BACHELORARBEIT

zur Erlangung des akademischen Grades

„Bachelor of Science in Engineering“ im Studiengang BIC5

Vergleich von NoSQL Datenbanken hinsichtlich Performance, Consistency und Durability

Ausgeführt von: Markus Hösel
Personenkennzeichen: 1210258001

1. BegutachterIn: Priv. Doz. Dipl.-Ing. Dr.techn. Karl M. Göschka

Wien, 14. Jänner 2015

FACHHOCHSCHULE
TECHNIKUM WIEN

Eidesstattliche Erklärung

„Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht.“

Kurzfassung

Vorliegende Arbeit untersucht die drei DBMS MongoDB, Couchbase und CouchDB, welche im Bereich der Document Stores als die populärsten gelten. Diese DBMS werden in einer Testumgebung unter verschiedenen Szenarien auf ihr Verhalten untersucht. Die gewonnenen Ergebnisse werden gegenübergestellt und liefern dabei wertvolle Erkenntnisse über mögliche Anwendungswzecke.
Die Arbeit beschreibt zuerst die Eigenschaften Volume, Velocity, Variability und Agility. Diese können von traditionellen RDBMS nur sehr schlecht erfüllt werden und gaben den Anstoß für die Entwicklung von NoSQL DBMS. Da diese Eigenschaften für verschiedene Business Cases in unterschiedlicher Ausprägung auftreten, haben sich vier verschiedene Unterkategorien etabliert - die Key Value Stores, Column Family Stores, Graph Databases und Document Stores. Der Fokus dieser Arbeit liegt im weiteren Verlauf auf den Document Stores, allerdings lassen sich viele der beschriebenen Definitionen auch auf die anderen Unterkategorien übertragen.
Damit NoSQL DBMS die in obrigen Absatz genannten Eigenschaften implementieren können, wird eine Scale-Out Architektur verwendet. Dabei besteht die Datenbank aus einem logischen Zusammenschluss mehrerer physischer Server (aus zumeist herkömmlicher Hardware). Basierend auf dem CAP Theorem muss jedoch in verteilten Systemen bei Entstehung von Netzwerkpartitionen eine Entscheidung zwischen Verfügbarkeit und Konsistenz getroffen werden. Das Theorem wird in PACELC erweitert welches beschreibt, dass bei Abwesenheit von Netzwerkpartitionen weiters über einen Trade-Off zwischen Performance und Konsistenz entschieden werden muss. Aus den notwendigen Designentscheidungen entstand das Akronym BASE, welches das Pendant zu dem aus der relationalen Datenbankwelt bekannten ACID Begriff ist und den meisten NoSQL DBMS zugrunde liegt.
Die Arbeit geht in mehreren Tests, für welche eine in Java programmierte Software verwendet wird, auf diese Trade-Offs ein und untersucht dabei das Performance-, Konsistenz-, und Durabilityverhalten. Die DBMS werden dabei unterschiedlichen Szenarien unterzogen und es zeigt sich, dass bei allen Tests verschiedene Performanceverhalten und Inkonsistenzen zwischen den einzelnen Servern gemessen werden. Dabei weisen MongoDB und Couchbase ähnliche Werte auf und CouchDB schneidet verhätnismäßig schlecht ab. Beim Simulieren eines Ausfalls des Master Servers kommt es bei allen drei Datenbanken zum Datenverlust. Auch hier ist der Datenverlust bei CouchDB wesentlich größer als jener bei MongoDB und Couchbase.
Das Konfliktmanagement der CouchDB, welches aufgrund der Architektur notwendig ist, zeichnet sich durch einen zweistufigen Prozess zur Ermittlung einer „Winning Revision“ aus. Um den Konflikt jedoch zu lösen muss seitens des Clients agiert werden.

Schlagwörter: DBMS, relationale Datenbanken, NoSQL, Document Stores, Performance, Dauerhaftigkeit, Konsistenz

Abstract

This paper analyzes three DBMS, notably MongoDB, Couchbase and CouchDB. These databases are currently the most popular document stores. The DBMS are installed in an test environment and the behavior during different scenarios is measured. The results, which provide useful knowledge for possible application purposes, are compared to each other.
The paper first explains the following characteristics: volume, velocity, variability and agility. Traditionel RDBMS don't support these characteristics very well, and therefor the developing of new systems started. Most of them are summarized as NoSQL Systems and categorized into one of four sub-categories: key-value stores, column-family stores, graph database and document stores. This paper focuses on document stores, but most of the described definitions do also fit the other sub-categories.
NoSQL DBMS are based on an scale-out architecture in order to implement the mentioned characteristics. Therefor the database exists of several physical servers (mostly commodity hardware) which form a cluster. In presence of a network partition, a decision between availability and consistency needs to be made. This trade-off is defined in the CAP theorem. Furthermore PACELC also describes the trade-off between performance and consistency, which needs to be made in absence of a network partition. The necessary decisions formed the term BASE, which can be seen as a pendant to the ACID concept and which is used in most NoSQL DBMS.
In order to analyze the trade-offs a software was written in Java. The software is executed during different scenarios and the performance-, consistency-, and durability behavior during the tests is recorded accordingly. The results show different performance times and inconsistencies between all nodes in the cluster and one can see that CouchDBs results are not as good as the results from MongoDB and Couchbase, which are very similiar. When simulating an error on the master server, all databases gain data loss. The data loss by CouchDB is considerably greater compared to MongoDB or Couchbase.
The paper also shows the conflict management of CouchDB, which is provided in order to solve possible conflicts. This conflict may occur, and are caused by the architecture. In case of a conflict a two-step procedure marks one version as the "winning revision". To resolve the conflict a client needs to operate on the database.

Keywords: DBMS, relational database, NoSQL, document stores, performance, durability, consistency

Danksagung

Bedanken möchte ich mich bei meinem Betreuer, Herrn Karl M. Göschka, für seine thematisch wertvollen Ratschläge als auch für seine engagierte Unterstützung zu kritischen Fragestellungen.

Inhaltsverzeichnis

1 Einleitung

Heutige Rechenzentren müssen in vielen Bereichen mit einem rasant ansteigenden Datenwachstum klarkommen. Vor allem Unternehmen, welche Usern über das Internet weltweit Services zur Verfügung stellen und dafür jegliche Aktivitäten mitprotokollieren, als auch Unternehmen welche eine enorme Anzahl an Messdaten aufzeichnen, analysieren und auswerten, müssen sich mit dieser Problematik im Detail auseinandersetzen [5].

Um steigenden Anforderungen mit entsprechender Rechenleistung gerecht zu werden können grundsätzlich zwei verschiedene Ansätze verfolgt werden: Scaling up und Scaling out. Beim Scaling up werden existierende Maschinen durch größere Maschinen ersetzt, welche mehr Prozessorleistung, Disk Storage und Memory bieten. Ein Nachteil dieses Ansatzes sind die überproportional steigenden Kosten für größere Maschinen. Des Weiteren läuft man hier schlussendlich in eine Limitierung existierender Hardware. Der alternative Ansatz ist Scaling out. Anstatt kleinere Maschinen durch größere zu ersetzen werden hier sogenannte Cluster (logischer Verbund aus Rechnern) eingesetzt. Diese werden zumeist aus handelsüblicher Hardware (Commodity Hardware) aufgebaut. Cluster sind nicht nur günstiger als größere Maschinen, sondern auch stabiler gegenüber dem Ausfall einzelner Hardware-Komponenten. Diese Eigenschaft stellt vor allem für die Hochverfügbarkeit eines Systems ein wichtiges Kriterium dar [2].

Mit dem Einzug des Clusters haben sich jedoch neue Herausforderung, vor allem beim Betreiben von Datenbanken, ergeben. Die seit den 70er Jahren eingesetzten relationalen Datenbanken sind nicht dafür entworfen über Rechnergrenzen hinweg betrieben zu werden. Zwar gibt es einige Ansätze, wie Oracle RAC oder Microsoft SQL Server, welche clusterfähig sind, jedoch basieren deren Konzepte auf der Basis eines Shared Disk Subsystem. Damit bleibt im Cluster ein Single Point of Failure erhalten. Ein anderer Ansatz ist das Aufteilen der Daten in sogenannte Shards, welche jeweils auf einen anderen Server beheimatet sind. Die Vorgehensweise hierfür wird als Sharding bezeichnet. Damit ergeben sich aber eine Reihe an neuen Problemen. Queries, Transaktionen und Consistency Control können zumeist nicht mehr unterstützt oder nur unter Kaufnahme von erhöhter Latency bewerkstelligt werden [1].

Diese Diskrepanz zwischen relationalen Datenbanken und Clusters haben in den letzten Jahren alternative Konzepte entstehen lassen.

Eine Ansammlung dieser Konzepte wird unter dem Namen NoSQL (Not Only SQL) geführt. Der Fokus liegt hierbei auf der Skalierbarkeit über mehrere Rechner, wobei herkömmliche Anforderungen an Datenbanken, wie die Unterstüztung der ACID Kriterien (Atomicity, Consistency, Isolation, Durability) gelockert und neue Ideen umgesetzt werden. Vorliegende Arbeit setzt sich mit den Document Stores auseinander, welche eine Unterkategorie der NoSQL Datenbanken darstellen. Dabei wurden basierend auf der Webseite [48], welche über verschiedene Berechnungsmethoden die Popularität von Datenbanken in einem Ranking listet, die drei führenden Document Stores ausgewählt. Diese Datenbankmanagementsysteme (DBMS) werden hinsichtlich ihrer Performance und Datenkonsistenz in Verbindung mit verschiedenen Fehlerszenarien und administrativ

gewollten Änderungen der Clusterarchitektur untersucht und die Ergebnisse sowohl grafisch als auch in geeigneter Tabellenform aufbereitet. Tabelle 1 listet die ausgewählten DBMS mit der bei den Tests verwendeten Versionen und den clientseitig verwendeten Java Driver Versionen auf.

DBMS	Version	Java Driver Version
MongoDB	2.6.4	2.10.1
Couchbase	2.2.0	1.4.4
CouchDB	1.0.1	- (REST Interface)

Tabelle 1: Liste der verwendeten DBMS mit Version und clientseitiger Java Driver Version.

1.1 Problem- und Aufgabenstellung

Diese Arbeit vergleicht die in Tabelle 1 angeführten DBMS hinsichtlich Performance, Consistency und Durability. Dazu wird eine Testumgebung aufgebaut, die aus bis zu vier virtuellen Servern besteht, welche zu einem Cluster zusammengeschalten werden. Mithilfe eines clientseitig ausgeführten Programms wird der Cluster verschiedenen Messungen unterzogen. Dabei werden verschiedene Szenarien simuliert, mit der sich Administratoren und Programmierer im produktiven Betrieb auseinandersetzen müssen. Es werden einerseits die Auswirkungen einer gewollten Erweiterung und Reduktion der Serveranzahl im Cluster betrachtet, andererseits widmet sich diese Arbeit aber auch Fehlersituationen mit denen Aufgrund statistischer Wahrscheinlichkeiten im produktiven Betrieb gerechnet werden müssen. Untersucht wird dabei der plötzliche Ausfall eines Replika Server oder auch der Ausfall eines Master Servers. Es wird dabei festgehalten, wie sich die Antwortszeit des Systems ändert und in welchem Maß sich hierdurch Consistency und Durability verändern. Die durchgeführten Tests sollen dabei Gemeinsamkeiten und Unterschiede der unter betracht gezogenen DBMS aufzeigen und gegenüberstellen.

1.2 Inhalt und Aufbau der Arbeit

Die Arbeit beschreibt im zweiten Kapitel den theoretischen Hintergrund, welcher sowohl die Gründe für den Einsatz von NoSQL DBMS als auch die dahinterstehenden Konzepte erläutert. Im dritten Kapitel werden die einzelnen DBMS beschrieben, wobei auf die jeweilige Architektur, als auch auf Trade-Offs und die Clientanbindung eingegangen wird. Anschließend wird im vierten Kapitel die Teststellung und das Setup der DBMS spezifiziert. Das fünfte Kapitel präsentiert die Ergebnisse der durchgeführten Tests, und geht auf eine Gegenüberstellung der Messergebnisse zwischen den einzelnen DBMS ein. Ähnliche wissenschaftliche Arbeiten werden im Kapitel sechs dargelegt. Abschließend werden im siebenten Kapitel die wichtigsten Aussagen zusammengefasst und ein Ausblick gegeben.

2 NoSQL Datenbanken

Relationale Datenbanken waren in den vergangenen Jahrzehnten die Lösung für beinahe alle Datenbankanforderungen und haben ihren Ursprung in den 70er Jahren. Der Erfolg dieser Datenbanken ist einerseits auf eine weite Verbreitung zurückzuführen, andererseits wurden diese Datenbanken über Jahre ständig weiterentwickelt und optimiert und konnten den Anforderungen meist gerecht werden [7].
Die Daten lassen sich dabei als Tabellen repräsentieren und setzen sich aus einer beliebigen Anzahl aus Zeilen und Spalten zusammen. Jede Zeile steht dabei für einen Datensatz. Die Datensätze können miteinander verknüpft werden. Für die Manipulation der Daten wird dabei SQL (Structured Query Language) als einheitliche Datenmanipulationssprache verwendet.
Mit dem Anbruch des Web 2.0 und dem damit verbundenen Datenwachstum ergaben sich jedoch Anforderungen an Datenbanken, welche mit herkömmlichen Relationalen DBMS nicht effizient abgedeckt werden können. Die vielfältige Beschaffenheit unstrukturierter Daten als auch die benötigte Flexibilität hinsichtlich Scaling, Design, Kosten und Disaster Recovery lies daher Ende der 90er Jahre neue Systeme entstehen, welche unter dem Namen NoSQL zusammengefasst werden [14].

Folgende Faktoren ließen Ende der 90er Jahre neue Datenbank Systeme entstehen:

- Anforderung große Datenvolumen von strukturierten, semi-strukturierten und unstrukturierten Daten zu verwalten.
- Effiziente Scale-Out Architekturen anstatt monolithischer Scale-Up Architektur.
- Verzicht auf Relationenmodell und auf Beziehungen zwischen einzelnen Datensätzen.
- Erhöhte Durchsatzanforderungen.
- Flexible Verwendung in Objekt-Orientierten Programmiersprachen anstatt Anwendung von komplexen Objekt-Relationalen Mapping Layern.
- Schemafreiheit und agile Möglichkeiten neue oder sich veränderte Datenstrukturen effizient abzubilden.

2.1 Business Drivers

Die neuen Anforderungen an DBMS wurden in [2] durch vier Business Drivers (Volume, Velocity, Variability und Agility) beschrieben. Abbildung 1 veranschaulicht sinnbildlich die Einwirkung dieser Größen auf Relationale DBMS. Die Risse symbolisieren dabei die Einwirkung der hohen Anforderungen neuer Anwendungen, welche herkömmliche RDBMS nicht gerecht werden können. Der Grund hierfür liegt den bei relationalen Datenbanken verfolgten Scale-Up Ansatz, welcher ab einer bestimmten Größe des Systems nicht mehr vertreten werden kann. Die vier Business Driver werden dabei wie folgt beschrieben:

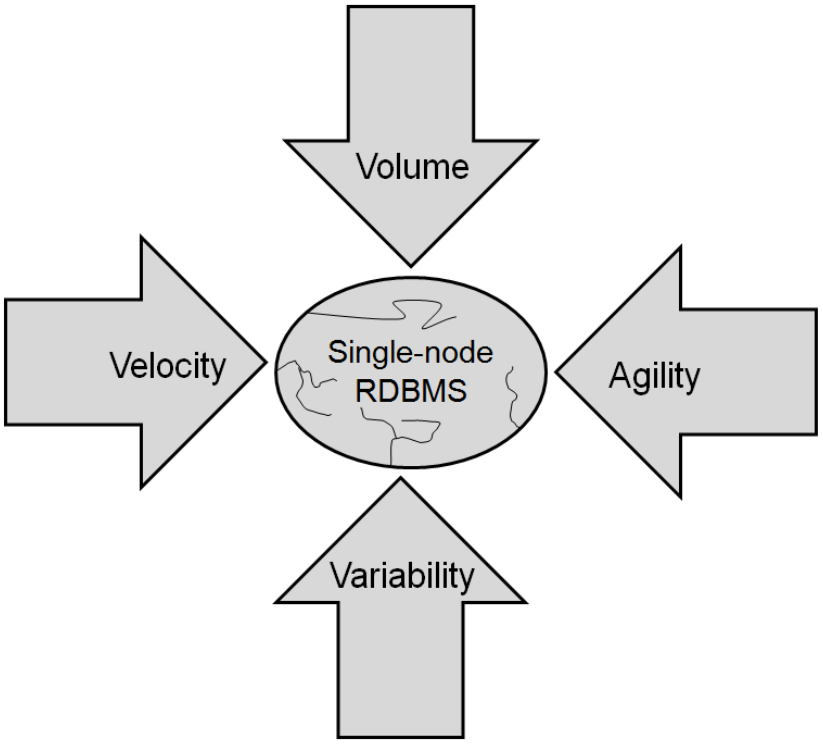


Abbildung 1: NoSQL Business Drivers [2]

- Volume: Der Hauptgrund schlechthin weshalb Organisationen nach Alternativen der bisher im Einsatz stehenden RDBMS suchen, ist die Möglichkeit rießige Datenmengen im Tera- und Petabyte Bereich effizient und kostengünstig mittels Clusters aus Commodity Hardware zu verwalten. Die Forderung Daten parallel anstatt seriel zu verarbeiten brachte Organisationen die Notwendigkeit horizontal anstatt vertikal zu skalieren, um somit die Arbeit über mehrere Prozessoren, Hauptspeichern und Festplatten über Rechnergrenzen hinweg aufzuteilen.
- Velocity: High-Velocity und Real-Time Anforderungen, in welchen Millionen von Datensätzen pro Sekunde gelesen und geschrieben werden, machten es ebenso erforderlich Systeme horizontal zu skalieren, um damit den wachsenden Anforderungen an Durchsatz gerecht zu werden.
- Variability: RDBMS fordern das Definieren eines Schemas in welchen die einzelnen Datensätze abgebildet werden. Für semi- und unstrukturierte Daten sind die Verwendung von relationalen Schemas jedoch nicht zweckmäßig. Viele der definierten Felder bleiben dabei einfach frei und werden mit „null“ aufgefüllt. Das Hinzufügen von zusätzlichen Attributen erfordert weiters eine Schemaänderung - ein Prozess welcher in vielen Fällen sehr kostspielig und zeitaufwendig ist und die Verfügbarkeit des Systems einschränken kann. NoSQL Systeme verfolgen daher schemafreie Modelle, in welchen sowohl semi- als auch unstrukturierte Daten effizient gespeichert werden können.
- Agility: Bezeichnet die Möglichkeit, schnell und effektiv neue oder sich ändernde Anforderungen bezüglich der Abfrage und Speicherung von Daten in den Applikationen abzubilden. Dies ist bei RDBMS oft aufgrund des Objekt-Relationalen

Mapping Layers problematisch. Die korrekte Implementierung der SQL-Operationen ist vorallem bei verschachtelten Daten komplex und stellt eine große Schranke für die rasche Adaptierung der Systeme dar.

