigen Master-Knoten. Zusätzlich muss der neue Knoten auch als Slave für im Ring vorher angeordnete Knoten bereitstehen, die er vom selben oder weiter gegen den Uhrzeigersinn liegenden Knoten im Ring kopieren muss. Der bestehende Master-Knoten muss sich ab dem Zeitpunkt des Kopiervorgangs alle eingehenden Schreibzugriffe zwischenspeichern, um diese dem neuen Knoten nachträglich zusenden zu können. Erfolgt keine Zwischenspeicherung, gehen Änderungen während des Hochfahrens des neuen Replikats verloren. Alternativ könnten während dieses Kopiervorgangs Schreibzugriffe blockiert werden.

Nachdem der neue Knoten alle für ihn relevanten Daten erfragt und vorliegen hat, wird der Knoten mit einer Nachricht an alle Knoten im Cluster sichtbar und verfügbar. Durch diese Aktualisierung können nun die bestehenden Knoten den neuen Knoten anhand der Hash-Funktion bei sich lokal dem Ring hinzufügen und ihre Aufgabenbereiche, die nun vom neuen Knoten übernommen wurden, entsprechend anpassen (vgl. 5.3.2).

5.3.2. Aktualisierungen im Consistent Hashing Ring

Die zweite Kernaufgabe des Datenbanksystems ist, dass bestehende Knoten im Cluster auf Änderungen reagieren und die Verteilung der Daten aktualisieren. Innerhalb der ersten Aufgabe ist bereits beschrieben, dass beim Hinzufügen eines neuen Knotens eine Aktualisierung beim Hinzufügen eines neuen Knotens erfolgt, worauf die Knoten ihre Zuständigkeit prüfen und gegebenenfalls anpassen. Diese Aktualisierung wird auch beim Verlassen eines Knotens z.B. im Fehlerfall nach einem gewissen Timeout aufgerufen. Allgemein muss auf ein Aktualisierung das lokale Replikat seine eigene Range überprüfen. Dabei wird immer der selbe Algorithmus ausgeführt, unabhängig von der tatsächlichen Veränderung des Hinzukommens, des Wegfallens oder der Anzahl von betroffenen Knoten.

Zunächst wird der Ring lokal aktualisiert. In der Aktualisierung enthaltene neue Knoten werden anhand der eindeutigen Kennung deterministisch über eine Hash-Funktion im Ring zugeordnet. Enthaltene Knoten, die entfernt wurden, werden entsprechend aus dem Ring entfernt. Anschließend berechnet anhand des neuen Rings der lokale Knoten, für welchen Bereich im Ring er zuständig ist. Dieser Range wird mit dem vorliegenden Range verglichen und, wie beim Hinzufügen eines neuen Knotens, ggfs. von einem anderen Knoten kopiert. Nicht länger benötigte Daten, deren Zuständigkeit verlagert wurde, können gelöscht werden. Durch die Replizierung von benachbarten Knoten im Ring wird die Aktualisierung zwar nur benachbarte Knoten direkt betreffen. Durch Veränderungen an einer beliebigen Stelle im Consistent Hashing Ring verschiebt sich jedoch die Zuständigkeit im kompletten Ring.

5.3.3. Client Requests ausführen

Damit jeder Knoten des Datenbanksystems Anfragen von Clients entgegen nehmen kann, soll neben der Fähigkeit Anfragen auszuführen jeder Knoten auch als Proxy für Client-Anfragen zur Verfügung stehen. Da beim Eintreffen einer Anfrage unklar ist, ob der vom Client gewählte Knoten selbst für die Verarbeitung zuständig ist oder als Proxy agieren muss, ist zunächst der

eintreffende Knoten Proxy, der die Anfrage ggfs. an sich selbst weiterleitet. Empfängt ein Knoten eine Client-Anfrage, muss dieser also zunächst anhand des Keys den zuständigen Bereich im Ring und darauf schließlich den zuständigen Knoten finden. Die Information der Verteilung von vorhandenen Knoten auf dem Consistent Hashing Ring liegt lokal vor, sodass der lokale Knoten den zuständigen Knoten für die eintreffende Anfrage performant durch lokale Berechnungen finden kann. Entspricht der zuständige Knoten dem lokalen Knoten, wird die Anfrage lokal ausgeführt. Ist ein anderer Knoten als der lokale Knoten zuständig, muss die Client-Anfrage gekapselt und als Unicast an den zuständigen Knoten versendet werden. Der lokale Knoten agiert nun tatsächlich als Proxy, speichert den Kommunikationspunkt zum Client und den zuständigen Knoten ab, damit bei einer Rückmeldung des Knotens der lokale Knoten die Antwort an den Client weiterleiten kann. Die Notwendigkeit des Proxys ist dadurch begründet, dass der Client eventuell nicht direkt von einem beliebigen Knoten erreichbar ist. Fällt zwischen dem Weiterleiten und der Antwort der Ziel-Knoten aus, kann der Proxy durch die Aktualisierung des Rings den Ausfall bemerken und die Anfrage erneut an den neu zuständigen Knoten weiterleiten. Bei der Entscheidung, ob ein Knoten für eine Anfrage zuständig ist oder nicht, werden dabei nur die Master-Knoten und nicht die Slaves berücksichtigt. Dadurch gelangen weder Schreib- noch Lesezugriffe an einen Slave, da dieser die Anfrage an den Master-Knoten weiterleiten würde.

Ein Knoten kann also Client-Anfragen entweder direkt vom Client oder von einem anderen Knoten erhalten. Die Ausführung darf dabei nicht mit anderen Anfragen auf den selben Datensatz kollidieren, sodass die Ausführung zwar Anfragen parallel verarbeiten darf, jedoch Anfragen auf den selben Datensatz sequentiell abarbeiten muss. Diese Eigenschaft hat zur Folge, dass Anfragen geblockt werden können, wenn auf den Datensatz bereits zugegriffen wird. Da die Operationen primitiv gehalten sind sollte dieses Locking nur kurzweilig sein, weshalb die Verarbeitungszeiten kaum beeinträchtigt werden.

5.3.4. Client Requests replizieren

Die letzte Kernaufgabe ist die Replizierung der neuen Daten nach einem Schreibzugriff. Nachdem ein Knoten eine Anfrage eines Clients bearbeitet und diesem das Ergebnis zurück gesendet hat, müssen im Fall einer Veränderung die Daten asynchron repliziert werden. Dazu muss der Knoten die nachfolgenden Replikate im Ring finden, die als Slaves verwendet werden sollen. Anschließend werden die neuen Daten in eine Nachricht verpackt und als Unicast an die ermittelten Knoten transportiert. Jeder Knoten muss daher mit eingehenden Replikats-Nachrichten umgehen können. Eine erhaltene Nachricht muss entpackt, die Zuständigkeit überprüft und die neuen Daten an die richtige Stelle im eigenen Datenspeicher kopiert werden.

Replikats-Nachrichten werden asynchron zur Client-Anfrage erzeugt und versendet, um dem Client sofort ohne zusätzliche Wartezeiten durch die Replizierung antworten zu können. Die Replizierung wird von einem separaten Thread ausgeführt, der eine Queue von Replikats-Nachrichten abarbeitet. Je nach Auslastung erfolgt die Replizierung sofort oder erst einige Zeit später, weshalb die Slaves eventually consistent sind. Ein Slave-Knoten hingegen verarbeitet eingehende Replikats-Nachrichten sofort und parallel. Daher muss die Reihenfolge der Nachrichten beim Slave mit der Reihenfolge des Versendens übereinstimmen, um einen konsistenten Zustand zu erreichen.

5.4. Zusammenfassung

Das vorliegende Kapitel beschreibt einen Anwendungsfall, für den ein fehlertolerantes Datenbanksystem entworfen werden soll. Ein mögliches Design eines solchen verfügbaren und skalierbaren Datenbanksystems stellen die Kapitel 5.2 und 5.3 in Aufteilung nach den Anforderungen und nach den Kernaufgaben des Systems dar.

Das resultierende Datenbanksystem soll zur Verteilung und dadurch optimaler horizontaler Skalierung das Consistent Hashing Verfahren verwenden. Zur Wahrung der Konsistenz soll für jede Partition ein Master vorhanden sein, der zur Ausfallsicherheit auf konfigurierbar viele Slaves repliziert. Die Slaves sollen, aufgrund der Anforderung einer schnellen Verarbeitung von Client-Anfragen, eventually consistent sein. Im Fall des Ausfalls eines Master-Knotens kann ein Slave-Knoten automatisch die Aufgabe des Masters übernehmen, sodass als Failover-Stategie Hot Standby verwendet wird.

Der Entwurf des Datenbanksystems aus dem aktuellen Kapitel soll auf Basis von Virtual Nodes implementiert werden. Das nachfolgende Kapitel beschreibt entsprechend die Umsetzung auf Basis des Frameworks.

6. Implementierung eines fehlertoleranten Datenbanksystems

Im vorherigen Kapitel 5 wurde das Design eines fehlertoleranten Datenbanksystems erläutert, dessen Implementierung auf Basis von Virtual Nodes im Folgenden skizziert wird. Zunächst werden die im vorherigen Kapitel beschriebenen Kernaufgaben auf Virtual Nodes bezogen betrachtet. Anschließend wird die Software-Architektur und die einzelnen Komponenten der Implementierung vorgestellt.

6.1. Implementierung mit Virtual Nodes

Die Kernaufgaben des in Kapitel 5 vorgestellten Designs eines fehlertoleranten Datenbanksystems sollen mit Virtual Nodes umgesetzt werden. Virtual Nodes verwendet zur Replizierung eine vollständige Replizierung der Knoten, sodass keine Partitionierung der Daten zur partiellen Replizierung vorgesehen ist. Die Replizierungen, die das Framework bereitstellt, sind aktive und passive Replizierung, wobei die passive Replizierung am ehesten für die Implementierung des Datenbanksystems geeignet ist [14].