NoSQL Datenbanken unterliegen keiner einheitlichen Definition. Jedoch gibt es eine Anzahl an Kriterien, welche eine Datenbank als NoSQL System kennzeichnen. NoSQL Datenbanken verwenden kein relationales Datenmodell, sondern sind zumeist schemafrei oder haben nur schwache Schemarestriktionen. Die Systeme verfügen über keine einheitliche Abfragesprache, wie es bei relationalen Datenbanken durch SQL der Fall ist, sondern implementieren APIs die entweder über eine REST-Schnittstelle ansprechbar sind oder ein proprietäres Protokoll einsetzen [7].

2.2 Gruppen

NoSQL Systeme werden grob in vier Gruppen eingeteilt: Key Value Stores, Column Family Stores, Document Stores und Graph Databases [1].

2.2.1 Key Value Stores

Key Value Stores verhalten sich ähnlich wie in Programmiersprachen zur Anwendung kommende Maps und Dictionaries. Die Daten werden als Byte Arrays abgebildet und über einen eindeutigen Identifier (Primary Key) angesprochen. Abfragesprache existiert bei diesen Datenbanken keine, womit Beziehungen zwischen den Daten auf Applikationsebene oder mittels server-seitiger Middleware abgebildet werden müssen. Aufgrund der einfachen Struktur sind Key Value Stores schemafrei. Die Daten umfassen sowohl einfache als auch verschachtelte Strings sowie Bilder und Audiodateien, welche zur Laufzeit ohne Modellierung der Datenbank hinzugefügt werden können [6].
Diese Datenbanken kommen unter anderem in Image Stores, Key-Based Filesystems oder Object Caches zum Einsatz [2]. Populäre Vertreter sind Memcache, Riak und DynamoDB.

2.2.2 Column Family Stores

Bei diesem Typ von Datenbank werden Spalten in sogenannte Column Families gruppiert. Jeder Datensatz besitzt dabei nur jene Spalten, welche seinem Datenmodell entsprechen. Die einzelnen Zeilen werden über einen eindeutigen Identifier angesprochen. Das Gruppieren der Spalten zu Column Families erlaubt die Implementierung eines ausgeklügelten Storage Mechanismus, wodurch es möglich wird Milliarden an Datensätzen abzubilden. Da jeder Datensatz nur die Spalten verwendet, welche seinen Datenmodell entsprechen, können auf intelligente Weise sowohl semi- als auch unstrukturierte Daten gespeichert werden. Verwendet werden Column Family Stores für Web Crawling, Fraud Detection und Messaging. Das Konzept wurde maßgeblich durch Googles BigTable inspiriert [8].

2.2.3 Graph Databases

Graph Databases werden bei beziehungsintensiven Daten eingesetzt. Diese Datenbanken sind in erster Linie dann die richtige Wahl, wenn die Daten auch auf einem Whiteboard in Form eines zusammenhängenden Graphen mit Knoten und Verbindungen modelliert werden können. Der Performancevorteil dieser Datenbanken ist bereits bei sehr einfachen Abfragen merkbar, und wird durch das ersetzen von rekursiver Joins durch effizientes Traversieren und unter Verwendung von Graphenalgorithmen erzielt [5].
Location Based Services, Path Finding Problems wie sie bei Navigationsgeräten vorhanden sind und Social Network Queries sind Anwendungsgebiete für Graph Databases [10]. Bekanntester Vertreter ist Neo4j.

2.2.4 Document Stores

Document Stores setzen auf dem Prinzip des Key Value Stores auf, fordern jedoch die Abbildung der Daten in einem für die Datenbank interpretierbaren Format. Damit besteht bei dieser Art von Datenbank die Möglichkeit, Queries (Abfragen) an den Daten vorzunehmen. Die am Markt befindlichen Datenbanken arbeiten dabei zumeist mit dem JSON (JavaScript Object Notation) Format. Aber auch das XML- (Extensible Markup Language) oder YAML-Format (YAML Ain't Markup Language) kommen in bestimmten Document Stores zur Anwendung [7].
Document Stores sind schemafrei und die Formate erlauben eine beliebige Verschachtelung der Daten. Damit können neue Dokumente einfach gespeichert werden, ohne im vorhinein ein Schema definieren zu müssen. Genauso ist es zur Laufzeit möglich, existierende Dokumente mit neuen Attributen upzudaten oder bestehende zu verändern und zu entfernen [6].
Verwendet werden Document Stores zumeist bei Daten, welche verschachtelt abgebildet werden können. Beispiele sind Office Dokumente, Sales Orders, Web Pages, Blogeinträge oder Product Descriptions. Weitere Einsatzszenarien sind CMS (Content Management Systeme), zentrales Logging und Real-Time Analysen [2].
MongoDB, Couchbase und CouchDB sind in dieser Kategorie die populärsten Datenbanken, und sind jene welche in dieser Arbeit untersucht werden. Viele der ausgearbeiteten Konzepte können jedoch auch auf die in diesem Kapitel erwähnten NoSQL Datenbanktypen übertragen werden, und wurden beschrieben um ein vollständiges und abgerundetes Bild zu diesem Thema geben zu können.

2.3 Eigenschaften von NoSQL DBMS

Der Fokus von NoSQL DBMS liegt vor allem auf der **Verfügbarkeit** (Availability), der **Zuverlässigkeit** (Reliability) und **Skalierbarkeit** (Scalability). Die Zuverlässigkeit beschreibt dabei die korrekte Ausführung von Operationen für die Dauer eines bestimmten Intervalls unter spezifizierten Bedingungen, während Verfügbarkeit die Erreichbarkeit eines Systems zu einem bestimmten Zeitpunkt definiert [19]. Als Beispiel sei genannt, dass ein System welches für eine Millisekunde pro Stunde nicht erreichbar ist eine Verfügbarkeit von 99,9999% aufweist und somit hochverfügbar, jedoch höchst unzuverlässig ist [20].
Skalierbarkeit definiert die Fähigkeit eines verteilten Systems, sich ändernder Lastverhalten anzupassen. Dies wird vor allem durch die horizontale Skalierung (Scale-Out Ansatz) erreicht. Dabei werden dem System zusätzliche Knoten hinzugefügt oder entfernt umso die Leistung des Gesamtsystems linear erhöhen oder vermindern zu können [20].
Dabei werden zwei Verteilungsmodelle unterschieden: Sharding (Datenpartitionierung) und Replizierung:

- Sharding: Daten werden über mehrere Knoten verteilt, um die Lese- und Schreibgeschwindigkeit zu erhöhen. Die meisten NoSQL Systeme unterstützen dabei das sogenannte Auto-Sharding, bei welchem nicht der Entwickler oder Administrator sondern das System als solches die Verantwortung übernimmt die Daten intelligent zu verteilen. In Kontrast zu relationalen Datenbanken, bei welchen Datensätze über mehrere Server hinweg konsistent gehalten werden müssen, wird bei NoSQL Systemen mit Aggregaten gearbeitet. Dabei wird ein Datensatz nicht über mehrere Server verteilt, sondern nur auf einem Server geschrieben. Damit sind bei Ausfall eines Servers nur jene Datensätze nicht abrufbar, die auf diesem beheimatet sind. Alle anderen Datensätze bleiben dabei konsistent und verfügbar. Um den Ausfall der Datensätze entgegen zu wirken, welche bei einem Ausfall eines Server betroffen sind, wird zusätzlich Replizierung verwendet.
- Replizierung: Daten eines Knoten werden auf einen oder mehreren zusätzlichen Knoten der verteilten Datenbank repliziert. Dadurch erhöht sich die Lesegeschwindigkeit, da Clients nun denselben Datensatz von mehreren Knoten lesen können. Weiters wird die Verfügbarkeit und Zuverlässigkeit gesteigert. Das Replizieren der Daten geht jedoch zulasten der Konsistenz. Vor allem bei einem hohen Schreib-/Leseverhalten müssen permanent Daten zwischen den beteiligten Knoten repliziert werden. In der Zeit, welche für die vollständige Replizierung von einem Datensatz benötigt wird, ist das System inkonsistent. Des Weiteren entstehen bei Netzwerkfehlern Partitionen, welche die Kommunikation zwischen den einzelnen Knoten verhindert. Dies hat zur Folge, dass die Daten des verteilten Systems nicht mehr synchronisiert werden können und Konsistenz nicht mehr garantiert werden kann. Kapitel 2.5 behandelt diese Fälle sorgfältig.

Elastizität definiert weiters die Fähigkeit eines verteilten Systems während der Laufzeit, also ohne Unterbrechung, zu skalieren und ist für NoSQL DBMS unerlässlich um den Anforderungen an Verfügbarkeit gerecht werden zu können.

2.4 ACID vs BASE

Relationale DBMS unterstützen Transaktionskontrolle durch atomare, konsistente, isolierte und dauerhafte Eigenschaften, welche unter dem Akronym ACID zusammengefasst werden. Die Eigenschaften stehen dabei für:

- Atomicity: Transaktionen werden entweder vollständig ausgeführt oder es erfolgt im Fehlerfall ein Rollback der bisher getätigten Aktionen, wodurch die Datenbank unverändert bleibt. Dies wird auch als All-or-Nothing Prinzip bezeichnet. Systeme müssen diese Eigenschaft in jeder Situation erfüllen können, auch unter Fehlerszenarien wie Power Failures, Fehlern in Software und Systemabstürzen.
- Consistency: Diese Eigenschaft garantiert den Übergang von einem konsistenten Zustand zu den nächsten. Zur Erfüllung müssen Transaktionen gültig gegenüber allen definierten Regeln wie Constraints, Cascades, Triggers und Kombinationen dessen sein, ansonsten erfolgt ein Rollback der Transaktion.
- Isolation: Gleichzeitig ausgeführte Transaktionen beeinflussen sich nicht gegenseitig. Es ist einer Transaktion somit nicht möglich Daten einer parallel laufenden Transaktion zu lesen, solange diese selbst nicht vollständig durchgeführt und mit einem Commit erfolgreich beendet wurde. Der Zustand des Systems ist daher bei parallel ausgeführten aber voneinander isolierten Transaktionen gleich dem Zustand der erreicht wird, wenn die Transaktionen seriell ausgeführt werden.
- Durability: Das Ergebnis einer erfolgreich abgeschlossenen Transaktion bleibt dauerhaft in der Datenbank gespeichert. Die dauerhafte Speicherung muss auch bei Systemfehlern garantiert sein.

ACID-Systeme fokussieren Konsistenz und Integrität der Daten gegenüber allen anderen Gesichtspunkten. Obwohl die Implementierung komplex ist, gibt es erfolgreiche Strategien um die geforderten Eigenschaften zu realisieren. Der Grundgedanke beruht dabei auf das Locking und Unlocking von Ressourcen, um während der Operationen den Zugriff von nebenläufigen Transaktionen zu verhindern [13]. Des Weiteren müssen diese Systeme mögliche Systemfehler eines Computersystems zu jedem beliebigen Zeitpunkt berücksichtigen.
Die ACID Bedingungen ermöglichen Transaktionskontrolle auf soliden, über Jahre permanent weiterentwickelten Technologien und stellen den Kern der meisten relationalen DBMS dar. Ein den ACID Garantien entgegengesetzter Ansatz wird als BASE bezeichnet. BASE offeriert „weichere Konsistenz“ zugunsten von Verfügbarkeit und Performance und findet bei den meisten NoSQL DBMS seinen Einsatz. Der Fokus liegt dabei auf

Verfügbarkeit. Eingehende Requests werden dabei permanent verarbeitet, auch unter dem Risiko hierfür kurze Zeit nicht konsistent zu sein. Der verfolgte Ansatz, anzunehmen, dass alle Partitionen zu einem bestimmten Zeitpunkt in der Zukunft konsistent werden, wird auch als optimistisch bezeichnet. Dieser Ansatz erlaubt Architekturen, auf welchen einfachere und schnellere Systeme entworfen werden können. Ziel ist die Entgegennahme und Beantwortung eingehender Requests, selbst in Gegenwart von Systemfehlern, um somit eine permanente Verfügbarkeit der Datenbank erreichen zu können.

BASE wird dabei beschrieben als:

- Basically Available: Auch wenn sich das System in einen inkonsistenten Zustand befindet werden eingehende Requests abgearbeitet und beantwortet. Die Informationen sind daher „grundsätzlich verfügbar“ - auch wenn die Möglichkeit besteht, dass sich diese zum Zeitpunkt der Abarbeitung in einen inkonsistenten oder sich verändernden Zustand befinden.
- Soft State: Die Daten unterliegen auch dann Veränderungen, wenn keine Operationen auf der Datenbank ausgeführt werden. Es besteht keine Garantie über die Aktualität der Daten zum Zeitpunkt einer Abfrage. Dieser Zustand wird als „soft“ bezeichnet.
- Eventual Consistency: Besagt, dass alle Daten aller beteiligten Partitionen zu einem bestimmten Zeitpunkt in der Zukunft in einem konsistenten Zustand sein werden. Dieser Zustand wird dann erreicht, wenn die Datenbank für ein bestimmtes, jedoch nicht konkret nennbares Zeitintervall keinen Änderungen unterliegt. Anders formuliert bedeutet die Aussage, dass keine Inkonsistenzen im stabilen Zustand dauerhaft bestehen bleiben.

Wichtig anzumerken ist, dass Systemdesigner und Entwickler sich bei der Implementierung auf einem Kontinuum zwischen den beiden Konzepten bewegen können. Es ist durchaus möglich, ACID Transaktionen an einigen Stellen zu fordern und diese an anderen Stellen zugunsten von Performance und Verfügbarkeit des Systems zu lockern [14].
Die Forderung an Datenbanken, ihre ACID Kriterien zu lockern kommt vor allem von verteilten Datenbanken. Diesen Datenbanken liegt das CAP Theorem zugrunde, welches den Gegensatz von drei essentiellen Systemanforderungen für das Design, Implementierung und Deployment von Applikationen in verteilten Systemen beschreibt. Die drei Eigenschaften lauten Consistency, Availability und Partition Tolerance und werden im folgendem Kapitel beschrieben.

2.5 CAP und PACELC

Das CAP Theorem wurde erstmals im Jahr 2000 von Eric Schmidt auf der ACM Symposium on the Principles of Distributed Computing präsentiert. Das Theorem besagt, dass es in einem verteilten System nicht möglich ist, alle drei Eigenschaften Konsistenz (Consistency), Verfügbarkeit (Availability) und Toleranz gegenüber Netzwerkpartitionen (Partition Tolerance) zum gleichen Zeitpunkt zu garantieren [15].
Die drei im Widerspruch stehenden Eigenschaften werden dabei wie folgt definiert:

- Consistency: Jeder Knoten antwortet zu jedem Request mit dem dazugehörigen Response. Vorraussetzung dafür ist, dass alle Knoten den gleichen Datenbestand haben (Replica Consistency) und dass gleichzeitige, verwobene Zugriffe auf einzelne Dateneinheiten korrekt ausgeführt werden (Concurrency Consistency). Ein Lesevorgang sieht somit das Ergebnis aller zuvor vollständig durchgeführten Schreibvorgänge.
- Availability: Jeder an das verteilte System gestellter Request wird schlussendlich mit einem Response beantwortet werden. Ein schneller Response wird gegenüber einem langsamen Response natürlich präferiert, das Theorem selbst berücksichtigt bei der Definition die Latenzzeiten jedoch nicht.
- Partition Tolerance: In Kontrast zu den ersten beiden Eigenschaften welche ein Service beschreiben, wird hier eine Aussage über die zugrunde liegende Struktur gemacht. Partitionstoleranz bezieht sich dabei auf die Toleranz gegenüber Netzwerkpartitionen. Netzwerkpartitionen entstehen bei Netzunterbrechungen und bilden dabei Servergruppen, welche untereinander nicht kommunizieren können [15].

Das starke CAP Theorem besagt nun, dass zu jedem Zeitpunkt nur zwei der drei Eigenschaften erreicht werden können. Abbildung 2 visualisiert diese Aussage als Dreieck, wobei sich die konkreten Anwendungsfälle jeweils auf einer Kante einordnen lassen.

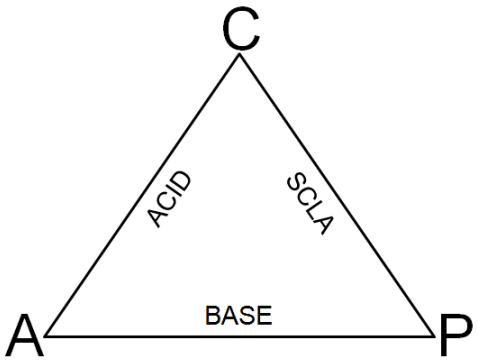


Abbildung 2: CAP Theorem [46]

Betrachtet man die Kanten, so können drei konkrete Fälle abgeleitet werden – abhängig davon welche Eigenschaften gewährleistet werden sollen.

- CP Systeme: verzichten auf hohe Verfügbarkeit in Gegenwart einer Netzwerkpartitionierung und sichern somit hohe Konsistenz. Es sind hierbei nur noch jene Knoten des Systems für Clientanfragen verfügbar, welche weiterhin trotz Netzwerkpartition absolute Konsistenz sicherstellen können. Der verfolgte Ansatz wird als SCLA (Strong Consistency, Loosely Available) definiert [16].
- AP Systeme: reagieren mit dem Verzicht vollständiger Konsistenz zugunsten der Verfügbarkeit eines verteilten Systems auf Netzwerkpartitionen. Alle Knoten (welche für die Clients von außen weiterhin erreichbar sind) reagieren auf Anfragen mit Antworten, welche kurzzeitig inkonsistent sein können. Dies wird auch als eventuelle Konsistenz bezeichnet und bildet die Grundlage für das BASE-Modell [13].
- CA Systeme: Konsistenz und Verfügbarkeit werden zugleich erreicht. Dies fordert aber die Abstinenz von Netzwerkpartitionen. Sobald eine Netzwerkpartition vorhanden ist, muss auch hier eine Priorisierung zwischen Konsistenz und Verfügbarkeit getroffen werden. Ein System als reines CA System zu klassifizieren ist somit nur in einem nicht-verteilten System möglich, in welchen Netzwerkpartitionen nicht möglich sind. Mit CA Systemen können die ACID Kriterien erfüllt werden [18].

NoSQL Datenbanken setzen auf horizontal skalierbare Architekturen und präferieren Verfügbarkeit gegenüber Konsistenz. Um hohe Verfügbarkeit zu garantieren müssen die Daten wie in Kapitel 2.3 beschrieben verteilt werden. Dies erfordert jedoch die Lockerung von Konsistenzbedingungen um Antwortszeiten kurz zu halten und damit hochverfügbar gegenüber Clients zu sein. Das System garantiert damit nur noch eine schlussendliche Konsistenz, und ist die Ursache für die Implementierung des BASE-Modells in NoSQL Datenbanken. Der Trade-Off, welcher zwischen Konsistenz und Verfügbarkeit aufgrund der Latenzzeiten durch Replizierung der Daten entsteht, wird im PACELC Rechnung getragen. Das Prinzip aus [12] besagt dabei:
"If there is a partition (P), how does the system trade off availability and consistency (A and C); else (E), when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C)?"
Es ergänzt damit das CAP Theorem um den Tradeoff zwischen Latenz und Konsistenz in Abwesenheit von Netzwerkpartitionen. Der Tradeoff zwischen Latenz und Konsistenz bezieht sich dabei nur auf die Replikation im verteiltem System. Viele der NoSQL DBMS werden von PACELC abgeleitet als PA/EL Systeme bezeichnet, da sie in Falle einer Netzwerkpartition Verfügbarkeit gegenüber Konsistenz bevorzugen und unter normalen Operationen Konsistenz für niedrigere Latenzzeiten lockern. Durch das Aufgeben strenger Konsistenz und das Anstreben eventueller Konsistenz wird, sobald festgelegt ist wie mit inkonsistenten Daten umgegangen wird, auch das Design des Systems einfacher. Des Weiteren bieten

NoSQL DBMS am Client Konfigurationsparameter an, über welche eine höhere Konsistenz zugunsten von Latency und Verfügbarkeit definiert werden kann [12].
Vorliegende Arbeit wird sich sowohl mit dem Trade-Off von Konsistenz und Latenz als auch mit dem Trade-Off von Verfügbarkeit und Konsistenz auseinandersetzen und dabei die unterschiedlichen Latenzzeiten als auch das Verhalten der Konsistenz aufzeichnen. Ebenso wird in dieser Arbeit das Verhalten der Durability (Dauerhaftigkeit) analysiert. Durability kennzeichnet die Eigenschaft, das Ergebnis von durchgeführten Operationen dauerhaft in der Datenbank bestehen zu lassen. Genauso, wie die Konsistenz gegenüber kürzeren Antwortszeiten gelockert werden kann, gibt es Anwendungsfälle in denen es vernünftig ist, die Durability zugunsten von mehr Performance zu lockern. So kann es sinnvoll sein, einen Datenbestand nur im Memory zu verwalten, jedoch mit der Gefahr bei einem Ausfall des Servers sämtliche Daten permanent zu verlieren. Ein Szenario ist das Speichern von User-Sessions. Mag es für den Kunden ärgerlich sein bei einem serverseitigen Systemfehler den aktuellen Status seiner Websession zu verlieren, so ist dies jedoch für permanent schnellere Antwortszeiten eine durchwegs vertretbare Entscheidung [1].

3 Datenbankmanagementsysteme (DBMS)

Dieses Kapitel beschreibt die Architektur, den Trade-Off zwischen Performance und Consistency/Durability als auch die Client-Anbindung zu den DBMS. Dabei werden verschiedene Konfigurationseinstellungen erläutert, um höhere Konsistenz und mehr Datensicherheit auf Kosten der Latenzzeiten zu bieten. Die Client Anbindungen werden mit Beispiel-Codes veranschaulicht, und es wird der Kern der für die Tests verwendeten Software beschrieben.

3.1 MongoDB

MongoDB ist ein schemafreier OpenSource Document Store und wird kommerziell von der Firma MongoDB, Inc. (bis 27. August 2013 unter dem Namen 10gen geführt) supported. Die Datenbank wurde entwickelt um Applikationen mit komplexen Datentypen und reichhaltigen Datenstrukturen als auch mit Anforderungen an Real-Time Abfragen mittels eines skalierbaren, performanten, kostengünstigen und permanent verfügbaren Datenspeichers abdecken zu können [23].
Obwohl die Datenbank auf keinen Relationenmodell basiert, sind zahlreiche Features von relationalen Datenbanken implementiert. Dazu gehört Sortierfunktionen, Secondary Indexes und RangeQueries. MongoDB speichert die Daten wie in Kapitel 2.2.4 beschrieben in Dokumenten ab. Die Dokumente werden als JSON Objects serialisiert, verwenden aber für die interne Speicherung binär dekodiertes JSON, welches als BSON (Binary JSON) bezeichnet wird [21].

Die Vorteile von BSON gegenüber JSON sind weniger Overhead (Lightweight), kann schneller durchsucht werden (Traversable) und ist effizient hinsichtlich Kodierung und Dekodierung von JSON zu BSON [22].
MongoDB erfüllt die ACID Kriterien auf Dokumentenebene und bietet die Möglichkeit, Schreiboperationen erst nach erfolgreichen Schreiben des Journals auf die Festplatte gegenüber der Applikation zu bestätigen, was dem gleichen Modus vieler traditioneller relationaler Datenbanken entspricht [23].
Einzelne Dokumente werden dabei in Collections verwaltet. Dies kann mit einer Relation in RDBMS gleichgesetzt werden. Jede Collection ist einer Datenbank untergeordnet. Entgegen der Definition einer Relation in RDBMS fordern Collections kein Schema. Die einzelnen Dokumente können eine beliebige Anzahl an unterschiedlichen Feldern haben. Collections beinhalten gebräuchlicher Weise eine Ansammlung an Dokumenten, welche den gleichen bzw. einen ähnlichen organisatorischen Vorsatz haben [24].

3.1.1 Architektur

MongoDB unterstützt horizontale Skalierung mittels Sharding und Replizierung. Abbildung 3 veranschaulicht die Architektur eines typischen MongoDB Clusters im Produktionsbetrieb.

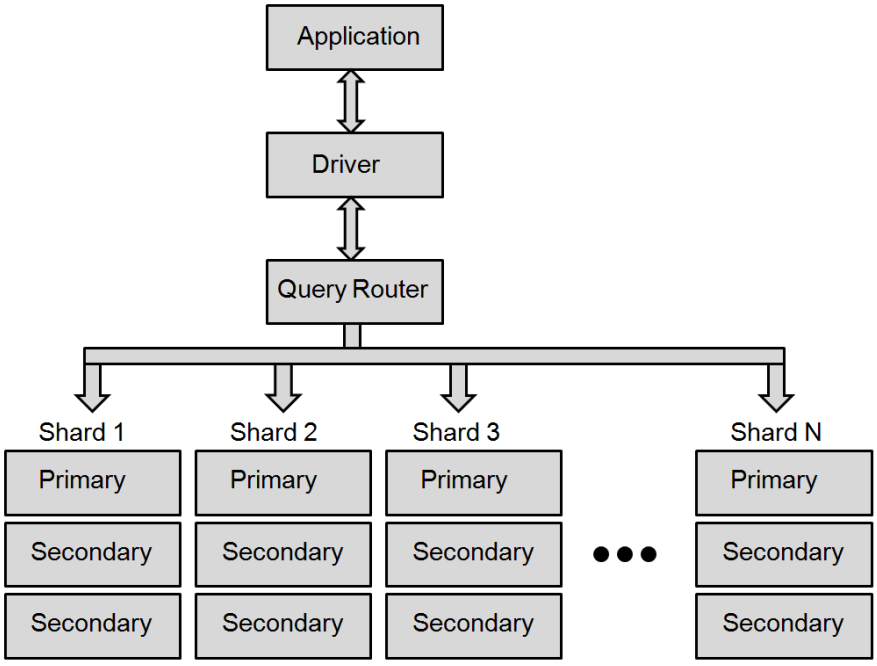


Abbildung 3: MongoDB Architektur [47]

Dabei werden die Daten im ersten Schritt über mehrere physische Partitionen verteilt. Die Anzahl an physischen Partitionen kann zur Laufzeit dynamisch erhöht und reduziert werden, wobei MongoDB im Hintergrund die Daten automatisch neu ausbalanciert. Jeder dieser Shards kann durch ein sogenanntes Replica Set aufgebaut werden. Ein Replica Set besteht dabei aus mindestens zwei Rechnern - einen Primary und einen Secondary. Je nach Anforderungen können dem Replica Set mehrere Secondaries hinzugefügt werden, jedoch

kann es zu einem Zeitpunkt nur einen Primary geben. In der Praxis hat sich das Design eines Replica Sets aus einem Primary und zwei Secondaries als sinnvoll erwiesen. Schreiboperationen werden dabei immer nur auf dem Primary durchgeführt. Abhängig vom Design der Applikation kann auch von den Secondaries gelesen werden. Kommt es zu der Situation, dass ein Primary aufgrund eines Fehlers ausfällt, so wird einer der verbliebenen Secondaries zu den neuen Primary gewählt. Damit erhöht sich die Verfügbarkeit des Clusters und Operationen können trotz defekter Rechner im Verbund weiter abgearbeitet werden. Sobald der Fehler auf dem ausgefallenen Rechner behoben ist, kann dieser als Secondary wieder dem Replica Set hinzugefügt werden. Alternativ kann der Rechner durch einen neuen ersetzt werden. Für die Applikation erfolgen diese Vorgänge transparent.

3.1.2 Performance und Consistency/Durability Trade-Off

Write Concerns beschreiben die Garantien, welche die MongoDB Datenbank gegenüber einer Client Applikation für Schreiboperationen bietet. Je niedriger der gewählte Level, umso kürzer sind die Latenzzeiten bei Schreibvorgängen. Allerdings erfolgt bei kürzeren Latenzzeiten eine Lockerung der Consistency und der Durability. So kann es sein, dass in gewissen Fehlerszenarien Daten nicht persistiert und somit verloren gehen. In Kontrast dazu weisen Schreibvorgänge bei höheren Leveln eine höhere Latenzzeit auf, jedoch wird auf diese Weise einem Datenverlust entgegen gewirkt. Der Trade-Off ist dabei auf die Anforderungen der Client Applikation zurückzuführen. So kann kritischen Operationen gegenüber Persistenz auch unter Fehlerszenarien garantiert werden, und weniger kritischen Operationen kürzere Latenzzeiten und damit höhere Performance eingeräumt werden. Folgender Absatz gibt eine Auflistung der vorhandenen Write Concerns, sortiert vom niedrigsten zum höchsten Level:

- Unacknowledged: MongoDB sendet keine Bestätigung über den Erhalt und Ausführung einer Schreiboperation an den Client zurück. Dieser Level bietet höchste Geschwindigkeit, mit dem Risiko Daten bei Netzwerkunterbrechungen zwischen Client und Server als auch bei Netzwerkpartitionen und Systemfehlern innerhalb der MongoDB Datenbank permanent zu verlieren.
- Acknowledged: Es wird der Client Applikation sowohl der Erhalt als auch die Ausführung einer Schreiboperation im Hauptspeicher der MongoDB Datenbank bestätigt. Damit können einerseits Fehler wie Netzwerkunterbrechungen zwischen Client und Server als auch andererseits Fehler wie Duplicated Keys abgefangen werden. Das Warten auf die Rückmeldung erhöht jedoch die Latenzzeit und vermindert die Performance und den Durchsatz des Clients.
- Journaled: Die Schreiboperation wird erst dann bestätigt, sobald diese erfolgreich im Journal festgehalten wird. Damit wird sichergestellt, dass nach Systemfehlern die zu einen plötzlichen Shutdown des Servers führen die Daten wieder hergestellt werden können. Schreiboperationen werden dabei zuerst im Hauptspeicher geschrieben,

und müssen danach auf das nächste Journal Commit warten. Dieses wird periodisch durchgeführt und kann serverseitig zwischen 2 und 300 Millisekunden konfiguriert werden. Der Default Wert ist dabei auf 100 Millisekunden eingestellt. Niedrigere Werte erhöhen die Performance auf Kosten der Diskauslastung.

- Replica Acknowledged: Die bisherigen Write Concerns haben sich auf dem Primary innerhalb eines Replica Set bezogen. Mit Replica Acknowledged wird es ermöglicht, die Schreiboperation erst dann zu bestätigen, wenn diese sowohl im Hauptspeicher des Primary als auch in den der Secondaries geschrieben wurde.

Allen Write Concern kann ein Timeout Parameter übergeben werden. Sobald dieser Timeout bei einem Schreibvorgang erreicht wird, bewirkt dieser das Auslösen eines Errors. Dieser wird auch ausgelöst, wenn der Schreibvorgang eventuell erfolgreich auf der MongoDB Datenbank durchgeführt wird, wobei in diesem Fall kein Rollback stattfindet. Wird kein Timeout Parameter übergeben, so blockt die Schreiboperation auf unbestimmte Zeit [25].

3.1.3 Client Anbindung

Für die Kommunikation der Clientapplikation zu dem MongoDB Cluster werden für alle gängigen Programmiersprachen entsprechende Driver zur Verfügung gestellt. Die Anbindung soll hier mit dem in Tabelle 1 beschriebenen Java Driver gezeigt werden. Um sich zu einer Datenbank zu verbinden, wird eine Instanz des MongoClient benötigt, welchem bei der Initialisierung IP-Adresse und Port der Server übergeben werden.

```
1 MongoClient mongoClient = new MongoClient(Arrays.asList(
2                 new ServerAddress("192.168.122.10", 27017),
3                 new ServerAddress("192.168.122.11", 27017),
4                 new ServerAddress("192.168.122.12", 27017))
5                                         );
```

Beispiel 1: Initialisierung eines MongoClients

Als nächsten Schritt kann der benötigte Write Concern Level konfiguriert werden. Wird dieser Schritt nicht durchgeführt, so wird Acknowledged als Default Wert gesetzt. Beispiel 2 zeigt die Konfiguration des Write Concern Unacknowledged.

```
1 mongoClient.setWriteConcern( WriteConcern.UNACKNOWLEDGED);
```

Beispiel 2: Konfigurieren des Write Concern Levels

Beispiel 3 zeigt die Herstellung einer Verbindung zu einer Datenbank. Wenn die spezifizierte Datenbank noch nicht existiert, wird diese neu angelegt.

```
1 DB db = mongoClient.getDB( "mydb" );
```

Beispiel 3: Herstellen einer Datenbankverbindung

Nun muss noch die Verbindung zu einer Collection hergestellt werden, bevor der Datenbank Dokumente hinzugefügt werden können.

```
1 DBCollection coll = db.getCollection("testCollection");
```

Beispiel 4: Herstellung einer Verbindung zu einer Collection

Beispiel 5 demonstriert das Einfügen eines Dokuments, welches aus dem Key mit dem Wert "1" und aus dem Value mit dem Text "Test" besteht [26].

```
1 collection.insert(new BasicDBObject("1", "Test"));
```

Beispiel 5: Einfügen eines Dokuments

Die in Beispiel 6 demonstrierte For-Schleife stellt den Kern der für dieser Arbeit programmierten Client Applikation dar. Dabei werden die in der For-Schleife definierte Anzahl an Dokumenten der MongoDB Datenbank seriell hinzugefügt. Das Programm selbst wird noch um Fehlerbehandlung und Messprotokolle ergänzt und ist im Anhang A einzusehen.

```
1 for (int i = 0; i < this.insert; i++) {
2     this.collection.insert(new BasicDBObject(String.valueOf(i), "test"));
3 }
```

Beispiel 6: Einfügen mittels For-Schleife

3.2 Couchbase

Couchbase ist eine einfache, skalierbare und hochperformante verteilte NoSQL Datenbank, optimiert für die Anforderungen moderner interaktiver Webapplikationen. Couchbase unterstützt die Funktionalität mehrere Server gleichzeitig dem Cluster hinzuzufügen oder zu entfernen, wobei die Daten automatisch neu verteilt werden. Über ein gut gestaltetes Web-Interface können einerseits alle administrativen Tasks ausgeführt und andererseits effizientes Monitoring betrieben und Ressourcenauslastung verfolgt werden. Des Weiteren lässt sich die Datenbank über ein REST-Interface administrieren [27].
Couchbase setzt bei seiner Architektur auf der In-Memory Datenbank Memcached auf, welche auf über 10 Jahre Caching Technologie zurückblicken kann. Das Storage Backend wurde dabei durch jenes der CouchDB Datenbank ersetzt. Die Idee dabei war eine Datenbank zu entwerfen, die das dokumentenorientierte Model als auch die Index und

Abfragemöglichkeiten der CouchDB mit den performanten, leicht skalierbaren und hochverfügbaren Eigenschaften der Membased kombiniert. Einige Codefragmente wurden dabei dennoch neu programmiert um die Performance und Ressourcenauslastung zu verbessern und Hochverfügbarkeit via Replikation zu garantieren [28].
Die Datenbank wird als Open-Source Projekt verwaltet, wobei kommerzieller Support seitens des Unternehmens Couchbase, Inc. möglich ist [29].
Client Libraries sind in allen gängigen Programmiersprachen verfügbar. Des Weiteren wird mittels Couchbase Mobile eine embedded Datenbanken für iOS, Android, Windows und Linux angeboten, welche über ein Sync Gateway Daten automatisch synchronisiert und es Entwicklern erleichtert moderne Applikationen mit nativer offline - und online Verfügbarkeit auszustatten [27].

3.2.1 Architektur

Eine Couchbase Installation setzt sich typischerweise aus mehreren Servern, bestehend aus Commodity Hardware, zusammen. Abbildung 4 zeigt den prinzipiellen Aufbau und veranschaulicht dabei die Replikation der Daten innerhalb des Clusters.