Das Datenbanksystem, als spezielles verteiltes System, wird teilweise als Service Implementierung wie von Virtual Nodes vorgesehen auf das Framework aufsetzen. Die Speicherung der Key/Value-Paare sowie der Zugriff darauf kann auf Ebene des Services realisiert werden. Die beiden weiteren Themen der Implementierung sind Replizierung und Partitionierung und werden nicht im notwendigen Umfang vom Framework unterstützt.

6.1.1. Replizierung

Das Datenbanksystem soll durch die Partitionierung mehrere Master im Cluster haben. Jedoch ist die Replizierung einer Partitionierung eine Single-Master Architektur, welche der passiven Replizierung von Virtual Nodes gemäß dem Leader/Follower-Prinzip folgt. Allerdings sollen die Slaves eventually consistent sein, sodass die Replizierung unabhängig von der Verarbeitung der Client-Anfragen stattfinden soll. Da die bestehende passive Replizierung diese Eigenschaft nicht in gewünschter Form zur Verfügung stellt, jedoch eine entsprechende Anpassung möglich ist, kann eine eigene Sharding Replizierung auf Grundlage der passiven Replizierung verwendet werden. Der Ablauf einer Client-Anfrage mit anschließender Replizierung wird in Abbildung 6.1 dargestellt.

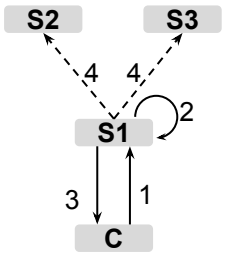

Abbildung 6.1.: Notwendige Replizierung der Implementierung

Im Vergleich zum Ablauf bei der passiven Replizierung in Abbildung 2.8 wird nach Eintreffen einer Client-Anfrage (1) die Anfrage ausgeführt (2) und das Ergebnis sofort an den Client zurück gesendet (3). Danach wird die Replizierung gezielt an die gewünschten Knoten als Unicasts ausgeführt (4). Die Unterscheidung zur passiven Replikation liegt darin, dass bei passiver Replikation die Replizierung vor der Antwort an den Client als Multicast stattfindet. Die notwendige Kommunikation zwischen den Knoten sowie die Kommunikation zwischen dem Client und dem Cluster erfolgen durch Komponenten das Virtual Nodes Frameworks.

6.1.2. Partitionierung

Während die Replizierung durch Anpassungen an der bestehenden passiven Replizierung entsprechend den Kernaufgaben des Designs implementiert werden kann, bietet Virtual Nodes keine Partitionierung des Clusters an. Das Framework definiert jeden Knoten als identisch im Bezug auf den Anwendungszustand, sodass von einer vollständigen Replizierung ausgegangen wird.

Das Datenbanksystem erfordert jedoch eine Verteilung des Anwendungszustands und damit verbunden die Verteilung der Zuständigkeiten im Fall einer Client-Anfrage. Jeder Knoten soll hierfür als Proxy agieren und dem Client einen beliebigen Einstieg in das Datenbanksystem ermöglichen. Die Funktionalität eines Proxys soll auf Ebene der Replizierung erweitert werden. Ebenso fehlt die Information beim Eintreffen von Client-Anfragen, welcher Knoten für die Anfrage zuständig ist. Neben der Funktionalität eines Proxys muss Virtual Nodes also um die Organisation der Verteilung erweitert werden, welche im Fall von Consistent Hashing die Verteilung der Knoten im Ring bedeutet.

Die fortlaufende Verteilung der Daten im Cluster wird durch die Replizierung und durch die Zustellung der Proxys an den zuständigen Knoten ausgeführt. Die Partitionierung muss jedoch im Fall einer Veränderung im Cluster angepasst werden. Virtual Nodes bietet ausreichend Unterstützung, um auf Veränderungen im Cluster reagieren zu können, sodass entsprechende Aktionen erweitert werden können. Für neue Knoten kann das Verfahren entsprechend wie in Abbildung 6.2 dargestellt erweitert werden. Virtual Nodes wird so erweitert, dass als Zustand der SET-Nachricht, die an einen neuen hochfahrenden Knoten übertragen wird, der Consistent Hashing Ring übertragen wird. Anhand dieses Rings kann der neue Knoten die notwendigen Daten anderer Knoten einholen. Die Zwischenspeicherung der zwischenzeitlich eintreffenden Anfragen muss in Virtual Nodes ebenso erweitert werden wie die Möglichkeit einen Bereich an Daten zwischen Knoten abzurufen bzw. abrufbar zu machen.

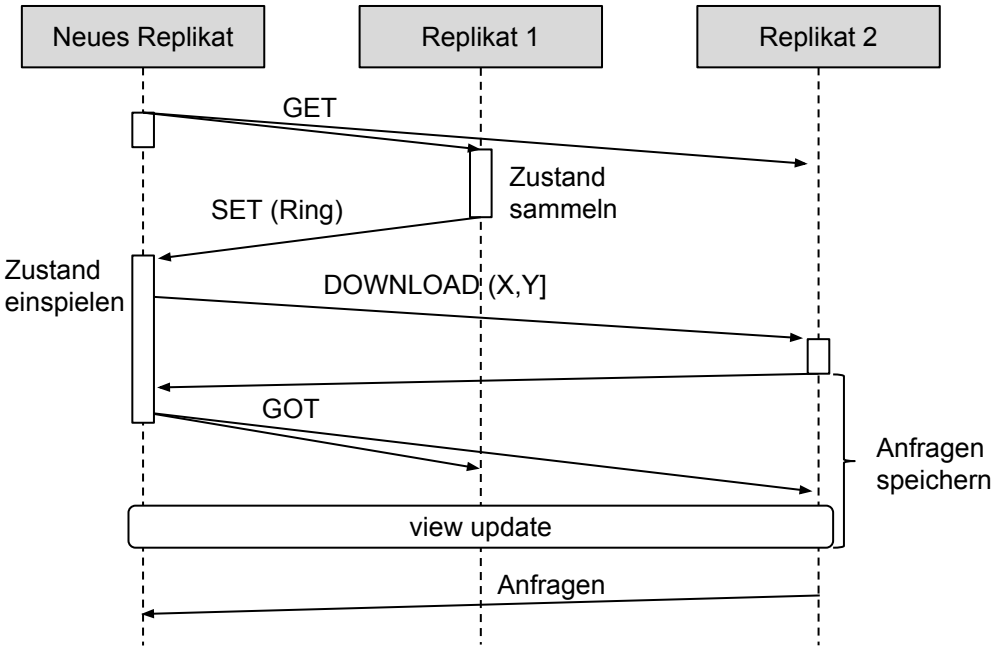

Abbildung 6.2.: Ablauf beim Hinzufügen eines neuen Knotens

6.2. Software-Architektur

Das Datenbanksystem auf Basis von Virtual Nodes besteht zunächst aus drei Komponenten, die von der passiven Replikation des Virtual Nodes Frameworks abgeleitet sind. Die erste Komponente ist die *Service Implementierung*, die gemäß des Frameworks auf dieses aufsetzt. Die zweite Komponente ist die *Middleware*, die das Virtual Nodes Framework mit der Service Implementierung verbindet. Die Middleware nimmt Anfragen auf Clientseite des Services entgegen und leitet diese an die darunter liegende Replikations-Schicht weiter. Auf serverseite empfängt die Middleware Anfragen von der Replikations-Schicht und leitet diese an die Service Implementierung. Virtual Nodes stellt u.a. eine RMI Middleware zur Verfügung, welche für das Datenbanksystem hinsichtlich der Partitionierung angepasst wurde. Die dritte Komponente ist schließlich die *Replikations-Schicht*.

Abbildung 6.3 stellt die drei Komponenten und ihre Abhängigkeiten untereinander graphisch dar. Durch die Notwendigkeit der Partitionierung sind die Komponenten der Replikation und der Middleware nicht ohne Anpassungen austauschbar. Die Implementierung der Middleware wird sowohl von der Replikations-Schicht als auch von der Service Implementierung benötigt. Die Replikation und die Service Implementierung stehen in keiner direkter Abhängigkeit zueinander, jedoch ist die Implementierung diverser Interfaces der Middleware auf Ebene des Services für die Replikation notwendig. Die nachfolgenden Kapitel 6.3, 6.4 und 6.5 erläutern die Implementierung der drei Komponenten.

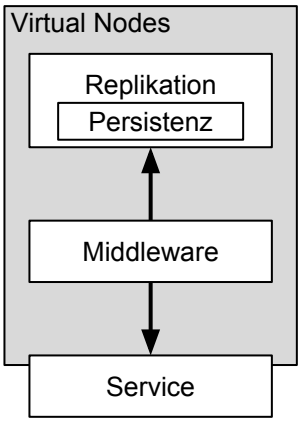

Abbildung 6.3.: Abhängigkeiten der drei Komponenten der Implementierung

6.3. Service Implementierung

Die Service Implementierung stellt drei Teile bereit. Eine Registry, die zum Hochfahren des Systems benötigt wird und auf RMI aufsetzt, den Client und den Server. Der Client kann Anfragen an das Cluster senden und dadurch die Methoden des Services über RMI aufrufen. Eine Client-Implementierung kann direkt in der Anwendung des Anwendungsfalls integriert sein oder als Proxy dienen, der z.B. Zugriff über eine REST-Schnittstelle ermöglicht. Der Server stellt den eigentlichen Service bereit, in dem er über die Registry eine Service-Implementierung exportiert. Die serverseitige Aufgabe des Datenbanksystems wird also vom Service bereitgestellt. Der Service enthält für den beschriebenen Anwendungsfall mindestens die Operationen get und store, kann optional noch Suchfunktionen oder Ähnliches für Clients bereit stellen.