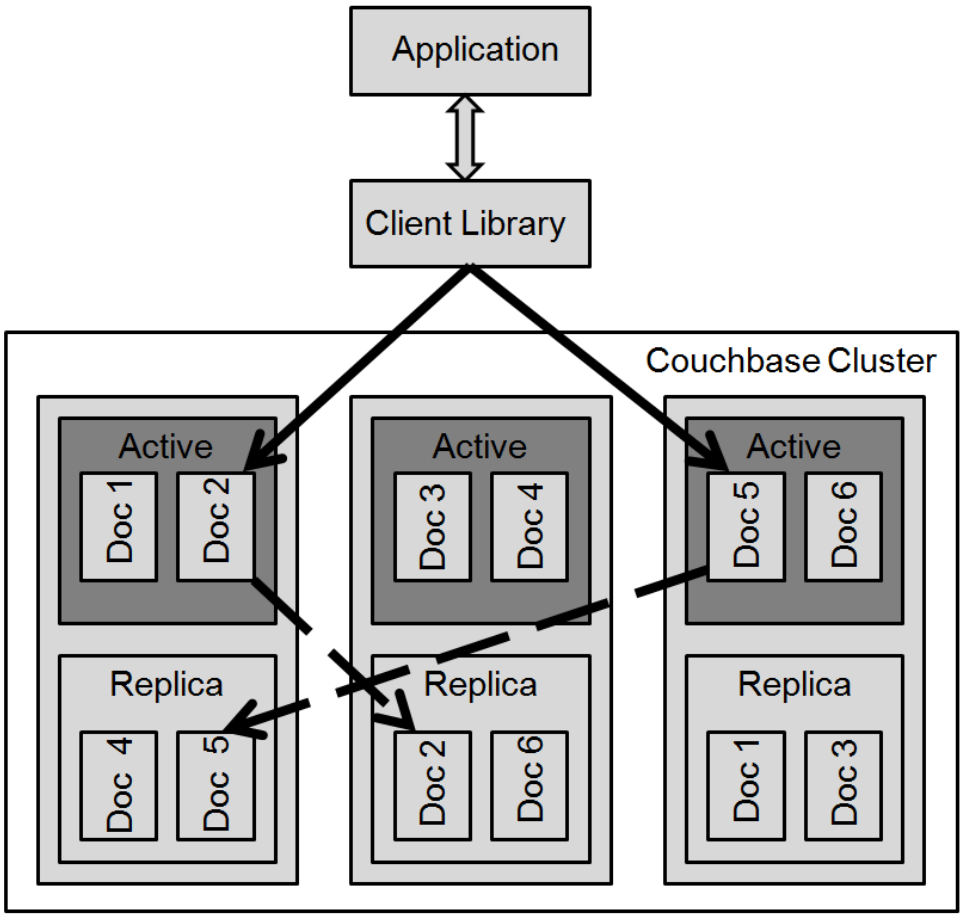


Abbildung 4: Couchbase Architektur [30]

Der Cluster kann sowohl Dokumente in JSON Format als auch Binäre Daten speichern, und kann daher sowohl als Document Store als auch als Key-Value Store betrieben werden. Dokumente werden dabei als atomare Einheiten verwaltet und in sogenannten Buckets gespeichert. Der Begriff Bucket kann dabei den unter MongoDB verwendeten Begriff Collection oder einer Tabelle in relationalen Datenbanken gleichgesetzt werden und bildet eine logische Gruppierung der Dokumente auf physischen Ressourcen ab [30].
Jedes dieser Buckets wird in 1024 logische Partitionen abgebildet, welche als vBucket bezeichnet werden. Diese vBuckets werden dabei symmetrisch auf die im Cluster zur Verfügung stehenden Server aufgeteilt. Die Aufteilung wird in einer Cluster Map verwaltet. Es ist Aufgabe des Couchbase Drivers, diese Cluster Map zu verwalten und Updates entgegenzunehmen um somit Lese- und Schreibvorgänge auf den entsprechenden Server durchzuführen [30].
Die Aufteilung der Daten in vBuckets und der damit erfolgten Verteilung der Schreib- und Leseoperationen zwischen den Servern ist die Art und Weise wie Couchbase Auto-Sharding implementiert.
Um die Verfügbarkeit zu erhöhen, können Daten repliziert werden. Diese Spezifikation wird auf Bucketebene vorgenommen, dabei kann zwischen der Replikation auf einen, zwei oder drei weiteren Servern gewählt werden. Ebenso wie bei MongoDB ist auch hier die Replikation auf zwei weitere Server sehr geläufig. Replizierte Daten können zwar gelesen aber nicht beschrieben werden. Wie in Abbildung 4 ersichtlich verwalten alle Server im Cluster sowohl aktive als auch replizierte Partitionen. Sobald ein Server aufgrund eines Fehlers nicht mehr erreichbar ist, wird für alle aktiven Partitionen welche sich zum Zeitpunkt des Ausfalls auf diesem Server befunden haben ein neuer Master gewählt [30]. Die Vorgänge hierzu erfolgen für die Client Applikation transparent.
Entgegen des Ansatzes von MongoDB, wo es einen eindeutigen Primary mit zugehörigen Secondaries gibt, verwalten hier alle Server sowohl aktive als auch replizierte Partitionen.

3.2.2 Performance und Consistency/Durability Trade-Off

Einer Couchbase Connection können zwei Parameter übergeben werden, um den Trade-Off zwischen Performance und Consistency zu bestimmen [3]. Die beiden Parameter lauten:

- PersistTo: Gibt den Entwickler die Möglichkeit zu bestimmen, auf wie vielen Servern ein Dokument persistiert werden soll. Dabei kann zwischen den Optionen ZERO, ONE, TWO, THREE und MASTER gewählt werden, wobei ONE und MASTER gleichzusetzen sind [31].
  ZERO entspricht dem Default, wobei ein Response an den Client erfolgt, nachdem die Operation erfolgreich im Hauptspeicher des Masters ausgeführt worden ist. Wird ein Wert abseits von Zero gewählt, so wird ein Response an den Client erst dann ausgeführt, sobald die Daten erfolgreich auf die Festplatten der spezifizierten Anzahl an Servern durchgeführt worden ist [3].

- ReplicateTo: Lässt den Entwickler bestimmen, auf wie viele Server ein Dokument repliziert werden soll, bevor dem Client die erfolgreiche Durchführung bestätigt wird. Auswählbare Optionen sind ZERO, ONE, TWO, THREE. Der Response an den Client wird dabei erst nach erfolgreicher Replizierung der konfigurierten Option zurückgesendet [32].
  Auf wie viele Servern die Daten repliziert werden sollen wird bereits bei der Konfiguration des Buckets definiert. Der hier beschriebene Parameter spezifiziert auf wie viele Servern die Replikation erfolgen soll, bevor der Response an den Client erfolgt. Ist der Wert ungleich ZERO, so entspricht dies einer synchronen Replizierung mit der spezifizierten Anzahl an Servern. Beim Wert ZERO wird eine rein asynchronen Replizierung durchgeführt.

In der Praxis haben sich dabei folgende Verfahren als sinnvoll erwiesen:

- PersistTo.Zero, ReplicateTo.Zero: Der Response an den Client erfolgt sobald die Operation im Hauptspeicher des Masters ausgeführt worden ist. Kommt es zu einem Systemfehler, so gehen dabei alle Daten verloren welche noch nicht persistiert oder repliziert worden sind.
- PersistTo.Master, ReplicateTo.Zero: Erst wenn die Daten auf Festplatte geschrieben wurde erfolgt der Response. Kommt es zum Ausfall des Masters und wurden die Daten noch nicht repliziert, so sind diese für den Zeitraum des Ausfalls nicht verfügbar und abhängig von der Schwere des Ausfalls möglicherweise verloren.
- PersistTo.Zero, ReplicateTo.Two: Der Response an den Client erfolgt sobald die Daten an zwei weitere Server repliziert wurden. Dabei wird davon ausgegangen, dass das Bucket auch nur zwei anstatt der maximal konfigurierbaren drei Server als Replikat konfiguriert hat.

Der Begriff Master bezieht sich bei den Definitionen auf alle aktiven Buckets der jeweiligen Servern im Verbund [33].

3.2.3 Client Anbindung

Dieses Unterkapitel zeigt die Verwendung des in Tabelle1 angegebenen Java Drivers, um eine Verbindung zum Cluster herzustellen und Dokumente einzufügen.
Zuerst werden die einzelnen Server einer ArrayList hinzugefügt, welche im nächsten Schritt dazu genutzt wird eine Verbindung zum Couchbase Client herzustellen. Beispiel 7 zeigt die Vorgehensweise:

```
1 ArrayList<URI> nodes = new ArrayList<URI>();
2 nodes.add(URI.create("http://192.168.122.10:8091/pools"));
3 nodes.add(URI.create("http://192.168.122.11:8091/pools"));
4 nodes.add(URI.create("http://192.168.122.12:8091/pools"));
5 client = new CouchbaseClient(nodes, "bucket1", "");
```

Beispiel 7: Herstellen einer Datenbankverbindung

Der Couchbase Client Instanz werden dabei sowohl die Serveradressen als auch der Bucketname und das Passwort übergeben. Das Bucket kann auch ohne Passwort erstellt werden, in diesem Fall wird ein leerer String übergeben. Im Gegensatz zur MongoDB, wo eine Collection neu angelegt wird wenn diese nicht existiert, muss bei Couchbase das Bucket bereits im Cluster vorhanden sein - andererseits schlägt die Operation fehl. Konnte die Verbindung erfolgreich hergestellt werden, können Operationen ausgeführt werden. Beispiel 8 zeigt das Einfügen eines einfachen Dokuments:

```
1 client.set("1", "Test", PersistTo.Zero, ReplicateTo.Two).get();
```

Beispiel 8: Einfügen eines Dokuments

Bei dem gewähltem Beispiel wird der Vorgang solange geblockt, bis das Dokument auf zwei weitere Server repliziert wurde. Mit der set()-Operation wird das Ergebnis abgefragt. Auf ein Error Handling wird an dieser Stelle verzichtet, und kann in Anhang A eingesehen werden.
Beispiel 9 zeigt das serielle Einfügen über eine For-Schleife, welches wie bereits in der MongoDB erläutert den Kern der Tests darstellt.

```
1 for (int i = 0; i < this.inserts; i++) {
2     client.set(String.valueOf(i), "Test",
3     PersistTo.Zero, ReplicateTo.Two).get();
4 }
```

Beispiel 9: Einfügen mittels For-Schleife

3.3 CouchDB

CouchDB ist eine Datenbank welche für das Web entwickelt wurde. Die Daten werden im JSON Format gespeichert und über HTTP angesprochen. Abfragen werden dabei in JavaScript formuliert. Web Applikationen können mittels REST Interface direkt mit der CouchDB Datenbank interagieren. Die Daten können via asynchroner Replikation auf mehrere Rechner verteilt werden. CouchDB unterstützt weiters den Ansatz einer Master Master Architektur, wobei Konflikte welche bei der Synchronisation entstehen können automatisch erkannt werden und Möglichkeiten zur Lösung bereitstehen [34].
CouchDB wird von Apache als OpenSource Projekt geführt. Kommerziellen Support wird derzeit von keinem Unternehmen angeboten.

3.3.1 Architektur

CouchDB implementiert eine Master Master Architektur. Damit wird es Clients ermöglicht Daten über mehrere Server zu verteilen - ein Prozess welcher als Sharding bezeichnet wird. Im Gegensatz zu MongoDB und Couchbase unterstützt CouchDB jedoch kein Auto Sharding. Um die Daten über mehrere Nodes zu verteilen, müssen entweder die Applikationen über entsprechende Logik verfügen oder diese wird mittels eines Middleware Layers bereitgestellt. Weiters gibt es einige Projekte, welche CouchDB um eine Auto Sharding Funktionalität erweitern. Für Interessenten sei auf Anhang B verwiesen.
CouchDB bietet zwar Replikationsmöglichkeiten, diese sind jedoch nur asynchron. Alternativ kann die Replikation auch von außerhalb angestoßen werden (in den meisten Fällen entweder durch den Client oder einer serverseitigen Middleware). Abbildung 5 zeigt das Design einer Master-Master Architektur inklusive asynchroner Replikation. Aufgrund der Gegebenheiten ermöglicht diese Architektur die Manipulation von gleichen Dokumenten auf unterschiedlichen Knoten des Systems. Werden die Dokumente zur selben Zeit unterschiedlichen Änderungen unterzogen, so kommt es bei einer anschließend erfolgten asynchronen Replikation zu Konflikten, da nicht klar ist welches Dokument nun das gültige ist. Diese Thematik wird in Kapitel 3.3.4. gezeigt.

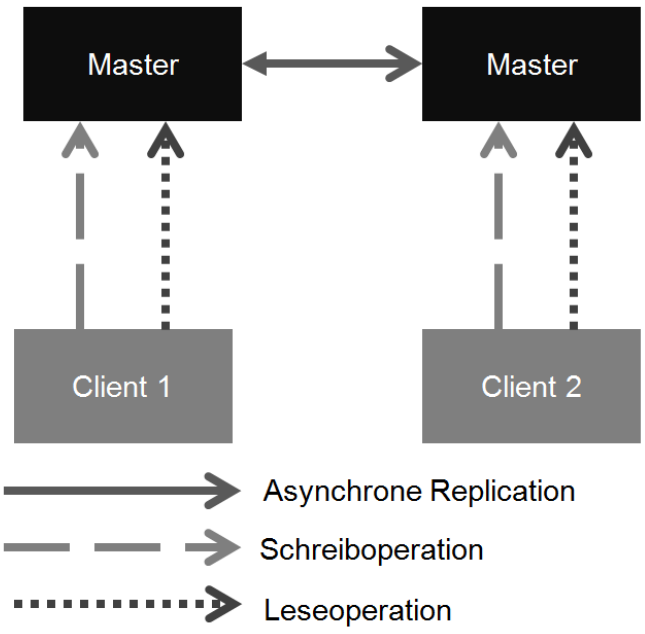


Abbildung 5: CouchDB Architektur

3.3.2 Performance und Consistency/Durability Trade-Off

CouchDB verfügt über folgende Parameter, um den Trade-Off zwischen Latenzzeiten und Konsistenz zu beeinflussen:

- Batch: Um den Indexing- und Disk Overhead zu minimieren, kann die Option batch=ok gesetzt werden. Dabei werden Batches von Dokumenten im Hauptspeicher verwaltet und die Persistierung auf Disk erfolgt erst sobald entweder ein Threshold erreicht wird oder von außerhalb zusätzlich ein Aufruf erfolgt. Der Schreibvorgang an sich wird gegenüber dem Client bereits bestätigt, sobald die Operation im Hauptspeicher durchgeführt wurde. Diese Option ermöglicht einen enormen Performancegewinn zugunsten der Durability, denn alle Dokumente welche bei einem Systemfehler des Servers nur im Hauptspeicher verwaltet wurden gehen dabei verloren. Defaultmäßig findet der Schreibvorgang auf Disk einmal pro Sekunde statt. Um die reduzierte Datenintegrität zu reflektieren, wird der Response Code Accepted anstatt des Response Code Created an den Client zurückgegeben [9].
- Synchron: Jede Schreiboperation wird dem Client erst bestätigt, sobald diese vollständig durchgeführt und auf Disk geschrieben wurde. Jedoch wird kein expliziter fsync angestoßen [9] ,wodurch Festplattenbuffer nicht geleert werden [35]. Kommt es zu einem Stromausfall, so können die Daten in Abwesenheit einer Battery Backup Unit [36] nicht persistiert werden.
- Fsynced: Mit der Option Delayed_Commits=false kann serverseitig konfiguriert werden, dass nach jedem Schreibvorgang ein fsync durchgeführt wurde. Dies bietet höchsten Grad an Durability, jedoch zulasten des Durchsatzes.

3.3.3 Client Anbindung

CouchDB verfügt über eine einheitliche REST-Schnittstelle, es sind daher keine Libraries für Programmiersprachen verfügbar. Beispiel 10 zeigt die Initiierung einer Datenbankverbindung mittels der in JAVA verfügbaren HTTP-Library. Da CouchDB keine Auto-Sharding Funktionalität besitzt, wird in diesem Beispiel die Verbindung zu lediglich einem der im Cluster verfügbaren Knoten hergestellt.

```
1 URI uri = new URI( "http://192.168.122.10:5984/TestDB" );
2 HttpURLConnection conn = (HttpURLConnection) uri.toURL().openConnection();
3 conn.setRequestMethod("POST");
4 conn.addRequestProperty("Content-Type", "application/json");
5 conn.setRequestProperty("Accept", "application/json");
```

Beispiel 10: Herstellen einer Datenbankverbindung

In Zeile 1 wird eine Variable vom Typ URI mit dem zu verwendeten Protokoll, der Serveradresse, den Serverport und den Datenbanknamen initialisiert. Diese Variable wird in Zeile 2 dazu verwendet eine HTTP Verbindung herzustellen. Zeile 3 spezifiziert die Request Methode POST. Diese Request Methode wird bei REST Schnittstellen für Einfügeoperationen verwendet. Für Updateoperationen würde PUT Verwendung finden, GET für Leseoperationen und DELETE um Dokumente aus der Datenbank zu entfernen. Zeile 4

spezifiziert JSON als Content-Typ des HTTP Body. Genauso wird ein Response vom Server im JSON Format erwartet, welcher in Zeile 5 spezifiziert wird.[37]
Um nun Dokumente der Datenbank hinzufügen zu können, wird noch eine Stream-Instanz benötigt. Beispiel 11 zeigt die Vorgehensweise.

```
1 OutputStreamWriter out = new OutputStreamWriter(conn.getOutputStream());
2 out.write("{\"1\":\"test\"}");
3 out.flush();
4 out.close();
```

Beispiel 11: Einfügen eines Dokuments

Zeile 1 öffnet einen Output Stream, über welchen in Zeile 2 ein Dokument in JSON Format geschrieben wird. Zeile 3 gibt explizit die Anweisung alle Schreibbuffer clientseitig zu leeren, bevor in Zeile 4 die Verbindung wieder geschlossen wird.
Zeile 1 des Beispiels 12 ermittelt im Anschluss den Response Code, bevor in Zeile 2 die Datenbankverbindung geschlossen wird. Beim Response Code handelt es sich um einen HTTP Response Code, welcher bei einer erfolgreichen Schreiboperation 201 Created bzw. bei Verwendung des Flags batch=ok 202 Accepted liefert. Die gesamte Klasse ist in Anhang A einzusehen, worin auch auf Error Handling eingegangen wird.

```
1 conn.getResponseCode();
2 conn.disconnect();
```

Beispiel 12: Schließen der Datenbankverbindung

Wie schon in den vorangegangenen Beispielen wird auch hier das Einfügen von Dokumenten via einer For-Schleife gelöst. Für die Realisierung sei auf Anhang A verwiesen. Zu beachten sind bei seriellen Einfügeoperationen verschiedene Netzwerkeinstellungen und Parameter des TCP-Protokolls. So war es für die in dieser Arbeit durchgeführten Tests notwendig die Defaultwerte folgender Optionen am Client zu überschreiben:

- Ephemeral Ports: Defaultwert von 32768 bis 61000 wurde überschrieben mit dem Wert 18000 bis 65535.
- Open Files: Defaultwert von 1024 wurde auf 999999 erhöht, um genügend File Handler für die Verbindungen zur Verfügung zu haben.
- tcp_tw_reuse: Der Defaultwert von 0 wurde auf 1 umgestellt um die Option zu aktivieren. Erlaubt die Verwendung von Verbindungen welche sich im TIME_WAIT Status befinden für neue Verbindungen zu nutzen [38].
- tcp_tw_recycle: Der Defaultwert von 0 wurde auf 1 umgestellt um die Option zu aktivieren. Diese Option ermöglicht schnelle Wiederverwendung von Sockets welche sich im TIME_WAIT befinden [38].

3.3.4 Conflict Detection

Aufgrund CouchDBs zugrunde liegender Architektur kann es zu Konflikten zwischen verschiedenen Versionen des gleichen Dokuments kommen. Ein Konflikt wird dabei dann erzeugt, wenn auf unterschiedlichen Servern der verteilten Datenbank Änderungen am gleichen Dokument vollzogen werden. Dieses Szenario wird mit Abbildung 6 veranschaulicht. Dabei wird ein repliziertes Dokument von zwei Clients bearbeitet, wobei beide Clients unterschiedliche Änderungen vornehmen. Nach Durchführung der Änderungen werden die beiden unterschiedlichen Versionen 2A und 2B auf jeweils einen Server des Verbunds erfolgreich geschrieben.

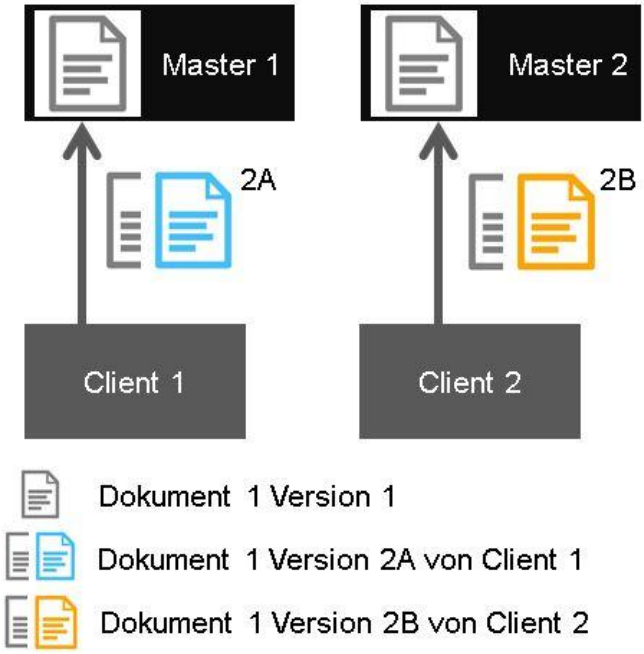


Abbildung 6: Erzeugen unterschiedlicher Versionen

Abbildung 7 zeigt das Ergebnis des Schreibvorgangs. Die beiden Server haben nun einen unterschiedlichen Datenbestand. Je nachdem, auf welche Seite nun lesend zugegriffen wird, wird entweder die Version 2A oder die Version 2B desselben Dokuments an den Client gesendet. In der Abbildung eingezeichnet ist weiters die asynchrone Replikation, welche im Anschluss an den Schreibvorgang erfolgt und die Daten der beiden Seiten abgleicht.

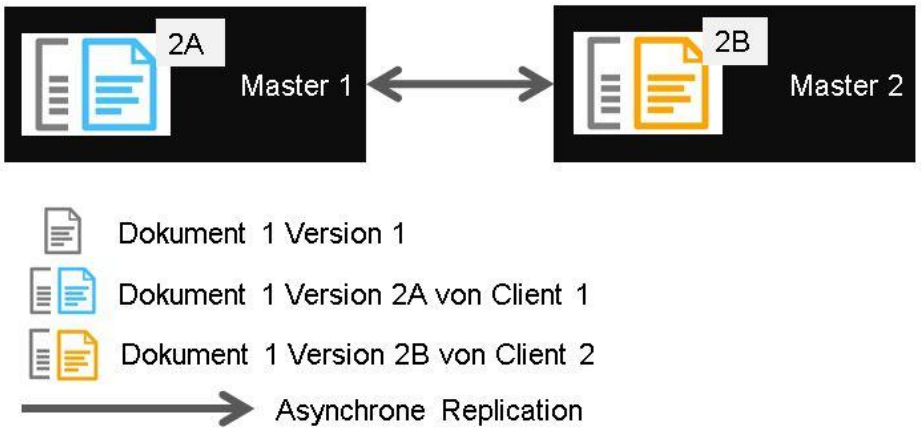


Abbildung 7: Asynchrone Replikation

Dabei kommt es nun zum Konflikt, da für den CouchDB Cluster unklar ist welches Dokument das gültige ist. CouchDB erkennt zwar diesen Konflikt, die Lösung bleibt allerdings den Client überlassen. Abbildung 8 zeigt den entstandenen Konflikt.

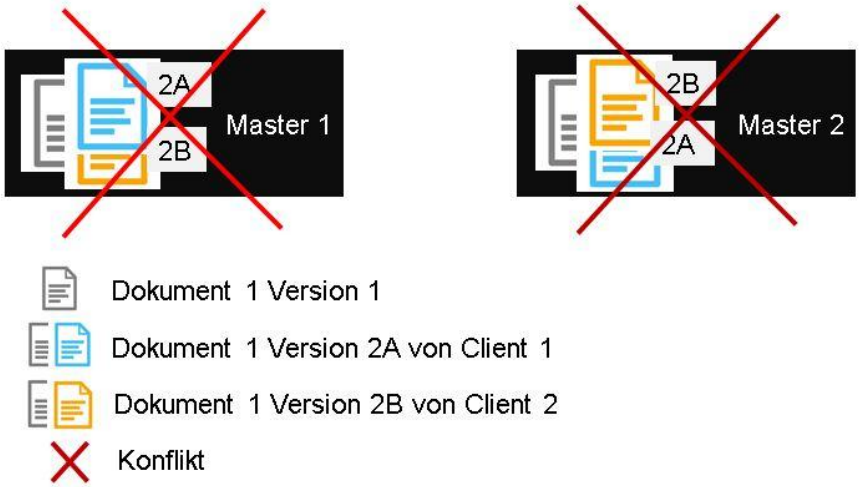


Abbildung 8: in Konflikt stehende Versionen

Obwohl die Konfliktlösung den Client unterliegt, wird von der Datenbank ein zweistufiger Prozess durchgeführt, welcher einer der beiden Versionen als Gewinnerrevision (winning revision) wählt und die andere Version als in Konflikt stehend kennzeichnet. Die Gewinnerrevision wird bei eingehenden Requests an den Client ausgehändigt, bis der Konflikt vom Client gelöst wird. Der zweistufige Prozess zur Wahl der Gewinnerversion sieht dabei wie folgt aus:

- Vergleich der Revision Number: Jede Änderung des Dokuments wird mit einer Erhöhung der Revisionsnummer festgehalten. Das Dokument, an welchen mehr Änderungen durchgeführt wurde, wird als Gewinnerrevision gewählt.
- Vergleich der Hash-Values: Dieser Vergleich wird durchgeführt, wenn die in Konflikt stehenden Dokumente jeweils gleich vielen Änderungen unterlagen. Ist dies der Fall, so wird der Hash-Value der Dokumente in ASCII sort order verglichen. Dabei ist der Gewinner jener, welcher den höheren Wert hat.

Die Prozedur wird auf jeden Knoten durchgeführt, womit schlussendlich ein konsistenter Zustand erreicht wird. Abbildung 9 zeigt das Resultat, wobei die Version 2A aufgrund eines höheren Hash-Values als Gewinnerrevision hervorgeht.

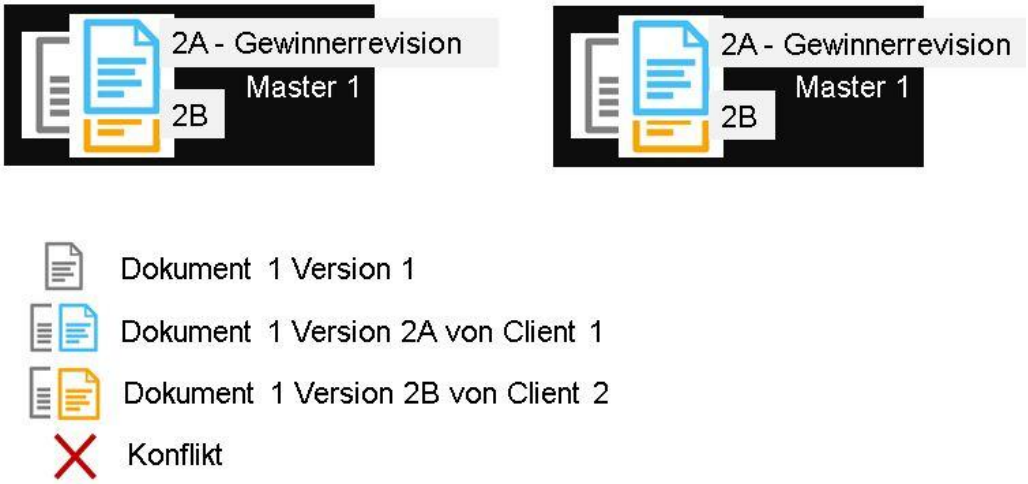


Abbildung 9: Lösung des Konflikts

Um den Konflikt jedoch zu lösen muss der Client eine der beiden Dokumentenversionen auswählen, oder die beiden zusammenführen. In Anschluss wird die vom Client als ungültig betrachtete Version mit einem Delete-Request gelöscht, womit der Konflikt gelöst wird [9]. Wie dies in der Praxis durchgeführt wird ist in Anhang C beschrieben.
Damit clientseitige Statusabfragen (Polling) nicht notwendig sind, haben Clients die Möglichkeiten Events bei Änderungen der Datenbank zu empfangen. Damit kann bei Entstehung von Konflikten reagiert werden [39].

4 Testumgebung

Alle Tests werden unter den gleichen Rahmenbedingungen durchgeführt. Dazu verwendet wird eine virtuelle Serverlandschaft bestehend aus bis zu vier Knoten, welche zu einem verteilten System zusammengeschalten sind. Das Wirtsystem fungiert dabei zeitgleich als Client, welcher auf den Servern Last erzeugt.

4.1 Hard- und Softwarekonfiguration

Als Wirtsystem wird ein handelsüblicher Rechner mit folgender Ausstattung verwendet:

- Hauptspeicher: Corsair Vengeance 32GB (4x8GB) DDR3 1600 MHz
- Prozessor: AMD FX-8120 8x3,10GHz
- Festplatte: 2x Western Digital WD30EZRX Green 3TB im RAID 0 Verbund
- Betriebssystem: Fedora 19
- Virtualisierungslayer: QEMU/KVM mit libvirt

Die Server werden als virtuellen Systeme realisiert, und haben als Ressourcen zugeordnet:

- Hauptspeicher: 1GB
- Prozessor: 1x3,10 GHz
- Festplatte: 100GB
- Betriebssystem: Ubuntu Server 12.04

Als Client wird das Wirtsystem verwendet, welcher das im Anhang A beigefügte Programm ausführt. Für die Entwicklung und Ausführung wurde der Java SDK 7 Update 70 verwendet. Die folgenden Unterkapitel beschreiben die Konfiguration der Datenbanken.

4.1.1 MongoDB Setup

Die MongoDB Testumgebung besteht aus einem Replica-Set, welches sich aus einen Primary (Master) und mehreren Secondaries (Slaves) zusammensetzt. Abbildung 7 zeigt dabei die Teststellung in ihrer Ausgangssituation, mit genau zwei Secondaries. Bei den Tests wird dieser Umgebung einerseits ein zusätzlicher Secondary hinzugefügt, als auch ein Secondary entfernt. Die Clients sind dabei so konfiguriert, dass sie auch Datensätze von den Secondaries lesen können. Schreibvorgänge können wie bereits in Kapitel 3.2.1 beschrieben nur auf dem Primary durchgeführt werden. Die Verbindungen zu den jeweiligen Servern werden clientseitig mittels des MongoDB Java Driver verwaltet. Es ist somit nicht Aufgabe des Entwicklers zu koordinieren, welche Operationen auf welchen Knoten stattfinden.

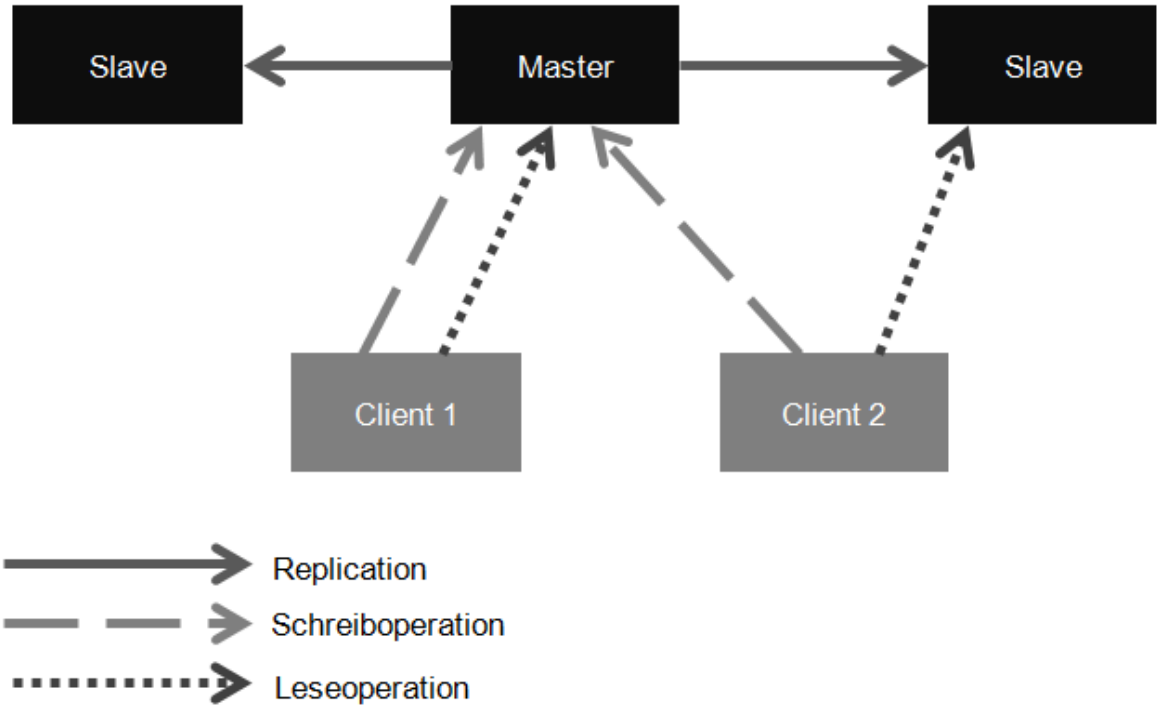
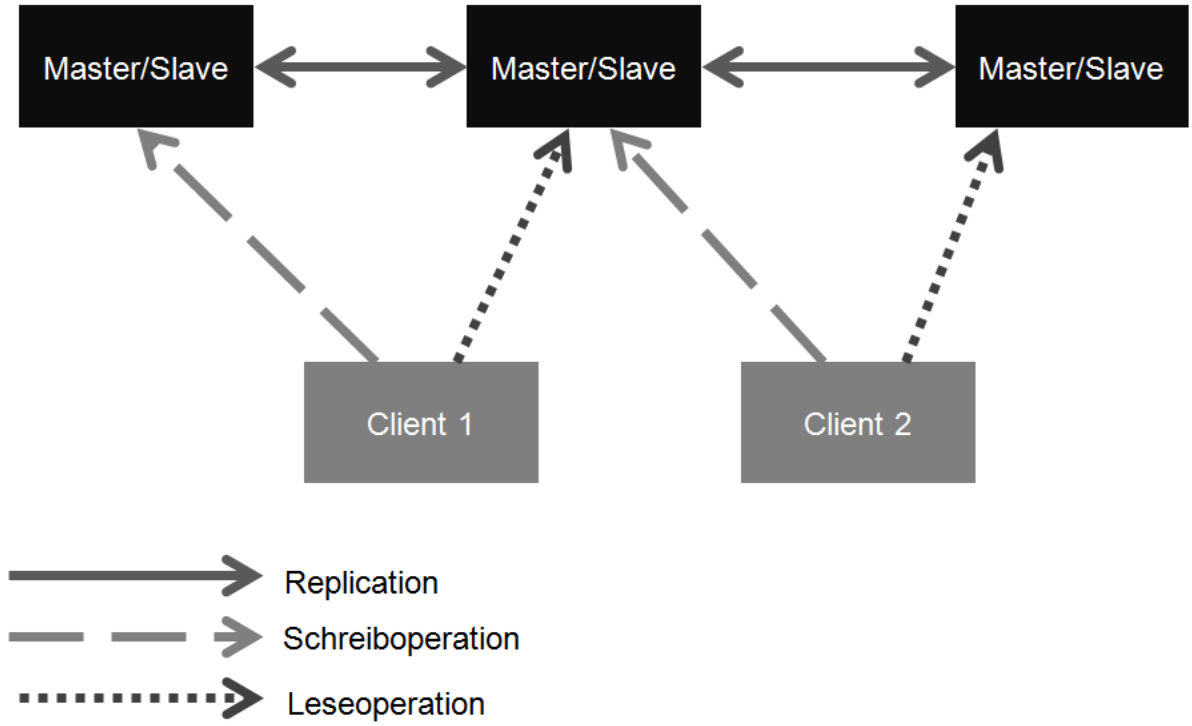


Abbildung 10: MongoDB Installation

4.1.2 Couchbase Setup

Im Gegensatz zur MongoDB mit einem zentralen Master werden bei der Couchbase Schreibvorgänge auf allen Servern durchgeführt, wobei jeder Server für die Verwaltung eines bestimmten Bereichs (Key Space) zuständig ist. Diese Aufteilung des Key Space erfolgt durch vBuckets, und erfolgt symmetrisch. Abbildung 10 zeigt auch hier die Teststellung zu Beginn, wobei danach eine Erweiterung als auch eine Reduktion um einen Knoten durchgeführt wird. Die Clients können alle Lesevorgänge auf allen Knoten durchführen. Wie in MongoDB wird auch hier die Verbindung mittels Driver verwaltet, welcher die Schreibvorgänge auf die entsprechenden Server durchführt. Auf Seiten der Entwickler muss daher nicht koordiniert werden, welcher Schreibvorgang auf welchen Knoten zu erfolgen hat.