Auf Ebene der Service Implementierung ist zudem definiert, welche Daten im Datenbanksystem gespeichert sein sollen. Dazu wird eine Klasse *DataItem* angelegt, die beliebige Klassenvariablen haben kann und so beliebige Daten speichern kann. Damit Objekte dieser DataItems gemäß der Service Implementierung korrekt im Netzwerk übertragen werden, muss ein *DataItemSerialiser* angelegt werden, der für die Serialisierung und Deserialisierung von DataItem-Objekten zuständig ist. Damit das gesamte Datenbanksystem zu speichernde Daten verteilen kann, ist ein Zugriff der Replikations-Schicht auf die Daten der Service Implementierung notwendig. Dabei muss eine Trennung des Anwendungszustands des Services und der im Datenbanksystem gespeicherten Daten erfolgen. Die Speicherung der Daten stellt die *StorageEngine* bereit, die für die Speicherung der DataItems des lokalen Knotens zuständig ist. Der Zugriff auf die StorageEngine findet über den sog. *PersistenceObjectSerialiser* statt, der ein Interface aus der Middleware-Schicht implementiert. Dieser Serialiser stellt Funktionen zur Serialisierung bzw. Deserialisierung des lokalen Anwendungszustands und der gespeicherten Daten für einzelne Keys oder einen Bereich von Keys bereit. Darüber kann die Replikations-Schicht im Fall einer Veränderung im Consistent Hashing Ring in jedem Knoten notwendige Daten platzieren.

6.4. Middleware-Schicht

Die Middleware-Schicht ist eine abgewandelte Variante der RMI Middleware aus Virtual Nodes. Die Middleware kennt z.B. die Methoden der serverseitigen Service-Implementierung. Sie bietet für die Clientseite einen Zugriff auf das Virtual Nodes Cluster und für die Serverseite Exportmöglichkeiten des Services. Die RMI Middleware wurde um notwendige Mechanismen erweitert, damit allgemein der Anwendungszustand von der Datenspeicherung getrennt und Partitionierung betrieben werden kann.

In der Middleware wird das *ReplicatedObject* definiert, also das Objekt, welches von Virtual Nodes verwaltet und repliziert werden soll. Dieses Objekt wird von der *Dispatcher*-Klasse der Middleware-Schicht implementiert. Abbildung 6.4 stellt einen Ausschnitt des Klassendiagramms der Middleware-Schicht und der bereits beschriebenen Service Implementierung dar.

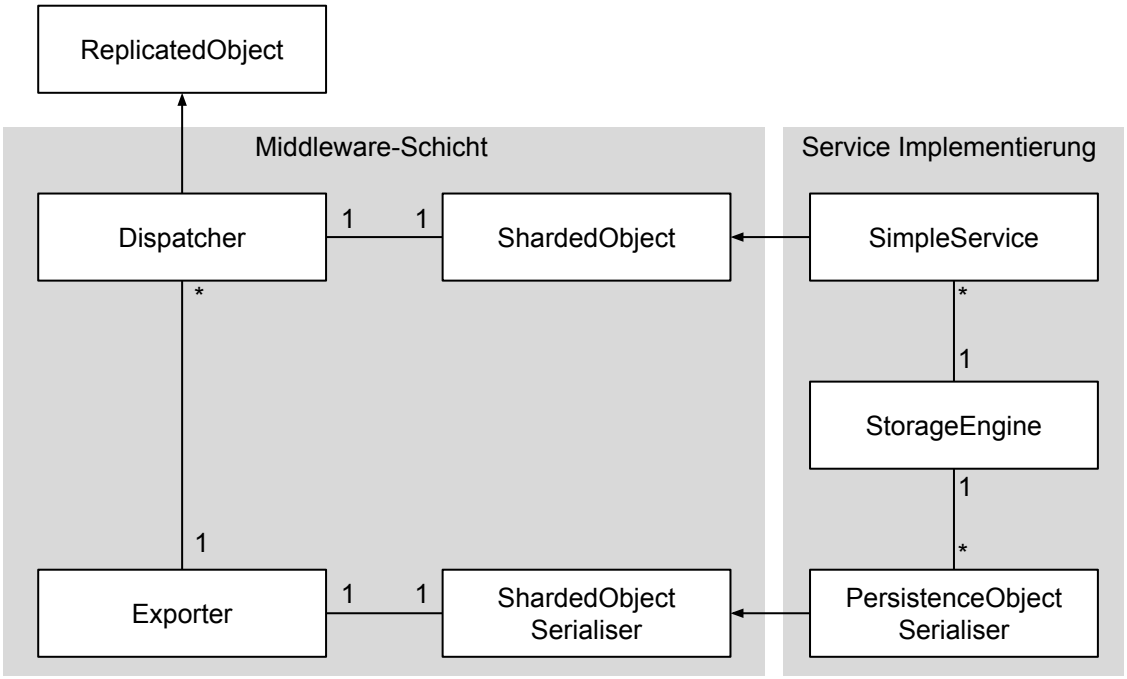

Abbildung 6.4.: Klassendiagramm der Middleware-Schicht und Service Implementierung

Beim initialen Start eines serverseitigen Services exportiert der Server über den *Exporter* ein Objekt, welches das Interface *ShardedObject* implementiert. Dem Exporter wird zudem eine Referenz auf einen *ShardedObjectSerialiser* gegeben, welcher für die Serialisierung und Deserialisierung des ShardedObjects zuständig ist. Die beiden Interfaces werden von Klassen der Service Implementierung implementiert, im Diagramm *SimpleService* und *PersistenceObjectSerialiser*. Der Exporter erzeugt beim Exportieren eines neuen Services ein Objekt des *Dispatcher*s. Aus Sicht von Virtual Nodes ist der Dispatcher das Replikat und der SchardedObjectSerialiser der dafür notwendige (De-) Serialisierer, um den Anwendungszustand des Replikats zu übertragen. Da für die Implementierung neben des Anwendungszustands der Datenspeicher (de-) serialisiert werden muss, bietet der PersistenceObjectSerialiser Zugriff auf die StorageEngine. Unabhän-

gig von der Service Implementierung muss der ShardedObjectSerialiser jedoch allgemein das ReplicatedObject, einen Range und einen ShardKey (de-) serialisieren können.

Der Exporter erzeugt zudem einen *ReplicationHandler*, der Methodenaufrufe an den lokalen Service von Virtual Nodes entgegen nimmt und über den Dispatcher am Service-Objekt ausführt. In diesem Methodenaufruf wird der erste Parameter, der an die Methode übergeben wurde, extrahiert und steht als sog. *ShardKey* zur Verfügung. Dieser ShardKey wird für die Partitionierung verwendet und ist im Fall dieser Service-Implementierung der Key des Key/Value-Datenspeichers. Die Aufteilung zwischen Service-Implementierung und Middleware ist jedoch gezielt flexibel gehalten, sodass beliebige Anwendungen, die partitioniert werden müssen, damit abgebildet werden können.

6.5. Replikations-Schicht

Die dritte, letzte und größte Komponente ist die Replikations-Schicht. Sie ist gemäß Abbildung 2.6 am weitesten in das Virtual Nodes Framework vertieft und hat die Aufgabe der Replikation. Wie in Kapitel 2.4.2 beschrieben, stehen bereits die aktive und die passive Replikation bereit. Die hier beschriebene Replikation nutzt grundlegende Elemente der passiven Replikation jedoch mit partieller statt vollständiger Replizierung. Die sog. *ShardingReplication* setzt in dieser Implementierung auf Consistent Hashing als Partitionierung auf, könnte theoretisch jedoch beliebige andere Verfahren nur Partitionierung nutzen. Im folgenden werden die beiden Teile *Sharding* und *Persistence* der ShardingReplication vorgestellt.

Sharding

Der erste Teil Sharding hat zur Aufgabe, die anfallenden Anfragen, deren Ausführung und anschließend deren Replizierung zu organisieren. In Kapitel 2.4.3 wurde das Programmiermodell von Komponenten in Virtual Nodes vorgestellt. Der dort genannte ReplicationInitialiser aktiviert durch entsprechende Konfiguration den *ShardingInitialiser* der betrachteten ShardingReplication Komponente. Dieser Initialiser sorgt zunächst für die Aktivierung der Sub-Komponente Persistence, in dem der *ShardingPersistencyInitialiser* aufgerufen wird.

Auf Sharding Ebene ist der ShardingInitialiser dafür zunächst zuständig, einen *ShardingReplicationContext* zu erzeugen, der zentrale Informationen für die Replikation sammelt und bereit stellt. Im Initialiser wird zudem der *ShardingSerialiser* erzeugt und an der sog. StateSerialisationRegistry registriert, um beim Sammeln des Anwendungszustandes des Virtual Nodes Knoten beitragen zu können. Der Zustand der ShardingReplication entspricht dem Consistent Hashing Ring und den darauf verteilten Knoten. Schließlich erzeugt der Initialiser das Kernobjekt der Komponente, eine Instanz der Klasse *ShardingReplication*. Die ShardingReplication wird für eingehende *ShardingClientRequestMessage*s, die über die Gruppenkommunikation eintreffen, registriert. Diese Nachrichten sind für die Weiterleitung von Client Anfragen innerhalb des Clusters notwendig, da die sog. *ClientReplicaMessage* dort verpackt und mit dem extrahierten ShardKey übertragen wird. Zudem exportiert der Initialiser die ShardingReplication als sog. *External Processor*, sodass die am Knoten eingehenden Client Anfragen (*ClientReplicaMessage*s)

von Virtual Nodes an diese Stelle weitergereicht werden. Der Initialiser registriert als letzten Handler den ShardingReplicationContext an der ViewRegistry, damit dieser über View Updates informiert wird.