Abbildung 11: Couchbase Installation



4.1.3 CouchDB Setup

Bei der CouchDB können Schreibvorgänge auf allen Knoten durchgeführt werden, wobei der Entwickler dafür zuständig ist zu definieren, welche Schreibvorgänge auf welchen Knoten ausgeführt werden. Die Testumgebung wurde daher dieser der MongoDB angeglichen. Dabei werden Schreibvorgänge auf einen Knoten ausgeführt und anschließend repliziert. Lesevorgänge werden dabei auch von den Replica Servern durchgeführt um das Konsistenzverhalten zu messen. Die Teststellung wird wie bei MongoDB und Couchbase einmal um einen Knoten erweitert und einmal reduziert. Abbildung 11 zeigt die Installation zu Beginn der Tests.

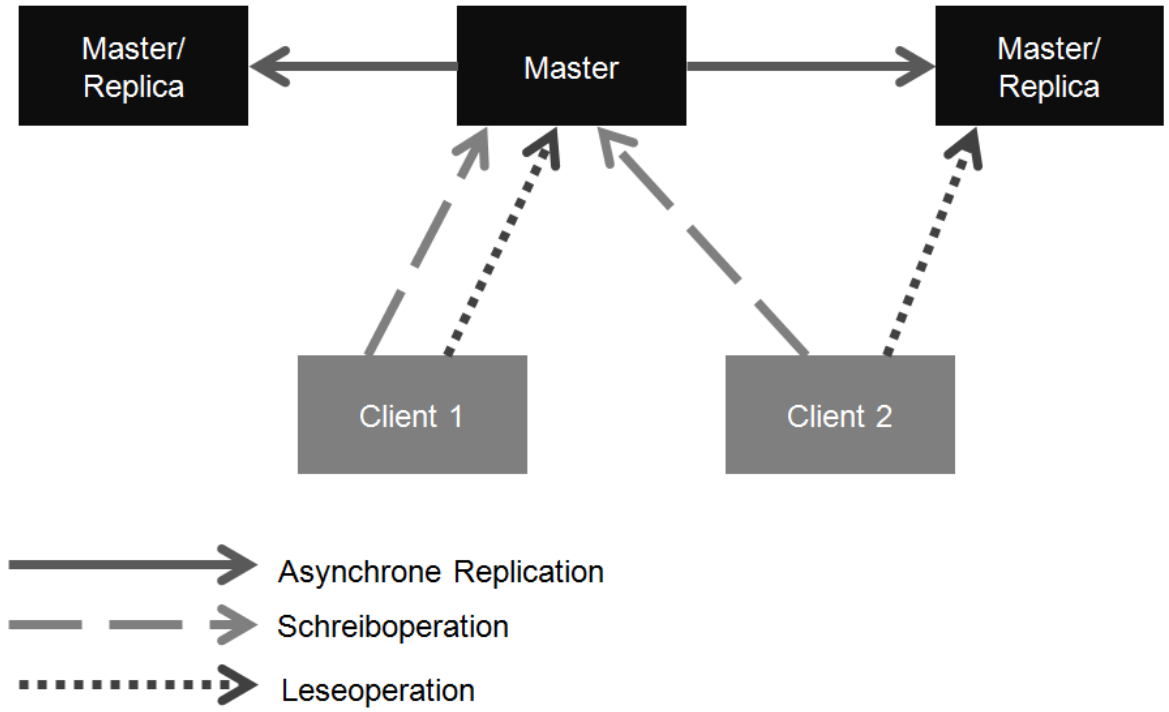


Abbildung 12: CouchDB Installation

4.2 Testfälle

Um die Datenbanken unter Last zu setzen und Messungen vorzunehmen, wurde basierend auf Java ein Programm entwickelt, welches seriell Dokumente im JSON Format an die Datenbanken schickt. Der Kern des Programms basiert auf einer in den Kapitel 3.1.3, 3.2.3 und 3.3.3 beschriebenen Schleife. Diese wurde um entsprechendes Error-Handling und um ein Messprotokoll erweitert, in welchen die Latenzzeiten und das Konsistenzverhalten festgehalten werden. Das Programm bietet weiters die Möglichkeiten Threads synchron bzw. asynchron ablaufen zu lassen, was starke Auswirkungen auf Performance, Konsistenz und auf die Durability hat. Des Weiteren wurden Methoden programmiert, welche die Datenbank für die Tests präparieren und nach den Tests wieder in den Ausgangszustand zurücksetzen. Die CPU-Auslastung der Server liegt bei der Ausführung der Tests zwischen 90% und 100%. Festgehalten werden in allen Tests folgende Ergebnisse:

- Performance (Latenzzeiten): Es wird untersucht wie lange eine Schreiboperation benötigt. Um Schwankungen zu eliminieren werden dreimal zehntausend Schreiboperationen gemessen und der arithmetische Durchschnitt berechnet.
- Konsistenzverhalten gegenüber Replikationen: Es wird gemessen, wie lange die Replikation der Dokumente auf weitere Knoten des verteilten Systems benötigt.
- Durability (Dauerhaftigkeit): Es wird untersucht unter welchen Bedingungen ein Datenverlust entsteht und welches Ausmaß dieser einnimmt.

Die einzelnen Tests wurden dabei unter folgenden Bedingungen durchgeführt:

- Normalzustand: Der Cluster (bestehend aus 3 Knoten) wird unter keinen Änderungen der Architektur getestet.
- Erweiterung: Der Cluster wird online um einen weiteren Knoten erweitert, und in Anschluss dessen online wieder um einen Knoten reduziert.
- Systemfehler eines Slaves: Es wird ein abrupter Ausfall eines Slaves simuliert. Dabei wird der Datenbankprozess mittels des Befehls kill -9 (SIGKILL) [40] beendet. Der Prozess wird nach Aufzeichnung der Messergebnisse wieder gestartet. Dieser Test wird beim Couchbase Cluster nicht durchgeführt, da dieser aufgrund der Architektur nicht über einen Slave verfügt.
- Systemfehler eines Masters: Es wird ein Ausfall des Masters simuliert. Dabei wird auch hier der Datenbankprozess mittels des Befehls kill -9 (SIGKILL) [40] beendet. Der Prozess wird nach Aufzeichnung der Messergebnisse wieder gestartet. Beim Couchbase Cluster wird dabei zufällig einer der Master ausgewählt.

5 Testergebnisse

Für einen aussagestarken Vergleich wird als Trade-Off zwischen Performance und Consistency/Durability spezifiziert, dass ein Schreibvorgang dann als erfolgreich gilt sobald dieser im Hauptspeicher ausgeführt wurde. Diese Eigenschaft ist bei MongoDB und Couchbase der Defaultwert und kann bei allen zu untersuchenden Datenbanken spezifiziert werden:

- MongoDB: Write Concern Acknowledged.
- Couchbase: Synchron Write mit PersistTo.Zero und ReplicateTo.Zero.
- CouchDB: Synchron Write mit gesetzten batch=ok Flag.

5.1 MongoDB

Die Messung umfasst das Senden des Requests an den Cluster, auf welchem nach Erhalt die Durchführung der Schreiboperation im Hauptspeicher stattfindet, und von welchem im Anschluss wiederum ein Response an den Client gesendet wird. Sobald der Client den Response mit dem Statuscode der Operation erhalten hat, gilt der Schreibvorgang als durchgeführt und die Messung wird aufgezeichnet. Im Hintergrund repliziert der Cluster die Schreiboperation auf die im Replica Set vorhandenen Slaves und die Daten werden persistiert.

5.1.1 Performance

MongoDB weist unter normalen Konditionen eine Latenzzeit von 0,6 – 0,7 Millisekunden pro Schreiboperation auf.

Wird nun eine bewusste Erweiterung des Replica Sets durch einen Administrator durchgeführt, so erhöhen sich die Latenzzeiten der einzelnen Schreibvorgänge. Dies kann aus Abbildung 13 entnommen werden. Der Graph zeigt zu Beginn die im Normalzustand vorhandene Latenzzeit von 0,6 – 0,7 Millisekunden. Sobald der vierte Knoten dem Cluster hinzugefügt wird, steigen die Latenzzeiten auf bis zu 1,5 ms an. Der Cluster startet dabei im Hintergrund einen zusätzlichen Replizierungsjob zur Verteilung der Daten. Sobald der neu hinzugefügten Knoten denselben Datenbestand wie die restlichen Knoten aufweist beginnen die Latenzzeiten wieder zu fallen und pendeln sich bei 0,8 ms pro Schreiboperation ein. Wird der Knoten anschließend wieder aus dem System genommen, so fallen die Latenzzeiten auf die ursprünglichen Werte zurück, da der Cluster einen Replizierungsjob weniger zu verwalten hat.

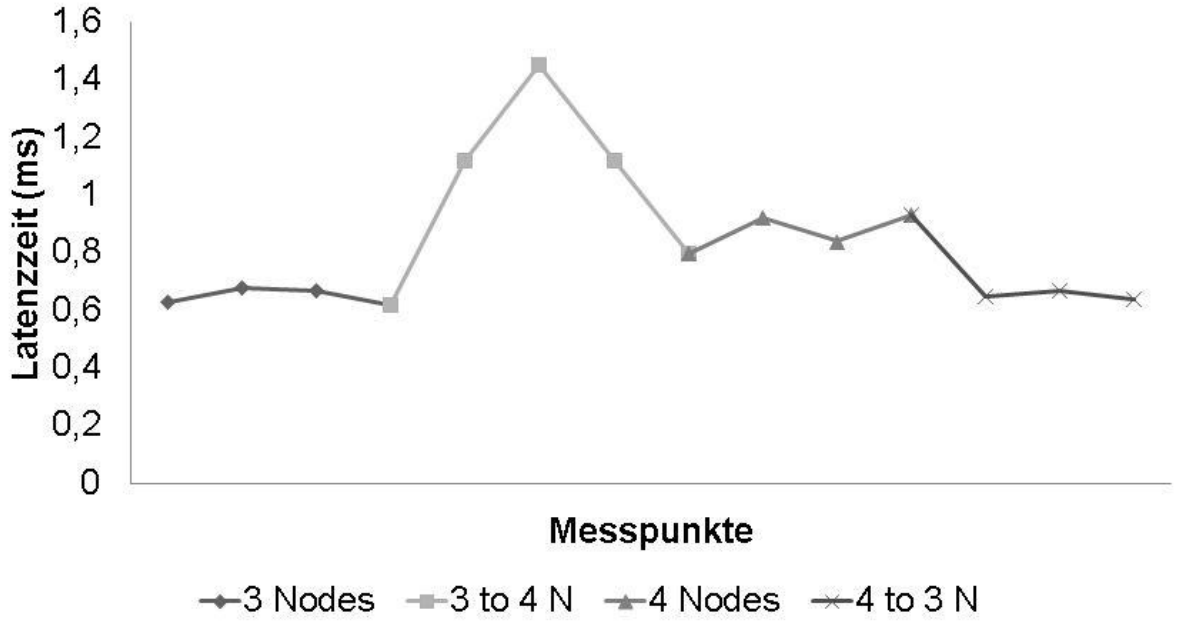


Abbildung 13: MongoDB Erweiterung

Im nächsten Schritt wird untersucht wie sich das Latenzverhalten ändert, wenn es zu einem Systemfehler eines Slaves kommt und dieser für den Cluster nicht mehr erreichbar ist. Abbildung 14 zeigt das Verhalten. Ausgangspunkt ist wieder der Cluster im Normalzustand (und setzt sich somit aus einem Replica Set bestehend aus einem Master und zwei Slaves zusammen). Die Latenzzeiten bewegen sich zwischen 0,6 und 0,7 Millisekunden. Im nächsten Schritt wird der Systemfehler eines Slaves simuliert. Die Latenzzeit steigt dabei auf bis zu 1,5 ms an. In diesem Zeitraum wird seitens des Clusters der Slave als fehlerhaft gekennzeichnet und aus dem Verbund entfernt. Der Cluster besteht damit nur noch aus einem Master und einem Slave. Dadurch sinkt die Latenzzeit auf 0,5 ms. Im letzten Schritt wird der Systemfehler auf dem Slave behoben. Dieser wird nach erfolgreichen starten der Datenbank wieder automatisch dem Cluster hinzugefügt, und es beginnt die Replikation der

Daten, welche zwischenzeitlich eingefügt wurden. Die Latenzzeit steigt dabei im ersten Moment sehr stark an und erreicht einen Wert von 8 ms. In den darauffolgenden Messungen sinkt die Latenzzeit langsam wieder ab und pendelt sich nach erfolgter Synchronisierung wieder zwischen 0,6 und 0,7 ms ein.

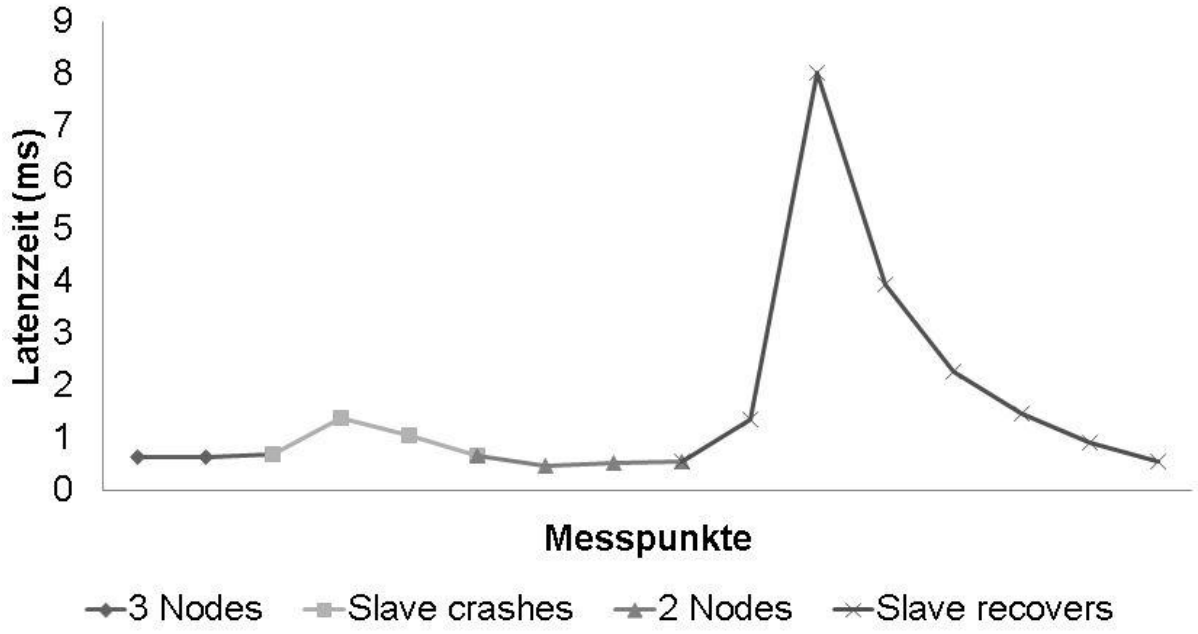


Abbildung 14: MongoDB Systemfehler des Slaves

Der nächste Test zeigt das Verhalten, wenn der Master der verteilten Datenbank ausfällt. Nachdem im vorgehenden Beispiel wieder der Normalzustand hergestellt wurde, wird nun ein Systemfehler am Master Server produziert. Der Performancegraph ist in Abbildung 15 zu sehen. Die Latenzzeiten befinden sich zu Beginn wieder zwischen 0,6 und 0,7 ms. In dem Moment wo es zum Ausfall kommt, steigen die Latenzzeiten theoretisch ins unendliche - dies wird mit dem schwarzen Balken repräsentiert. Bei der praktischen Durchführung kam es zum Abbruch der Operation. Der Cluster selbst benötigt nun 20 Sekunden für den Votingprozess, bei welchen einer der verbliebenen Slaves zum neuen Master gewählt wird. Sobald dieser Prozess durchgeführt wurde, werden die Schreiboperationen fortgeführt. Wie schon im obigen Beispiel pendeln sich auch hier die Latenzzeiten bei nur 2 Knoten auf 0,5 ms ein. Im nächsten Schritt wird der dritte Node wieder gestartet, welcher nun als Slave den Cluster joined. Die Latenzzeiten steigen dabei wieder an und erreichen Werte von 1,5 ms. Diese pendeln sich nach Synchronisation des Deltas (jene Daten, welche in der Zwischenzeit eingefügt wurden) wieder bei einer Latenzzeit von 0,6 -0,7 ms ein.

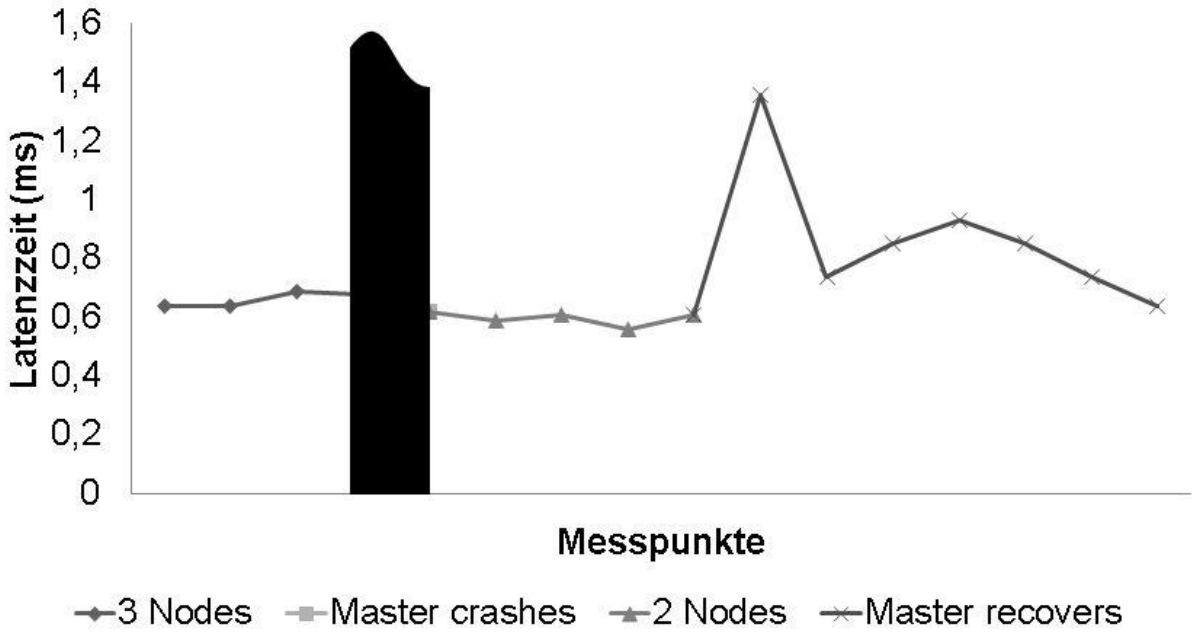


Abbildung 15: MongoDB Systemfehler des Masters

5.1.2 Consistency

Werden Schreiboperationen am Cluster im Normalzustand ausgeführt, so entsteht eine Verzögerung von 2 Sekunden für die Replikation der Daten auf die Replica Server. Während diesem Zeitraum befinden sich die Daten in einem inkonsistenten Zustand. Am Master Server ausgeführte Lesezugriffe werden so stets erfolgreich beantwortet - im Gegensatz dazu kann es aufgrund der Dauer für die Replizierung vorkommen, dass Dokumente noch nicht auf den Slaves vorhanden sind. Trifft dieser Fall ein ist der Lesezugriff nicht erfolgreich.
So kann es passieren, dass ein Client welchem der Erfolg einer Schreiboperation bestätigt wurde, bei erneuter Anforderung des Dokuments einen Fehler zurückbekommt, wenn die erneute Anforderung auf einen der Slaves durchgeführt wird und die Replikation dazu noch nicht abgeschlossen ist. Die Daten erreichen daher eine schlussendliche Konsistenz - sie sind konsistent wenn keine Schreibzugriffe mehr stattfinden und die Replikation vollständig erfolgt ist.
Viel stärker zeigt sich das Verhalten bei einer Erweiterung des Clusters um einen zusätzlichen Knoten. Auf diesen müssen die gesamten Daten der Datenbank repliziert werden, ein Vorgang für welchen der Cluster bei 100000 Dokumenten 2 Minuten und 45 Sekunden benötigt. Entsprechend niedrig sind dabei die erfolgreich auf diesem Knoten stattfindenden Lesevorgänge. Angemerkt sei, dass während der Synchronisation keine Operationen durchgeführt wurden. Um eine derart hohe Anzahl an inkonsistenten Lesevorgängen zu vermeiden, werden in der Praxis die Daten bereits synchronisiert bevor der Node den Cluster hinzugefügt wird und Leseoperationen beantwortet werden.
Steht der Cluster während der Synchronisation der Daten unter Last, so wird das Delta zwischen den Datenbeständen sogar größer. Dieser Vorgang wurde gemessen, nachdem an

einem Slave ein Systemfehler simuliert wurde. Zum Zeitpunkt t1 (umittelbar nachdem auf dem Slave der Systemfehler korrigiert wurde und dieser wieder im Cluster eingehängt wurde) betrug die Dauer für die Synchronisation 655 Sekunden und wird in Beispiel 13 fett gedruckt hervorgehoben.

source: 192.168.122.147:27017
syncedTo: Fri Nov 14 2014 15:39:27 GMT+0100 (CET)
0 secs (0 hrs) behind the primary
source: 192.168.122.148:27017
syncedTo: Fri Nov 14 2014 **15:50:22** GMT+0100 (CET)
-655 secs (-0.18 hrs) behind the primary

Beispiel 13: Status der Synchronisation zum Zeitpunkt t1

Zum Zeitpunkt t2 betrug die Dauer für eine Synchronisierung bereits 712 Sekunden. Die Messung fand eine Minute (57 Sekunden) nach t1 statt.

“rs0”:SECONDARY> db.printSlaveReplicationInfo()
source: 192.168.122.147:27017
syncedTo: Fri Nov 14 2014 15:39:27 GMT+0100 (CET)
0 secs (0 hrs) behind the primary
source: 192.168.122.148:27017
syncedTo: Fri Nov 14 2014 **15:51:19** GMT+0100 (CET)
-712 secs (-0.2 hrs) behind the primary

Beispiel 14: Status der Synchronisation zum Zeitpunkt t2

Das Beispiel zeigt, dass der Cluster unter Last die Daten mit den ihm zur Verfügung stehenden Ressourcen nicht synchronisieren kann und das Delta zwischen den Daten anwächst. Damit verbunden steigt auch der Wert an inkonsistenten Lesevorgängen. Erst nachdem die Schreibvorgänge gestoppt wurden, konnte der Cluster die Daten vollständig synchronisieren. Dieses Beispiel veranschaulicht dabei nochmals das Verhalten von schlussendlicher Konsistenz in verteilten Datenbanken.

5.1.3 Durability

Kommt es zu einem Systemfehler des Master Servers, so wird dieser vom Java Driver clientseitig erkannt und der Schreibvorgang wird für die Dauer des Votingprozesses gestoppt. Sobald einer der verbliebenen Knoten als neuer Master Server gewählt wurde, werden die Schreiboperationen automatisch fortgeführt. Beim Test wurden 30000 Schreiboperationen durchgeführt. Die Datenbank weist nach dem vollständig durchgeführten Test und nach Synchronisation des gestoppten Knoten einen Datenbestand zwischen 29994 und 29997 Dokumenten auf. Es kommt somit zu einer Verletzung der Durability Eigenschaft der Datenbank. Die Dokumente, welche auf einen erwarteten Datenbestand von 30000 Dokumenten fehlen, konnten aufgrund des herbeigeführten Systemfehlers nicht persistiert

oder repliziert werden, wurden aber den Client gegenüber als erfolgreich durchgeführt gemeldet. Dieses Beispiel adressiert genau den Trade Off zwischen Performance und Durability und zeigt die Auswirkung der für diesen Test getroffene Entscheidung, den Write Concern Acknowledged zu verwenden und dabei das Risiko einzugehen alle Daten zu verlieren die sich nur im Hauptspeicher befanden.

5.2 Couchbase

Das angewendete Messverfahren entspricht dem der MongoDB und umfasst somit die Zeit vom Senden des Requests bis zum Empfang des Responses, wobei der Response vom Cluster erst verschickt wird, sobald die Operation im Hauptspeicher durchgeführt wurde. Die Replizierung und das Persistieren erfolgt auch hier im Anschluss transparent im Hintergrund.

5.2.1 Performance

Im Normalzustand, welcher bei den Tests aus 3 aktiven Knoten besteht, beträgt die Latenzzeit Werte zwischen 0,6 und 0,7 ms und ist damit in der gleichen Performanceklasse wie die MongoDB.
Wird dem Cluster ein weiterer Knoten hinzugefügt, so erhöhen sich vorerst die Latenzwerte auf bis zu 1 ms. Die Werte bleiben während der Datensynchronisierung auf diesem Niveau. Während dieses Zeitfensters werden jedoch nicht nur die Daten synchronisiert, sondern auch die Zuständigkeiten für die einzelnen Datenbereiche (vBuckets) neu verteilt. Nachdem dies erfolgt ist sinken die Latenzzeiten im Gegensatz zur MongoDB wieder auf den Ausgangswert von 0,6 - 0,7 ms ab. Der Grund dazu liegt in der Architektur. Während MongoDB um einen zusätzlichen Slave erweitert wurde und somit jeden Datensatz dreimal anstatt zweimal replizieren muss, wurde die Couchbase um einen Masterserver erweitert. Dieser nimmt wie die bisherigen Knoten Schreiboperationen für einen bestimmten Bereich der Datenbank entgegen. Die Daten innerhalb des Clusters werden weiterhin nur zweimal repliziert.
Wird der hinzugefügte Knoten wieder entfernt, so steigen die Latenzzeiten erneut auf Werte von bis zu 1ms. Das Verhalten der Latenzzeiten ist dabei synchron der Latenzzeiten bei der Erweiterung. Dies ist auf die erneute Verschiebung der Zuständigkeitsbereiche auf die im Cluster verbleibenden Server zurückzuführen. Auch hier kann der Unterschied zur MongoDB festgestellt werden, bei welchem der Node ohne steigende Latenzzeiten entfernt werden kann, da sich auf diesem nur Replikate befinden.

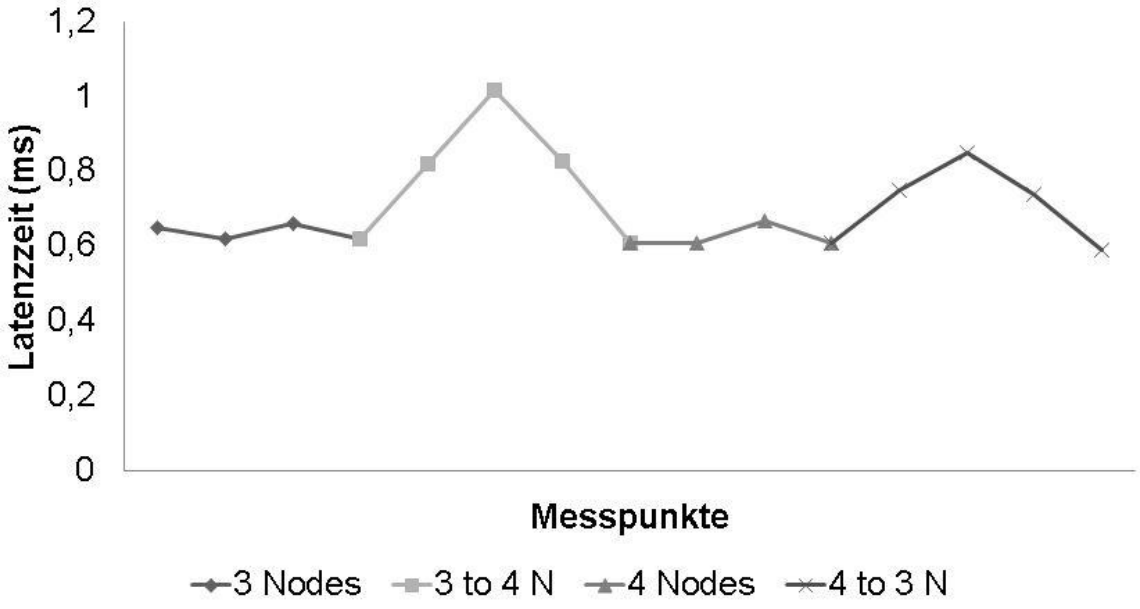


Abbildung 16: Couchbase Erweiterung

Kommt es bei der Couchbase zum Ausfall eines Master Servers, so können weiterhin 2/3 aller Schreibvorgänge erfolgreich durchgeführt werden. All jene Schreibvorgänge jedoch, denen der Datenbereich dem ausgefallenen Server zugeordnet ist, können nicht mehr ausgeführt werden. Dies wird in Abbildung 16 durch den grau hinterlegten Balken dargestellt. Des Weiteren erhöht sich die Latenzzeit der erfolgreich durchgeführten Schreiboperationen auf 7ms. Bis ein Failover für die nun nicht erreichbaren Datenbereiche eingeleitet wird vergehen 60 Sekunden. Danach werden alle Schreibvorgänge wieder fortgesetzt.

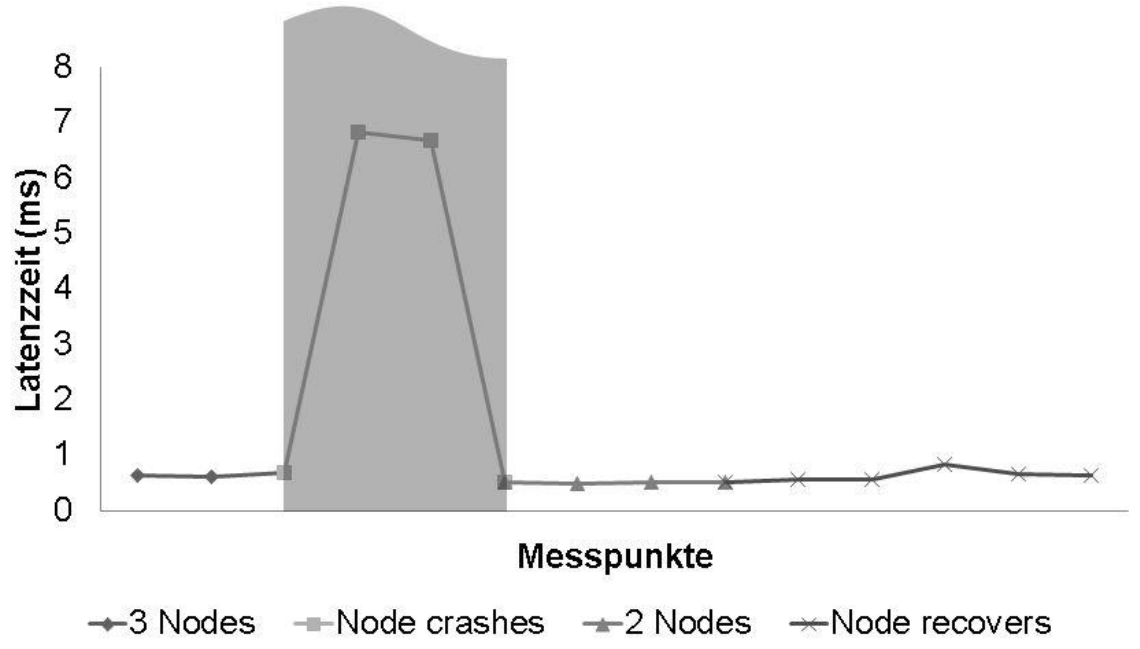


Abbildung 17: Couchbase Systemfehler

Die Latenzzeiten sind dabei wieder gleich hoch wie im Normalzustand und betragen Werte zwischen 0,6 und 0,7 ms. Wird der fehlerhafte Knoten wieder dem Cluster hinzugefügt, so steigen erneut aufgrund der Verteilung der Datenbereiche die Latenzzeiten auf eine Millisekunde, und fallen danach wieder auf die Ausgangswerte ab.

5.2.2 Consistency

Ähnlich der MongoDB kommt es auch bei der Couchbase zu einer Inkonsistenz der Daten zwischen den Servern aufgrund eines Delays verursacht durch den Replizierungsprozess. Die LagTime umfasst auch hier einen Wert von 2 Sekunden. Das Delta zwischen den Master und den Replikat nimmt dabei ein Ausmaß von bis zu zehn Dokumenten ein.
Wird der Cluster um einen weiteren Node erweitert, so müssen die Daten neu verteilt werden – dieser Prozess wird bei Couchbase als Rebalancing bezeichnet. Dieser Prozess nimmt, sofern der Server nicht unter Last steht, für 10 000 Dokumente 5 Minuten ein. Dabei werden auch die Zuständigkeitsbereiche der einzelnen vBuckets zwischen den Knoten neu verteilt. Wenn der Server während der Verteilung unter Last steht erhöht sich die Dauer des Vorgangs von 5 auf 20 Minuten für 10 000 Dokumente. Beispiel 15 zeigt die entsprechenden Ausschnitte von je einem Durchlauf:

- Ohne Last (auf den Server werden keine Operationen ausgeführt):
  Rebalance completed successfully. **11:16:44** - Wed Dec 17, 2014
  Started rebalancing bucket testBucket **11:11:30** – Wed Dec 17, 2014
- Unter Last (es werden Schreiboperationen ausgeführt):
  Rebalance completed successfully. **11:39:10** - Wed Dec 17, 2014
  Started rebalancing bucket testBucket **11:20:24** - Wed Dec 17, 2014

Beispiel 15: Logeinträge der Datenverteilung

Entgegen der MongoDB, bei welcher unter Last das Delta immer größer wird, können bei der Couchbase die Daten abgeglichen und neu verteilt werden. Der Zeitrahmen dafür beträgt aber das Vierfache gegenüber des Datenabgleichs und der Verteilung wenn die Knoten nicht unter Last stehen. Während des Rebalancing finden keine Schreib- und Leseoperationen auf dem neu hinzugefügten Knoten statt. Es besteht bei der Couchbase daher nur bei aktiv stattfindenden Schreibvorgängen die Gefahr, inkonsistente Daten zu lesen.

5.2.3 Durability

Beim Ausfall eines Servers wird ein Votingprozess eingeleitet, welcher innerhalb von 60 Sekunden einen neuen Master für die betroffenen vBuckets bestimmt. Der Ausfall des Servers wird dabei clientseitig durch den Java Driver erkannt. Dieser löst eine Runtime-Exception aus, versucht aber entgegen des Verhaltens bei MongoDB welcher alle Schreibvorgänge während des Votingprozesses blockiert, weiterhin Dokumente auf den fehlenden Node einzufügen. Dabei werden im Mittel während den 60 Sekunden 25

Schreiboperationen ausgeführt, welche der Anwendung als fehlerhaft zurückgemeldet werden. Diese Schreibvorgänge müssen seitens der Client Applikation nach erfolgreichen Failover des Masters erneut ausgeführt werden. Das entsprechende Error Handling dazu wurde in der für diese Tests verwendeten Software nicht implementiert. Es ergibt sich daher folgender Bestand in der Datenbank:

- 29 965 - 29 975 Dokumente von 30 000 wurden erfolgreich geschrieben
- 15 - 25 Dokumente konnten währen des Votingprozesses nicht geschrieben werden, wurden aber an die Applikation zurückgemeldet, wo durch entsprechende Logik die Fehler abgefangen und erneut gegen die Datenbank geschrieben werden können.
- 5 - 10 Dokumente wurden nicht persistiert, wurden aber den Client als erfolgreich durchgeführt gemeldet, da die Operationen im Hauptspeicher ausgeführt wurden. Es kommt zum Datenverlust, welcher auf dem gewählten Trade-Off zwischen Performance und Durability (PeristTo.Zero, ReplicateTo.Zero) zurückzuführen ist.

5.3 CouchDB

Die Messung der Latenzzeiten umfasst bei der CouchDB einerseits den Aufbau der HTTP-Verbindung, das Senden des POST-Requests, den Empfang des Responses als auch den Abbau der HTTP-Verbindung. Seitens der Datenbank wird der Response nach Durchführung der Operation im Hauptspeicher gesendet. Die Replizierung und das Persistieren erfolgt damit wie schon bei der MongoDB und der Couchbase parallel im Hintergrund und trägt nicht zur Latenzzeit bei.