Der *ShardingReplicationContext* erzeugt und beinhaltet den *ShardingRing*, eine Implementierung eines Consistent Hashing Rings mit Funktionen zum Hinzufügen und Löschen von Knoten oder zur Navigation im Ring. Der Context erzeugt außerdem ein Objekt von *ShardReplicaImpl*, was das lokale Replikat innerhalb der Komponente repräsentiert. Wichtige zentrale Informationen sind neben des Rings und dem Replikat auch Daten der Proxy-Funktion für ClientReplicaMessages. Im Context werden die Kommunikationskanäle zu allen noch offenenen Clients gesammelt. Dadurch bleiben die Clients für Knoten des Clusters erreichbar, da bspw. bedingt durch die Verwendung einer Firewall oder einer NAT nicht für eingehende Verbindungen erreichbar sind. Da der ShardingRepicationContext vom ShardingInitialiser für View Updates registriert wurde, muss der Context entsprechend im Fall einer Änderung für eine Aktualisierung des ShardingRings sorgen.

ShardingReplication ist die zentrale Klasse der ShardingReplication-Komponente und wird vom Initialiser instanziiert. Die Implementierung ist für die Nachrichtenverteilung zuständig, wobei *ClientReplicaMessage*s von Clients oder *ShardingClientRequestMessage*s von anderen Knoten eintreffen können. Für die Verarbeitung von ClientReplicaMessages ist der ExternalProcessor zuständig. Beim Eintreffen einer Anfrage wird zunächst der ShardKey extrahiert, indem die Replikation-Schicht in die Middleware-Schicht und damit dem Aufruf einer Methode über RMI einblickt. Anschließend wird mit Hilfe des ShardingRing das passende Replikat gefunden. Die ClientReplicaMessage mit dem zuständigen Replikat wird zur Verarbeitung an die *ShardReplica*-Implementierung gegeben. Trifft eine ShardingClientRequestMessage ein, wird zunächst untersucht, ob es sich um eine Anfrage oder eine Antwort handelt. Im Fall einer Anfrage wird die enthaltene ClientReplicaMessage entpackt und gemäß einer direkt eintreffenden ClientReplicaMessage verarbeitet. Im Fall einer Antwort wurde die enthaltene ClientReplicaMessage bereits verarbeitet und muss vom aktuellen Knoten an den Client übermittelt werden. Dazu wird der Kommunikationskanal anhand der MessageId aus dem Context geholt um die Antwort aus der ShardingClientRequestMessage zu extrahieren und per Unicast an den Client zu schicken.

Das lokale Replikat wird also von ShardReplicaImpl repräsentiert, wobei hier die tatsächliche Verarbeitung von ClientReplicaMessages stattfindet. Die ShardReplicaImpl nimmt hierzu Nachrichten entgegen und überprüft das übergebene zuständige Replikat. Ist ein fremdes Replikat zuständig, wird der Kommunikationskanal zum Client im Context abgelegt, die ClientReplicaMessage in eine ShardingClientRequestMessage verpackt und an das zuständige Replikat per Unicast übertragen. Ist die ClientReplicaMessage vom lokalen Replica zu verarbeiten, wird die Nachricht in eine sog. *LeaderExecutionLoop* gelegt. Dort werden Anfragen gemäß einer *ShardingStrategy* verarbeitet. Diese Strategie definiert, welche Aktionen vor oder nach einer Ausführung stattfinden sollen. Außerdem beinhaltet die Strategie einen Sheduler, der die zeitliche Ausführung der Anfragen bestimmt. Die ShardingStrategy definiert, dass nach jeder Ausführung das Ergebnis an die Persistence-Komponente übertragen wird. Der Scheduler sorgt dafür, dass pro ShardKey nur eine Anfrage zur selben Zeit ausgeführt wird.

Persistence

Die Persistence Komponente wird von der übergeordneten Shard Komponente geladen, in dem der ShardingPersistencyInitialiser aufgerufen und ausgeführt wird. Intern wird die Klasse *ShardingPersistency* instanziiert. Der Initialiser registriert diese ShardingPersistency für eingehende *PersistenceMessage*s, welche Daten eines ShardKeys nach einem Schreibzugriff von einem Master-Knoten an seine Slave-Knoten übertragen.

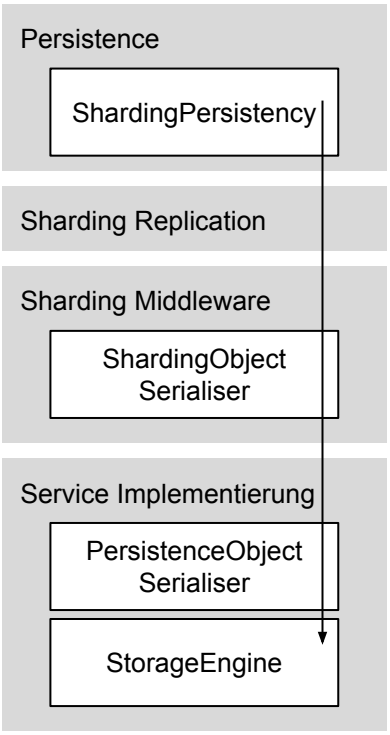

Abbildung 6.5.: Zugriff der ShardingPersistence auf Daten der Service Implementierung

Die wichtigste Klasse ist die *ShardingPersistency*, welche vier Aufgaben erfüllt. Zum Einen nimmt sie PersistenceMessages entgegen und pflegt die darin enthaltenen Daten lokal ein. Dabei wird über den ShardingObjectSerialiser der Middleware auf die Service Implementierung zugegriffen und auf Ebene der Service Implementierung für einen ShardKey die neuen Daten in der StorageEngine platziert. Zum Zweiten werden Daten nach einem Schreibzugriff an die Replikat-Knoten verteilt. Dazu findet ein Aufruf der ShardReplicaImpl nach jeder Ausführung einer Client Anfrage an der ShardingPersistency statt. Die Replizierung muss in einem neuen Thread oder Threadpool gestartet werden, damit die Client Anfrage nicht blockiert sondern die Replizierung asynchron erfolgt. Nachdem über den lokalen ShardingObjectSerialiser die aktuellen Daten des geänderten ShardKeys aus der lokalen Service Implementierung serialisiert wurden, werden die Slave-Knoten im lokalen ShardingRing ermittelt. Pro Knoten wird per Unicast eine PersistenceMessage versendet, welche die serialisierten Daten beinhaltet. Abbildung 6.5 stellt graphisch den Zugriff über die einzelnen Klassen der Persistenz-Schicht auf die Service Implementierung dar.

Die dritte Aufgabe der ShardingPersistency ist es, einen *SocketServer* zur Verfügung zu stellen. Der Port dieser Verbindung ist durch ein Hash-Verfahren aus der Replika-Id jedes Knotens im ShardingRing ableitbar, damit jeder Knoten von jedem anderen auffindbar und erreichbar ist. Eine eintreffende Verbindung überträgt den gewünschten Start- und Endwert, worauf der SocketServer mit den Daten dieses Bereiches antwortet. Die Daten werden, analog zur Beschaffung der Daten eines einzelnen ShardKeys, über die Middleware aus der Service-Implementierung extrahiert und direkt binär übertragen. Im Gegenzug zum SocketServer steht die vierte Aufgabe. Nach einer Änderung am ShardingRing startet der ShardingReplicationContext die Überprüfung der lokalen Daten an der ShardingPersistency. Zudem wird beim Hinzufügen eines neuen Knotens diese Funktion genutzt, um initial Daten zu beschaffen. Hierbei wird an einen ermittelten Knoten eine Socket-Verbindung aufgebaut, der benötigte Range übertragen und die erhaltenen Daten der Antwort in die lokale Service-Implementierung integriert.

Im Gegensatz zur passiven Replikation von Virtual Nodes enthält die Persistence Komponente hier keine Daten. Während bei der passiven Replikation der komplette, vollständige Zustand der Service Implementierung als Daten für alle Knoten in der Persistenz-Schicht vorliegen, muss in der ShardingReplication eine Partitionierung erfolgen. Dadurch, dass die Partitionierung auf Kriterien der Service Implementierung erfolgen muss (ShardKey) und die Daten sowohl von ShardingReplication als auch von der Service Implementierung erreichbar sein müssen, ist eine Speicherung in der Service Implementierung mit entsprechenden (De-) Serialisierern sinnvoll.

6.6. Diskussion

Die Implementierung des Datenbanksystems, dessen Design im Kapitel 5 vorgestelt wurde, auf Basis des Virtual Nodes Frameworks erfordert tiefe Eingriffe in das Framework. Das Virtual Nodes stellt ein Framework für fehlertolerante verteilte Systeme bereit und bietet hierfür aktive oder passive Replikation mit bspw. einer RMI Middleware als Schnittstelle für Service Implementierungen an. Werden keine zusätzlichen Mechanismen wie partielle Replikation oder eine Repliktion der Slaves mit eventual consistency ist eine verteilte Anwendung mir geringem Einarbeitungsaufwand erstellbar. Im Fall der vorliegenden Implementierung ist jedoch ein tiefer Einblick in Virtual Nodes notwendig.

Nach einer zeitintensiven Einarbeitungsphase in die Internas des Virtual Nodes Frameworks sind Anpassungen durch die Komponenten-orientierte Architektur und das Programmiermodell flexibel möglich. Durch diverse *Registry*s kann auf Ereigbnisse wie Veränderungen im Cluster oder neu eingehende Nachrichten reagiert werden. Das Versenden und Empfangen von Zuständen und Nachrichten erfordert jeweils die Implementierung aufwändiger und fehleranfälliger Serialisierungs- und Deserialisierungs-Mechanismen. Dadurch ist jedoch eine beliebige Erweiterung um neue Nachrichten oder zusätzliche Inhalte in bestehende Nachrichten oder Zuständen möglich.

Die notwendige Verteilung zur Partitionierung erfordert Einblicke der Replikations-Schicht in die Client-Anfragen auf Informationen der Middleware- oder Service-schicht, um den erfragten Key zu erhalten. Im bestehenden Virtual Nodes Framework ist aufgrund der Austauschbarkeit und Trennung der Komponenten kein solcher Zugriff möglich, der durch die Implementierung der eigenen Middleware hergestellt wird. Ein alternativer Ansatz zur Lockerung der daraus

entstehenden Kopplung der Komponenten wäre ein noch tieferer Eingriff in die Verarbeitung der *ClientReplicaMessage*s, sodass die Middleware beliebig Informationen hinzufügen kann, die von der Replikations-Schicht ausgewertet werden können. Eine ähnliche Problematik ergibt sich daraus, dass die Replikations-Schicht auf die partitionierbaren Daten der Service Implementierung zugreifen muss. Dieser Zugriff, der vom Virtual Nodes Framework nicht unterstützt wird, ist durch die Einführung neuer (De-)Serialiser im Stil des Frameworks umgesetzt worden.

Zusammenfassend unterstützt Virtual Nodes die Implementierung fehlertoleranter verteilter Systeme sehr gut. Für spezielle Anwendungsfälle wie der vorliegenden Implementierung des Datenbanksystems sind Eingriffe realisierbar, jedoch mit initialem Aufwand verbunden. Durch die Organisierung in Komponenten liese sich durch zusätzliche, strengere Einführung von dokumentierten Schnittstellen zwischen Komponenten und Generalisierungen diverser interner Nachrichten das Frameworks könnten künftige Erweiterungen einfacher realisiert werden. Zusätzlich wären Hilfskomponenten für wiederkehrende Implementierungen wie (De-)Serialiser denkbar, um den Aufwand von Ergänzungen zu reduzieren. Dennoch nimmt das Framework vorallem im Bezug auf die Kapselung der Clusterbildung, von Client-Anfragen oder Cluster-interner Kommunikation enormen Aufwand gegenüber einer Neuentwicklung ab.

6.7. Zusammenfassung

Die Implementierung des fehlertoleranten Datenbanksystems auf Basis von Virtual Nodes besteht zunächst aus zwei Komponenten. Zum einen wurde die passive Replizierung auf eventually consistent der Slave-Knoten angepasst. Zum anderen wurde die vollständige Replizierung durch eine partielle Replizierung mit Consistent Hashing ersetzt. Auf Ebene der Replikations-Schicht können auf dem Framework aufbauende Anwendungen durch diese Erweiterung partitioniert werden.

Die Implementierung besteht zunächst aus der Anwendung selbst, die als Service Implementierung auf Virtual Nodes aufsetzt. Damit die neu entwickelte Replikations-Schicht Partitionierung beherrscht, ist zudem eine Anpassung der Middleware-Schicht notwendig. Die drei Komponenten der vorliegenden Implementierung sind also die Service Implementierung, die Middleware-Schicht und die Replikations-Schicht. Die drei Komponenten sind abhängig voneinander und ersetzen bestehende Komponenten des Virtual Nodes Frameworks.

Die Service Implementierung stellt die Speicherung der Key/Value-Datensätze und den Zugriff über die Operationen get und store ur Verfügung. Die Middleware basiert auf der RMI Middleware von Virtual Nodes und unterstützt zusätzliche Serialiser zum Zugriff auf einzelne Partitionen. Zusätzlich filtert die Middleware den sog. ShardKey aus der Client-Anfrage heraus, anhand dessen der Zuständige Knoten im Cluster berechnet werden kann. Die Replikations-Schicht ist die größte Komponente und stellt neben der Verarbeitung von Client-Anfragen den Proxy für Clients und die Persistierung in Form der Replikation auf die Slaves bereit.

7. Diskussion

Im vergangenen Jahrzehnt sind Veränderungen in der Verwendung eingesetzter Datenbanksysteme sichtbar geworden. Lange Zeit waren relationale Datenbanksysteme als Speicherort für Anwendungen aller Art verwendet worden, die durch das ACID-Modell eine strenge Konsistenz gewährleisten. NoSQL Datenbanksysteme lösen Skalierbarkeitsprobleme relationaler Datenbanksysteme, in dem das Speichermodell Datensätze weniger stark untereinander verknüpft und somit einfache Partitionierung der Daten ermöglicht. NoSQL Datenbanksysteme ermöglichen durch die lose Kopplung der Datensätze häufig mehrere Knoten in einem Cluster und skalieren durch eine einfachere Verteilung der Daten horizontal besser als bewährte relationale Datenbanksysteme mit verknüpften relationalen Datensätzen. Jedoch verfolgen die meisten der NoSQL Datenbanksysteme das BASE-Modell und garantieren dadurch keine strenge Konsistenz sondern die clientzentrierte Konsistenz *eventually consistent*. Diverse Datenbanksysteme legen ihre Schwerpunkte nun, wie in Kapitel 4 dargestellt, in einem Bereich zwischen skalierbar und konsistent.
Vor Beginn der Entwicklung einer datenbankgestützten Anwendung muss also eine Abwägung der Kriterien skalierbar oder konsistent für die Wahl eines geeigneten Datenbanksystems erfolgen. Erste Ansätze relationale Datenbanksysteme clusterfähig zu machen um skalierbar und konsistent gleichermaßen zu sein sind zwar verfügbar und werden aktuell von einer neuen Bewegung unter dem Begriff NewSQL vorangetrieben. Da jedoch viele Anwendungen, die derzeit auf NoSQL Datenbanksysteme setzen, als zusätzlichen Faktor neben hoher Skalierbarkeit die Anforderung an geringe Latenzzeiten haben, fällt die Wahl aufgrund längerer Latenzzeiten nicht auf die ACID-basierten, relationalen Datenbanksysteme. Die Verbreitung und Weiterentwicklung der Datenbanksysteme scheint noch kein Ziel erreicht zu haben, sodass nach einem längeren und noch aktiven Sieg der relationalen Datenbanksysteme neue Ansätze und Technologien zu erwarten sind.

In dieser Ausarbeitung wurden elf Datenbanksysteme betrachtet und in fünf Gruppen eingeteilt. Diese Gruppen unterteilen Datenbanksysteme nach Kriterien der Fehlertoleranz wie Single-Master oder Multi-Master und der Möglichkeit einer Partitionierung. Die Gruppenbildung passt auf die betrachteten elf Datenbanksysteme, kann jedoch beim Hinzufügen weiterer, eventuell exotischer Datenbanksysteme nicht ausreichend sein. Da die betrachteten Datenbanksysteme nach ihrer Bekanntheit und Zahl der Installationen gewählt wurden, sind die fünf Gruppen aus Kapitel 4 jedoch durchaus für ein breites Spektrum an existierenden Datenbanksystemen repräsentativ. Da vorhandene Literatur Datenbanksysteme üblicherweise aus der Perspektive der Speichertypen betrachten, ist die neue, alternative Klassifizierung aus der Perspektive der Fehlertoleranz für einen Großteil an Datenbanksystemen ausreichend. Im Hinblick auf Fehlertoleranz sind zudem die Kriterien aus Kapitel 3 auf jedes Datenbanksystem anwendbar, sodass ein Vergleich zwischen beliebigen Datenbanksystemen ähnlich zur Tabelle in Anhang A möglich ist.

Teil der Ausarbeitung ist die Implementierung eines Datenbanksystems auf Basis von Virtual Nodes (vgl. Kapitel 6). Die Implementierung setzt dabei auf Konzepte der Gruppe MMC aus Kapitel 4, sodass eine Partitionierung und Replizierung entsprechend des Consistent Hashing Verfahrens umgesetzt wurde. Um ein Datenbanksystem produktiv für eine Anwendung zu implementieren sollte die Implementierung im Hinblick auf Sicherheit und Performance optimiert sein. Die vorliegende Implementierung dient als *Proof of Concept* und soll die Möglichkeit eines Datenbanksystems mit Virtual Nodes untersuchen. Die Implementierung zeigt, welche Komponenten in Virtual Nodes ausgetauscht werden müssen und ob dadurch die Verwendung des Frameworks überhaupt sinnvoll ist. Dass durch Virtual Nodes erheblicher Aufwand gegenüber einer vollständig eigenständigen Implementierung abnimmt, wurde in Kapitel 6.6 dargelegt. Jedoch ist die Verwendung des Ergebnisses aus verschiedenen Gründen nicht für ein Produktivsystem geeignet. Allgemein eignet sich Virtual Nodes dennoch gut für die Erweiterung und die Implementierung eines Datenbanksystems.
Die Implementierung müsste für die Verwendung in einem Produktivsystem grundlegend im Hinblick auf Laufzeitoptimierung untersucht werden, um eine maximale Performanz zu erzielen. Zudem wird mit zunehmendem Datenvolumen die gewählte Kommunikation zur Replizierung und Partitionierung nicht korrekt ausgeführt werden, da das Volumen der Nachrichten über die verwendete Gruppenkommunikation auf wenige MB begrenzt ist. Zudem wird beim Hochfahren eines neuen Knotens der benötigte Anwendungszustand über ein TCP-Socket heruntergeladen. Bei entsprechendem Datenvolumen kann mit erheblichen Verzogerungen des neuen Knotens aber auch des bestehenden Knotens durch dessen zusätzliche Auslastung neben seiner eigentlichen Aufgabe entstehen.
Neben der Performanz ist die Sicherheit in der gesamten Implementierung vernachlässigt worden. Um als Datenbanksystem produktiv einsetzbar zu sein sollte eine Authentifizierung von Clients und neuen Knoten stattfinden. Momentan kann durch Verbinden zu einem beliebigen Client die Funktion des Herunterladens des Anwendungszustands für neu hinzukommende Knoten verwendet werden um Kopien der gespeicherten Daten zu erhalten. Die beiden betrachteten Schwachstellen der Implementierung wurden jedoch bewusst akzeptiert, da die Schwerpunkte auf Zuverlässigkeit und Verfügbarkeit im Rahmen von Fehlertoleranz liegen.

8. Zusammenfassung und Ausblick

Die vorliegende Ausarbeitung besteht aus den beiden Teilen “Design” und “Implementierung”. Unter dem Themenfeld “Design” wurden Datenbanksysteme im Bezug auf deren Umsetzung von Fehlertoleranz vorgestellt und in Klassen eingeteilt. Zudem wurde parallel zur Ausarbeitung eine Implementierung angefertigt, die das Virtual Nodes Framework im Hinblick auf die Implementierbarkeit eines Datenbanksystems untersucht.
Im Folgenden werden zunächst die Inhalte der Ausarbeitung zusammengefasst. Anschließend wird ein Ausblick beschrieben, in dem zukünftige Arbeiten im Bereich dieser Arbeit genannt werden. Dazu gehört eine erweiterte Recherche oder diverse Erweiterungen der Implementierung.

8.1. Zusammenfassung

Wie in den Grundlagen in Kapitel 2 vorgestellt, werden Datenbanksysteme üblicherweise zunächst in relationale und nicht relationale Datenbanksysteme unterteilt. Zu den nicht relationalen Datenbanksystemen gehören die NoSQL Datenbanksysteme, deren Speichertypen weiter unterteilt werden können. Diese Kategorisierung von Datenbanksystemen orientiert sich stark an der internen Struktur der Implementierung, wie die Speicherung oder die Art der Manipulation von Daten. Die Kategorien sagen jedoch wenig über die Fehlertoleranzen der Datenbanksysteme aus, wozu die zusätzliche Klassifizierung aus Kapitel 4 in fünf Gruppen zu jedem Datenbanksystem erstellt wurde. Die Auswahl eines Datenbanksystems erfolgt von Anwendungsentwicklern entsprechend der Anforderungen an Skalierbarkeit und Konsistenz, was anhand der Zuordnung des Datenbanksystems in eine der Gruppen die Einordnung des Datenbanksystems im Bezug auf Fehlertoleranz für eine grobe Übersicht auf einen Blick ermöglicht.

Datenbanksysteme, als spezielle Form eines verteilten Systems, sind durch Skalierung und Replizierung fehlertolerant. Zunächst lassen sich verschiedene Konzepte finden, nach denen sich Datenbanksysteme richten. Diese in Kapitel 3 vorgestellten Konzepte lassen sich zunächst in die Bereiche Replizierung, Partitionierung und Dauerhaftigkeit einteilen. Durch die redundant gehaltenen Daten ergeben sich zusätzlich die Anforderungen Konfliktmanagement und Konsistenz. Die Konzepte wurden jeweils tabellarisch zusammengefasst, sodass eine Sammlung von Kriterien und möglichen Werten jedes Kriteriums ergeben. Diese Sammlung von Konzepten lässt sich auf beliebige fehlertolerante Datenbanksysteme anwenden. Die Ausarbeitung enthält in Anhang A einen entsprechenden tabellarischen Vergleich von elf betrachteten Datenbanksystemen. Die Sammlung der Konzepte und die darauf aufbauende Klassifizierung der Datenbanksysteme dient als Übersicht, wie Fehlertoleranz in Form von Zuverlässigkeit und Verfügbarkeit in Datenbanksystemen realisiert wird. Dieses Verständnis ist für den zweiten Teil der Ausarbeitung, dem Entwurf und der Implementierung eines eigenen Datenbanksystems notwendig.

Der Vergleich von verbreiteten Datenbanksystemen lässt also auf fünf Gruppen schließen, die in Kapitel 4 definiert wurden. Die Gruppen definieren die Hauptmerkmale der enthaltenen Datenbanksysteme. So betrachten die drei Gruppen SML (Single-Master und Lese-Skalierung), SMP (Single-Master und Partitionierung) und MMC (Multi-Master und Consistent Hashing) die verwendete Anordnung der Knoten innerhalb des Clusters nach Single-Master oder Multi-Master und die Möglichkeit der horizontalen Skalierung. Die beiden übrigen Gruppen MMO (Multi-Master Offline-By-Default) und RCL (Relationale Cluster) beschreiben zwei Sonderformen von Datenbanksystemen, wobei bei der ersten Gruppe keine zuverlässige Verbindung zwischen den Knoten erwartet wird und in die zweite Gruppe horizontal skalierbare relationale Datenbanksysteme eingeordnet sind, wobei einfache gängige relationale Datenbanksysteme der Gruppe SML angehören. Die Klassifizierung bietet eine Übersicht auf einem übergreifenden, höheren Abstraksionsniveau, welche Konzepte der Fehlertoleranz von vorhandenen Datenbanksystemen verwendet und unterstützt werden.

Der zweite Teil der Ausarbeitung beschäftigt sich mit der Implementierung eines eigenen Datenbanksystems auf Basis des Virtual Nodes Frameworks. Zunächst wurde in Kapitel 5 ein Design eines fehlertoleranten Datenbanksystems vorgestellt. Dieser Entwurf orientiert sich an den Datenbanksystemen aus der Gruppe MMC und setzt daher Consistent Hashing zur Partitionierung und Replizierung ein. Als Motivation wurde der Anwendungsfall beschrieben, der das Datenbanksystem als Warenkorb in einer e-commerce-Anwendung verwenden soll. Die Anforderungen an das Datenbanksystem sind performant, skalierbar und verfügbar, weshalb die Implementierung eine RAM-basierte Key/Value-Datenhaltung mit Consistent Hashing und Single-Master zur Wahrung der Konsistenz verwenden soll.

Nachdem das Design des zu implementierenden fehlertoleranten Datenbanksystems mit einigen definierten Kernaufgaben beschrieben wurde, folgt in Kapitel 6 die Erläuterung, wie die Implementierung auf Virtual Nodes übertragen werden kann. Die benötigte Software-Architektur unterteilt die Implementierung in drei Komponenten, die verschieden tief in das Virtual Nodes Framework eingreifen. Die Service Implementierung setzt auf Virtual Nodes auf und realisiert die Get- und Store-Operationen des Datenbanksystems. Die Middleware-Schicht sorgt für die einfache Implementierung der Service-Schicht und verknüpft diesen Service mit Virtual Nodes. Die Replikations-Schicht schließlich ist für die Ausführung, Partitionierung und Replizierung innerhalb des Clusters zuständig. Die sog. Sharding-Replikation ist an die Komponente für passive Replikation des Virtual Nodes Frameworks angelehnt und im Hinblick auf Sharding für Consistent Hashing erweitert worden. Das Kapitel zur Implementierung zeigt zudem, dass Virtual Nodes zwar mit einigen tiefgreifenden und komplexen Änderungen aber dennoch erfolgreich für den Einsatz eines Datenbanksystems genutzt werden kann.

8.2. Ausblick

Die Ergebnisse der gesammelten Konzepte und der Klassifizierung des Design-Teils der Ausarbeitung lassen sich auf weitere Datenbanksysteme anwenden. Durch die teilweise Verdrängung von relationalen Datenbanksystemen durch NoSQL Datenbanksysteme entstehen aktuell relationale Datenbanksysteme, die den Grund der Abwanderung zu NoSQL Datenbanksystemen in Angriff nehmen. Fehlende Skalierbarkeit durch mangelnde Partitionierung auf Ebene des Datenbanksystems sind Gründe, die gegen klassiche relationale Datenbanksysteme sprechen. Da hinter

vielen der relationalen Datenbanksysteme große Firmen wie Oracle stecken, ist mit weiteren Bewegungen und neuen Technologien im Bereich der relationalen Cluster zu rechnen. Bspw. ist MySQL Cluster im Vergleich zu MySQL oder der NoSQL-Bewegung deutlich jünger. Eine mögliche Erweiterung dieser Arbeit ist also ein Blick in die aktuellen Forschungen der Hersteller von Datenbanksystemen. Zudem kann die Analyse in Anhang A um weitere Datenbanksysteme erweitert werden, um die Klassifizierung der fünf Gruppen weiter zu verifizieren.

Die Implementierung im zweiten Teil der Ausarbeitung weißt die in Kapitel 7 beschriebenen Schwachstellen wie Sicherheit oder fehlende Optimierung der Performanz für den Betrieb des Datenbanksystems in einer produktiven Umgebung auf. Eine Erweiterung der Implementierung kann den Aspekt der Performanz oder der Sicherheit bearbeiten. Ebenso ist eine alternative Middleware statt RMI bspw. mit HTTP ein interessanter Ansatz, um beliebige Clients ohne das Virtual Nodes Framework über HTTP und bspw. REST mit dem Datenbanksystem verbinden zu können. Auf Ebene der Implementierung kann zudem eine Erweiterung einer dauerhaften Persistierung auf eine Festplatte jedes Knotens denkbar. Die RAM-basierte Lösung verliert durch Neustarten des kompletten Clusters sämtliche Daten. Eine Persistierung von Snapshots mit einer Garantie der Dauerhaftigkeit von Eventual Persistent kann für Backups und gezielte Neustarts verwendet werden.

Die Optimierung der Performanz der Implementierung ist eine Option, die vorweg durch Benchmarks und einer ausführlichen Evaluation des resultierenden Datenbanksystems begründet werden kann. Interessante Aspekte einer Evaluation des Datenbanksystems mit Virtual Nodes könnten ein Vergleich mit einem alternativen Datenbanksystem sein. Außerdem kann betrachtet werden, wie sich das Datenbanksystem unter verschiedenen Auslastungen verhält.

A. Tabellarischer Vergleich

Teil der beschriebenen Arbeit dieses Dokuments ist eine tabellarische Gegenüberstellung von elf verbreiteten Datenbanksystemen nach in Kapitel 3 vorgestellten Kategorien aus dem Bereich der Fehlertoleranz. Eine Auswertung der Analyse, deren Ergebnis nachfolgende Tabelle darstellt, findet sich in Kapitel 4. Die Tabelle enthält Anmerkungen, welche zunächst abgedruckt sind. Nachfolgend befindet sich die Tabelle selbst.

1 Berücksichtigung ohne Cluster-Support von Fremdsoftware!
2 Berücksichtigung ohne Cluster-Support!
3 ehemals Membase, Zusammenschluss mit CouchOne (Hersteller von CouchDB)
4 Neo4J ist in verschiedenen Varianten erhältlich (Community, Enterprise, mit Apache Zookeper). Hier: Enterprise (Replizierung, ohne Sharding)
5 Keine. Es stehen aber verschiedene Datentypen zur Verfügung wie Listen, Hashtables, Sorted Lists
6 Auf Collection-Ebene ohne Sharding, mit Sharding auf Shard-Ebene.
7 Einzelne Shards betrachtet jedoch Single-Master
8 Allerdings durch Sharding auch Multi-Master!
9 Je nach Konfiguration Warm oder Hot-Standby (Poll&Hold)
10 Je nach Konfiguration Warm oder Hot-Standby (Poll&Hold)
11 Replizierung kann synchron als Push oder asynchron als relgemäßiges Poll erfolgen. Daher Cold bis Hot-Standby möglich.
12 Außer mit Sharding, dann partielle Replikation.
13 Drei aufeinander aufbauende Stufen: 1) asynchron logbasiert 2) streambasiert 3) synchron (Master wartet auf einen Slave)
14 Cluster-intern. Cluster lassen sich komplett asynchron replizieren, mit Konflikterkennung.
15 Client kontaktiert irgendeinen Knoten, der Leitet Anfrage an entsprechende Knoten weiter. Client bekommt je nach gewähltem Quorum nach Rückmeldung von x Knoten Antwort.
16 Slave sendet SYNC-Kommando an übergeordneten Server. Dieser erstellt Dump-File, welches Slave einspielt.
17 Zwei Replizierungen möglich: Cluster-Intern, Cluster-Cluster. Hier: Cluster-Intern!
18 Primary eines RepicaSets protokolliert Schreibzugriffe in Oplog für Slaves.
19 Optimistische, Asynchrone Replizierung mit Kollissionserkennung und deterministischer Konfliktlösung.
20 Fullsync oder Realtime. Mit Failover.
21 Bei Schreibzugriffen auf dem Slave, synchronisiert Slave mit Master und sperrt sowohl Master als auch Slave.
22 Ist Implementiert, wird nicht verwendet! Kann für eigene Erweiterungen verwendet werden.
23 Externe Software/Systeme werden z.B. für Indizierung verwendet und sind durch ein Zwei-Phasen-Commit Protokoll angebunden.
24 Ohne Rollback und nur Knoten-intern (auf Master)

25 Benötigt separate Synchronisierung der Uhrzeit, z.B. über NTP. Kollissionen bei gleichem Zeitstempel: lexikalisch größerer Key gewinnt.
26 Versionen werden keine festgehalten, allerdings hat der Datensatz eine Sequenznummer (entspricht Anzahl Mutationen, also Updates)
27 Client bekommt zwar nach r Nodes schon Rückmeldung, Proxy-Node wartet dennoch alle Antworten ab um Fehler festzustellen.
28 System nimmt neuesten Eintrag ODER gibt Anwendung alle kollidierten Einträge zurück
29 Konflikte werden deterministisch gelöst aber mit einem Flag gekennzeichnet. Anwendung kann anhand des Flags entscheiden, ob Konflikt anders gelöst werden soll.
30 Abhängig von gewähltem Quorum
31 Master-Knoten im Single-Master-Fall bzw. der Knoten, mit dem die Anwendung im Augenblick verbunden ist.
32 Innerhalb eines Knotens sind die Daten aufgrund des sequentiellen append-only Logs Sequenziell Konsistenz.
33 Abhängig von Durable-Write-Quorum
34 Alle weiteren Knoten im Cluster, außer dem Ziel-Knoten der momentan betrachteten Anwendung.
35 Abhängig von Quoren
36 Gelesen wird, wie auch geschrieben, immer vom selben Knoten. Innerhalb eines Clusters hat nur ein Knoten aktiv die Daten.
37 Anwendung kommuniziert über Application Nodes. Daten sind auf Data Nodes verteilt gespeichert, Verteilung wird von Management Nodes organisiert.
38 Streng genommen (!) ist für ein vBucket ein Knoten Master und mehrere Slave. Fällt der Master aus, wird einer der Slaves zum Master.
39 Über Drittsoftware (Apache Zookeeper) oder Graphenpartitionierung
40 Hashing über Primarykey auf Daten-Knoten
41 Über sog. Regions.
42 Auf Clientseite kann Partitionierung nach Bereichen, Hashing oder Consistent Hashing erfolgen.
43 Innerhalb einer Collection muss ein indiziertes Feld definiert werden, auf Grundlage dessen Inhalts ein Datensatz entweder Range-Based oder über Hashing einem Cluster zugeordnet wird.
44 Schreibzugriffe an Slaves werden an den Master synchronisiert.
45 Treiber übernehmen üblicherweise diese Aufgabe. Java-Treiber kann das wohl.
46 Einige Treiber unterstützen einen Failover, allerdings nicht für alle Programmiersprachen.
47 Üblicherweise Client-basiertes Hashing mit mehreren Servern. Alternativen mit zusätzlicher Software: a) Proxy-gestütztes Hashing b) Query-Routing (Umleiten des Clients)
48 Abh. vom Typ des Knotens. Datenknoten werden automatisch verteilt, Management- und Anwendungs-Knoten müssen manuell ausreichend viele zur Verfügung stehen.
49 Fällt Slave weg kein Problem. Fällt Master weg, muss manuell Slave zu Master gemacht werden. Abhängig von Partitionierung ist Rückskalierung anwendungsabhängig evtl. manuell möglich.
50 Replikation: Knotenausfall (evtl. neuer Primary) möglich, Sharding: Rebalancieren im Hintergrund
51 Unabh. voneinander Konfigurierbar: 1) RDB: Snapshot-File 2) AOF: Write-Ahead Log Ist beides aktiviert, ist Dauerhaftigkeit wie bei RDBMS

		Relationale	Relationale	Relationale	Spalten-orientierte	Spalten-orientierte	Key-Value	Key-Value	Key-Value/Dokum.	Dokum.-or.	Dokum.-or.	Graphen-orientierte
		PostgreSQL[1]	**MySQL[2]**	**MySQL Cluster**	**HBase**	**Cassandra**	**Riak**	**Redis**	**Couchbase[3]**	**MongoDB**	**CouchDB**	**Neo4J[4]**
Allgemein												
	Version	9.3	5.7	7.3	0.94	2.02	1.4.2	2.6.16	2.2	2.4	1.4	2.0.0
	Dateneinteilung	Datenbank > Tabelle	Datenbank > Tabelle	Datenbank > Tabelle	Tabelle > Spaltenfamilie	Tabelle	Buckets	-[5]	Buckets	Datenbank > Collection	Datenbank	Knoten
	Gruppeneinteilung	SML	SML	RCL	SMP	MMC	MMC	SMP	MMC	SMP	MMO	SML
Replizierung												
	Konf. Granularität	Datenbank	Datenbank	Komplett	Komplett	Komplett	Komplett	Komplett	Komplett	Collection[6]	Datenbank	Komplett
Architektur												
	Allgemein	Single-Master	Single-Master	Multi-Master	Single-Master	Multi-Master	Multi-Master	Single-Master	Multi-Master[7]	Single-Master[8]	Multi-Master	Single-Master
	Master Failover	Hot-Standby[9]	Hot-Standby[10]	Hot-Standby	Hot-Standby	Hot-Standby	Hot-Standby	Warm-Standby	Warm-Standby	Warm-Standby	Cold-Standby	Hot-Standby[11]
	Master Wahl	konfigurierbar	manuell	automatisch	automatisch	alle Master	alle Master	manuell	automatisch	automatisch	alle Master	automatisch
	Verteilung Granularität	vollst. repl.	vollst. repl.	partielle repl.	vollst. repl.	partielle repl.	partielle repl.	vollst. repl.	partielle repl.	vollst. repl.[12]	vollst. repl.	vollst. repl.
Knoten Aktualis.												
	Synchronität	asynchron[13]	asynchron	synchron[14]	asynchron	synchron[15]	synchron[15]	asynchron[16]	asynchron[17]	asynchron[18]	asynchron	synchron / async.
	Informationsbeschaffung	Poll&Hold	Poll&Hold	Push	Push	Push	Push	Poll&Hold	Push	Poll	Poll / Push	Poll / Push
	Informationsaustausch	Operationen	Operationen	Operationen	Operationen	Operationen	Neue Daten	Neue Daten	Neue Daten	Operationen	Neue Daten	Neue Daten
	Inform. Granularität	je Transaktion	je Transaktion	je Transaktion	je Operation	je Operation	je Operation	je Operation	je Operation	je Operation	je Operation	-
Cluster-Replizierung												
	Möglich	nein	nein	ja[19]	nein	ja	ja[20]	nein	ja	nein	nein	nein
Konfliktmanagement												
	Strategie	Locking	Locking	Locking	Versionierung	Versionierung	Versionierung	Locking	-	Locking	Versionierung	Locking[21]
	Umsetzung	2PC[22]	-	2PC	-	Quoren	Quoren	-	-	-	-	2PC[23]
	Transaktionen	auf Master	auf Master	verteilt	keine	keine	keine	optional[24]	keine	keine	keine	auf Master
Versionierung												
	Versionskennung	keine	keine	keine	Timestamp	Timestamp[25]	Vektor Uhren	keine	keine[26]	keine	Vektor Uhren	keine
	Konfl. Erkennung	-	-	-	async. Job	Lesezugriff[27]	async.&Lesezug.	-	-	-	async. Job	-
	Konfl. Lösung	-	-	-	-	System	konfigurierbar[28]	-	-	-	Anwendung[29]	-
Konsistenz												
	Modellbezeichnung	Master ACID sonst BASE	Master ACID sonst BASE	ACID	BASE	BASE	BASE	BASE	BASE	BASE	BASE	Master ACID sonst BASE
	CAP Modell	CA	CA	CP	CP	AP/CP[30]	AP/CP[30]	CA	CP	CP	AP	CA
	PACELC Modell	PCEC	PCEC	PCEC	PCEL	PAEL	PAEL	PAEC	PAEC	PAEC	PAEL	PCEC
Aktiver Knoten[31]												
	Datensicht	Strenge K.	Strenge K.	Strenge K.	Strenge K.	Sequentielle K.[32]	Strenge K.[33]	Strenge K.	Strenge K.	Strenge K.	Strenge K.	Strenge K.
	Clientsicht	Strenge K.	Strenge K.	Strenge K.	Strenge K.	Sequentielle K.	Strenge K.	Strenge K.	Strenge K.	Strenge K.	Strenge K.	Strenge K.
Sonstige Knoten[34]												
	Clientsicht	Eventual Consistent	Eventual Consistent	Strenge K.	Eventual Consistent	Eventual Consistent	Eventual Consistent[35]	Eventual Consistent	Read-Your-Writes [36]	Eventual Consistent	Monotonic Read	Eventual Consistent
Partitionierung												
	Dynamik	keine	keine	Automatisch[37]	Automatisch	Automatisch	Automatisch	Automatisch	Automatisch[38]	Automatisch	keine	keine[39]
	Strategie	-	-	Consistent Hashing[40]	Regions[41]	Consistent Hashing	Consistent Hashing	-[42]	Consistent Hashing	Sharding[43]	-	-
	Zugangsknoten	-	-	zentraler Knoten	zentraler Knoten	beliebiger Knoten	beliebiger Knoten	Direktzugriff	beliebiger Knoten	zentraler Knoten	-	beliebiger Knoten [44]
	Zuordnung	-	-	serverproxy	-	clientseitig[45]	clientseitig[46]	clientseitig[47]	serverproxy	-	-	clientseitig
	Rückskalierung	-	-	-[48]	automatisch	automatisch	automatisch	-[49]	automatisch	automatisch[50]	-	-
Dauerhaftigkeit												
	Speichermedium	Disk	Disk	Disk	konfigurierbar	Disk	Disk	konfigurierbar[51]	Disk[52]	Disk	Disk[53]	Disk
	Journaling	ja	ja	ja	ja	ja	-[54]	optional	nein	ja	nein[55]	ja[56]
	Garantie	Streng	Streng[57]	Streng	Streng	Streng	Quorum[58]	konfigurierbar	eventual persistent [59]	Streng[60]	Streng	Streng

52 Optional kann die Speicherung asynchron auf Disk erfolgen. Mehrere I/O-Prozesse arbeiten Queues ab.
53 append-only, auch Löschoperationen
54 Abhöngig von gewählter Storage Engine. Diese kann z.B. MySQLs InnoDB sein, welche Journaling unterstützt.
55 Nicht notwendig, da Schreiben append-only sehr schnell direkt stattfinden kann.
56 logical log file wird nach Transaktion auf Disk geschrieben. Append-only.
57 Dauerhaftigkeit hängt von gewählter StorageEngine ab (z.B. Memory nur RAM, InnoDB mit Crash Recovery, usw.)
58 Über DW (durable write) kann eine Anzahl an Knoten angegeben werden, die die Operation persistent ausgeführt haben müssen, bevor Anfrage erfolgreich ist.
59 Schreibzugriff wird nicht direkt auf Disk getätigt sondern gelangt in eine Queue, die irgendwann asynchron abgearbeitet wird.
60 Update-in-Place

Literaturverzeichnis

[1] Abadi, Daniel: *Problems with CAP, and Yahoo's little known NoSQL system*. Jan. 2014. – URL http://dbmsmusings.blogspot.de/2010/04/problems-with-cap-and-yahoos-little.html

[2] Anderson, J. C. ; Lehnardt, Jan ; Slater, Noah: *CouchDB - The Definitive Guide*. Jan. 2014. – URL http://guide.couchdb.org/editions/1/en/index.html

[3] Apache Software Foundation: *The Apache HBase Reference Guide*. Jan. 2014. – URL http://hbase.apache.org/book.html

[4] Apache Software Foundation: *Cassandra Wiki - Architecture Overview*. Jan. 2014. – URL http://wiki.apache.org/cassandra/ArchitectureOverview

[5] Basho Technologies Inc.: *Riak Docs*. Jan. 2014. – URL http://docs.basho.com/riak/latest/

[6] Brewer, Eric A.: Towards robust distributed systems. In: *PODC* (2000)

[7] Chang, Fay ; Dean, Jeffrey ; Ghemawat, Sanjay ; Hsieh, Wilson C. ; Wallach, Deborah A. ; Burrows, Mike ; Chandra, Tushar ; Fikes, Andrew ; Gruber, Robert E.: Bigtable: A Distributed Storage System for Structured Data. In: *ACM Trans. Comput. Syst.* (2008)

[8] Citrusbyte LLC.: *Redis Documentation*. Jan. 2014. – URL http://redis.io/documentation

[9] Couchbase Inc.: Couchbase Server Under the Hood. (2013). – URL http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/Couchbase_Server_Architecture_Review.pdf

[10] Couchbase Inc.: *Couchbase Manual 2.2*. Jan. 2014. – URL http://docs.couchbase.com/couchbase-manual-2.2/

[11] DataStax Inc.: *Apache Cassandra 2.0*. Jan. 2014. – URL http://www.datastax.com/documentation/cassandra/2.0/webhelp/index.html

[12] Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce: Simplified Data Processing on Large Clusters. In: *Communications of the ACM* (2008)

[13] DeCandia, Giuseppe ; Hastorun, Deniz ; Jampani, Madan ; Kakulapati, Gunavardhan ; Lakshman, Avinash ; Pilchin, Alex ; Sivasubramanian, Swaminathan ; Vosshall, Peter ; Vogels, Werner: Dynamo: Amazon's Highly Available Key-value Store. In: *SIGOPS Operating Systems Review* (2007)

[14] Domaschka, Jörg: *A Comprehensive Approach to Transparent and Flexible Replication of Java Services and Applications*, Universität Ulm, Dissertation, 2012

[15] Edlich, Stefan ; Friedland, Achim ; Hampe, Jens ; Brauer, Benjamin ; Brückner, Markus: *NoSQL, Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. Hanser, 2011

[16] Fox, Armando ; Brewer, Eric A.: Harvest, Yield, and Scalable Tolerant Systems. In: *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on* (1999)

[17] George, Lars: *HBase Architecture 101 - Write-ahead-Log*. Jan. 2014. – URL `http://www.larsgeorge.com/2010/01/hbase-architecture-101-write-ahead-log.html`

[18] Karger, David ; Lehman, Eric ; Leighton, Tom ; Panigrahy, Rina ; Levine, Matthew ; Lewin, Daniel: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. (1997)

[19] Lakshman, Avinash ; Malik, Prashant: Cassandra: a decentralized structured storage system. In: *SIGOPS Oper. Syst. Rev.* (2010)

[20] MongoDB Inc: *The MongoDB 2.4 Manual*. Jan. 2014. – URL `http://docs.mongodb.org/manual/`

[21] Neo Technology: *The Neo4j Manual v2.0.0-M05*. Jan. 2014. – URL `http://docs.neo4j.org/chunked/milestone/`

[22] Oracle: Guide to Scaling Web Databases with MySQL Cluster. In: *A MySQL White Paper* (2013)

[23] Oracle: *MySQL 5.7 Reference Manual*. Jan. 2014. – URL `http://dev.mysql.com/doc/refman/5.7/en/`

[24] Redmond, Eric ; Wilson, Jim R.: *Seven Databases in Seven Weeks*. The Pragmatic Programmers, 2012

[25] Rys, Michael: Scalable SQL. In: *Communications of the ACM* (2011)

[26] Sadalage, Pramod J. ; Fowler, Martin: *NoSQL Distilled*. Addison-Wrsley, 2013

[27] Tannenbaum, Andrew ; Van Steen, Marten: *Verteilte System - Grundlagen und Paradigmen*. PEARSON Studium, 2003

[28] The PostgreSQL Global Development Group: *PostgreSQL 9.3.2 Documentation*. Jan. 2014. – URL `http://www.postgresql.org/docs/9.3/static/index.html`

[29] Vogels, Werner: Eventually consistent. In: *Commun. ACM* (2009)

[30] Wiesmann, Matthias ; Pedone, F. ; Schiper, A. ; Kemme, B. ; Alonso, G.: Understanding Replication in Databases and Distributed Systems. In: *Proceedings. 20th International Conference* (2000)

[31] Wiesmann, Matthias ; Pedonet, Fernando ; Schiper, Andre ; Kemmet, Bettina ; Alonso, Gustavo: Database replication techniques: a three parameter classification. In: *SRDS-2000. Proceedings The 19th IEEE Symposium* (2000)

[32] Wiesmann, Matthias ; Schiper, Andre: Comparision of Database Replication Techniques Based on Total Order Broadcast. In: *Knowledge and Data Engineering, IEEE Transactions* (April 2005)

Erklärung

Ich, Christopher B. Hauser, Matrikelnummer 647768, erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Christopher B. Hauser