5.3.1 Performance

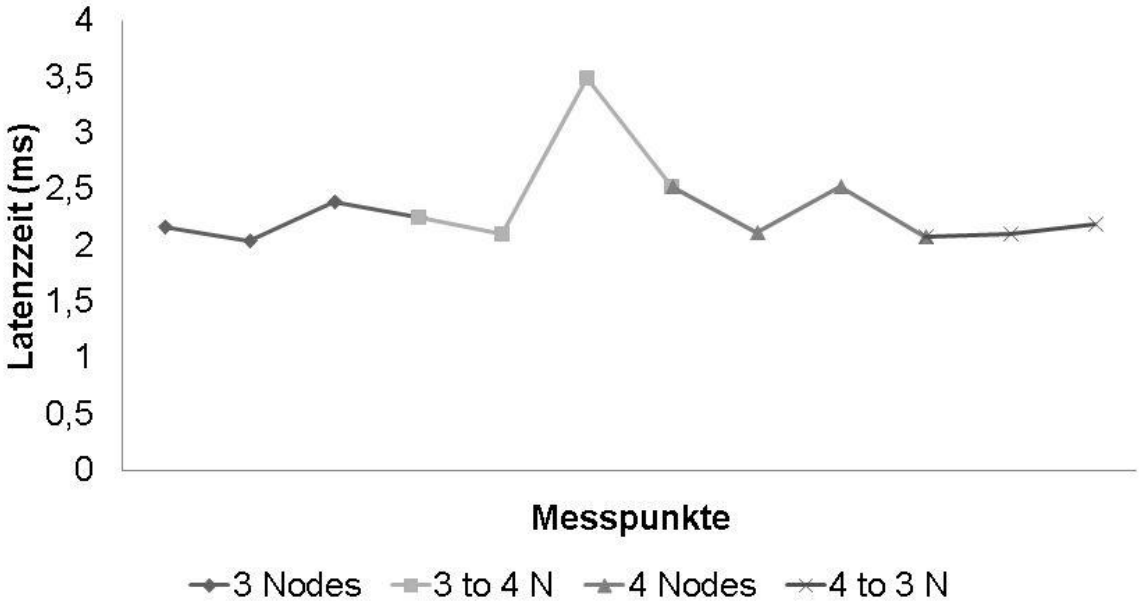


Abbildung 18: CouchDB Erweiterung

Bei der CouchDB beträgt die Latenzzeit im Normalzustand einen Wert zwischen zwischen 2 und 2,4 ms und ist damit deutlich höher als der Wert bei MongoDB und Couchbase.
Wird der Cluster um einen Knoten erweitert, so steigt die Latenzzeit auf 3,5 ms an. Dieser Wert fällt gegen Ende der Datensynchronisation wieder ab und pendelt sich zwischen 2 und 2,5 ms ein. Eine Erhöhung der Latenzzeiten ist trotz eines zusätzlichen Replikationsjobs im 4 Node Cluster nicht zu erkennen. Die Latenzzeiten bleiben auch dann auf den gleichen Niveau wenn der Knoten wieder entfernt und auf seine ursprünglichen 3 Nodes reduziert wird. Abbildung 18 zeigt den Verlauf.
Wird an einem Slave ein Systemfehler simuliert, so fällt die Latenzzeit von über 2ms auf 1,7ms ab. Der Cluster kann somit durch die Reduzierung von zwei auf einen Replikationsprozess schnellere Responses liefern. Wird der Slave wieder den Cluster hinzugefügt, so steigen die Latenzzeiten wieder auf die ursprünglichen Werte an. Der gesamte Verlauf ist in Abbildung 19 nachzuvollziehen.

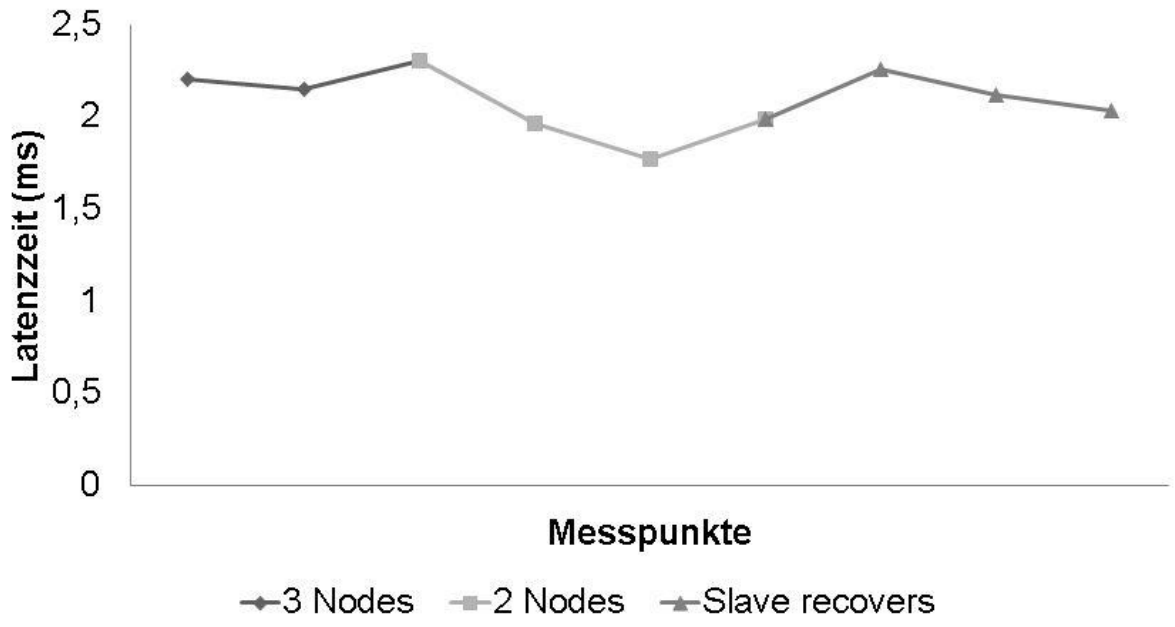


Abbildung 19: CouchDB Systemfehler des Slaves

Kommt es am Masterserver zu einem Systemfehler, so können die Schreibvorgänge nicht mehr erfolgen. Im Gegensatz zur MongoDB und zur Couchbase gibt es bei der CouchDB keinen Mechanismus welcher einen Failover des defekten Masters vornimmt. Da zusätzlich die Adressen clientseitig hardcodiert im Programm hinterlegt sind, werden die Schreibvorgänge auch nicht automatisch an einem anderen Server fortgesetzt. Hierfür wäre eine implementierte Logik notwendig (Es sei an dieser Stelle nochmals auf Anhang B verwiesen, welcher Projekte vorstellt die sich mit dieser Problematik auseinandersetzen). Die Latenzzeiten steigen somit theoretisch ins unendliche. In der Praxis gibt es nach 60 Sekunden eine Timeout-Exception seitens der verwendeten HTTP-Library.

Die Schreibvorgänge werden in diesem Beispiel erst fortgesetzt, nachdem der Master neu gestartet wurde und wieder erreichbar ist. Die Latenzzeiten nehmen dabei wieder Zeiten zwischen 2 und 2,5 ms ein und sind somit gleich den Latenzzeiten zu Beginn des Tests.

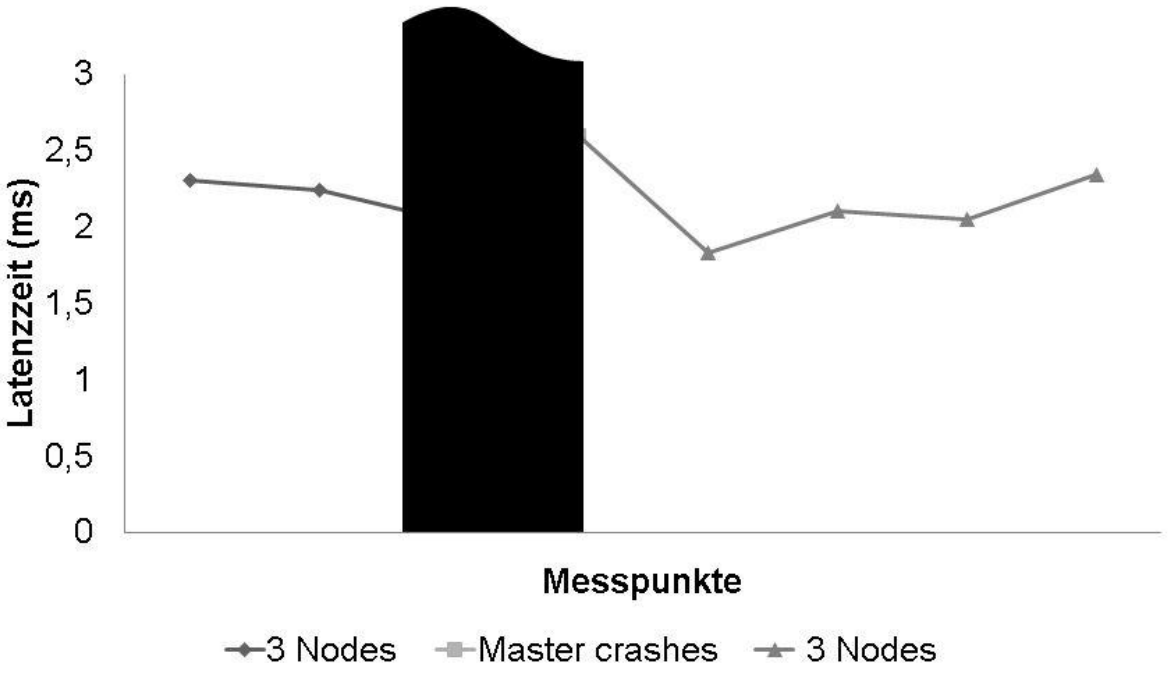


Abbildung 20: CouchDB Systemfehler des Masters

5.3.2 Consistency

Im Gegensatz zur MongoDB und zur Couchbase laufen die Daten zwischen Master und Replica Server bei CouchDB unter Dauerlast (fortlaufende Schreiboperationen) auch im Normalzustand auseinander. Damit nimmt die Anzahl an Dokumenten, welche nicht konsistent sind, mit fortlaufendem Betrieb des Clusters zu. Erst wenn die Last am Cluster reduziert wird, kann wieder ein konsistenter Datenbestand hergestellt werden. Abbildung 21 stellt das Prinzip eventueller Konsistenz grafisch dar. Die Grafik veranschaulicht dabei nicht nur das Delta zwischen Master und Replica Server im Normalzustand, sondern auch das Delta wenn der Cluster aus vier Knoten (3 Replica) oder nur aus zwei Knoten (1 Replica) besteht. Der Prozess ist dabei umso langsamer, je mehr Replizierungsjobs laufen. In der Grafik ist dabei zu erkennen, dass 30 000 Dokumente innerhalb von 60 Sekunden in die Datenbank geschrieben werden, die Anzahl der Dokumente aber je nach Replizierungsgrad zu diesem Zeitpunkt erst zwischen 5000 und 10 000 Dokumenten betragen. Sobald der Server nicht mehr unter Last steht (nach Beendigung des Schreibvorgangs) nimmt die Geschwindigkeit der Datenreplizierung um ein vielfaches zu, und es kann innerhalb einiger Sekunden ein über allen Knoten hinweg konsistenter Datenbestand hergestellt werden.

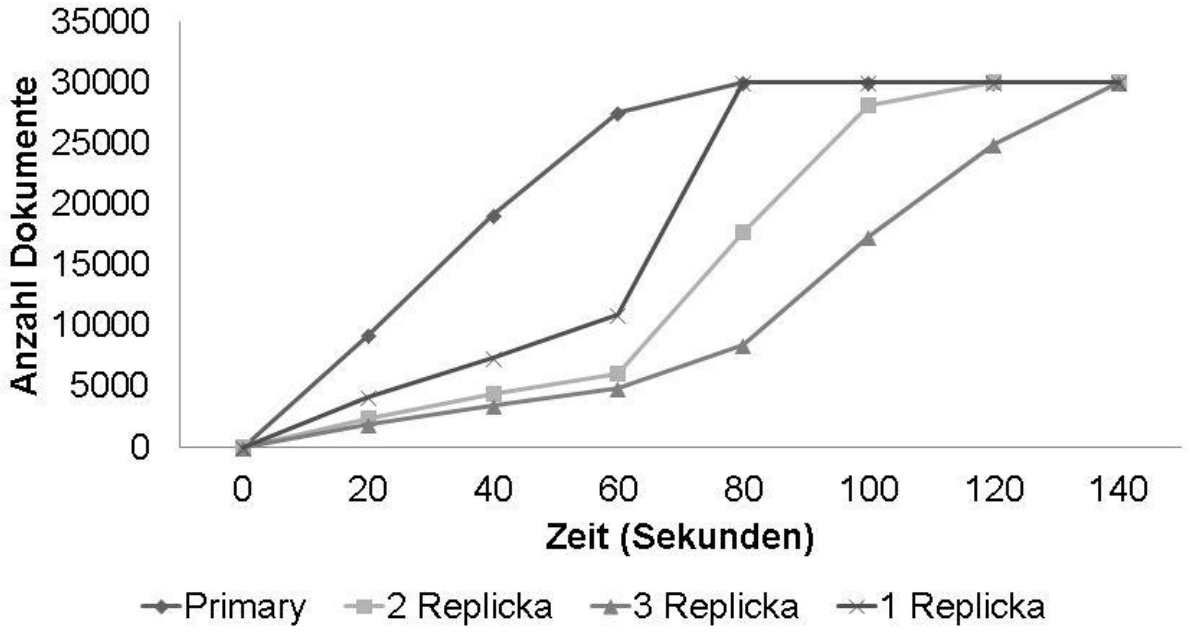


Abbildung 21: CouchDB Consistency Verhalten

5.3.3 Durability

Kommt es zum Ausfall des Masters, so können auf dem Cluster bei fehlender Logik auf Clientseite keine Dokumente gespeichert werden. Wurden gerade Schreiboperationen durchgeführt, so bleiben die beiden Slaves solange in einem inkonsistenten Zustand, bis der Master Server wieder erfolgreich den Cluster joined, und die Replikation der Daten fortsetzt. Es gehen dabei jedoch alle Dokumente verloren, welche zum Zeitpunkt des Systemfehlers weder repliziert noch persistiert waren. Das Ausmaß nimmt dabei einen Wert zwischen 400 und 600 Dokumenten an, und ist damit deutlich größer als bei MongoDB und Couchbase. Kann der Rechner aufgrund seines Defekts nicht mehr gestartet werden oder sind die Daten aufgrund eines Hardwaredefekts nicht mehr lesbar, so nimmt der Datenverlust um all jene Dokumente zu, welche zwar persistiert aber noch nicht repliziert wurden. In Abbildung 21 ist ersichtlich, das bei einem vier Node Cluster mit 3 Replica Servern nach 60 Sekunden erst 5000 Dokumente repliziert wurden. Das würde bedeuten, dass es in dieser Konstellation zu einen Datenverlust von 25000 Dokumenten kommen würde, was 83.3% aller geschriebenen Dokumente bedeuten würde.

5.4 Gegenüberstellung

Dieses Kapitel zeigt eine Gegenüberstellung der gemessenen Werte in Tabellenform.

5.4.1 Performance

Verglichen werden die Latenzzeiten unter den Ausgangszustand (Normal), bei Erweiterung durch einen zusätzlichen Node (Erweiterung), nach durchgeführter Erweiterung - sprich nachdem die Daten synchronisiert wurden (4Nodes), bei Reduzierung des Clusters auf 3 Knoten (Reduzierung), bei Systemcrash eines Slaves (Crash), bei 2 aktiven Knoten (2Nodes) und bei erneuten hinzufügen des Slaves (Recovery). Alle Werte wurden in Millisekunden angegeben.

DBMS	Normal	Erweiterung	4Nodes	Reduzierung	Crash	2Nodes	Recovery
MongoDB	0,65	1,45	0,91	0,65	1,42	0,50	8,03
Couchbase	0,65	1,0	0,67	0,92	6,87	0,53	0,85
CouchDB	2,04	3,5	2,53	2,19	2,31	1,77	2,26

Tabelle 2: Vergleich der Latenzzeiten.

5.4.2 Consistency

Folgende Tabelle geht auf die unterschiedlichen Konsistenzverhalten der Datenbanken ein. Dabei wird unterschieden zwischen den Verhalten der Konsistenz zwischen den Knoten bei fortlaufenden Schreiboperationen, die benötigte Dauer für die Synchronisierung von 30000 Dokumenten wenn am Cluster keine Schreib- oder Lesevorgänge stattfinden, und die Dauer für eine Synchronisierung bei fortlaufenden Schreiboperationen.

DBMS	Fortlaufende Schreiboperationen	Synchronisierung von 30000 Dokumenten bei keiner Last	Synchronisierung von 30000 Dokumenten unter Last
MongoDB	2 Sec.	45 Sec.	Delta nimmt zu
Couchbase	2 Sec.	15 Minuten (Rebalancing)	60 Minuten (Rebalancing)
CouchDB	Delta nimmt zu	25 Sec.	Delta nimmt zu

Tabelle 3: Vergleich des Konsistenzverhaltens.

5.4.3 Durability

Vergleich der Durability Eigenschaften bei Systemcrash des Masters. Eingegangen wird auf die von der Datenbank durchgeführten Aktion wenn ein Master Server ausfällt. Des Weiteren wird die Anzahl an Dokumenten erfasst, welche nicht persistiert wurden und wie sich die Clientapplikation dabei verhalten hat.

DBMS	Failover Szenario	Datenverlust bei Systemfehler am Master	Aktion der Clientapplikation
MongoDB	Neuer Master wird in 20 Sec. gewählt.	3 - 6 Dokumente	Blockt bis neuer Master gewählt ist.
Couchbase	Neuer Master für vBuckets wird in 60 Sec. gewählt.	5 - 10 Dokumente	Versucht weitere Dok. einzufügen, bis neuer Master gewählt ist.
CouchDB	Kein Failover möglich.	400 - 600 Dokumente	Wartet bis der Server wieder verfügbar ist.

Tabelle 4: Vergleich der Durability Eigenschaften

6 Related Work

Vergleiche und Benchmarks in ähnlichen Arbeiten beschäftigen sich ebenso mit Antwortszeiten und Durchsatzverhalten, gehen aber nicht oder nur am Rande auf Datenverlust und Konsistenzverhalten ein. In [41] und [44] werden Performancemessungen durchgeführt, [42] testet das Verhalten von Query Statements und [43] führt einen theoretischen Vergleich zwischen NoSQL Systemen durch.

In [41] und [44] wird für den Vergleich der DBMS der YCSB (Yahoo! Cloud Serving Benchmark) verwendet, welches speziell für den Vergleich von NoSQL Systemen entwickelt wurde. Das Framework soll dabei das einheitliche Testen von den einzelnen DBMS ermöglichen, welche sich oft in ihrer Architektur grundsätzlich unterscheiden. YCSB verwendet dafür im Kern einen Workload Generator. Dieser wird mittels eines Moduls, welches speziell für die jeweilige Datenbank entwickelt wird, angesprochen. [45] gibt eine Übersicht über die unterstützten DBMS.

[41] behandelt in seiner Gegenüberstellung Cassandra, HBASE, Yahoo!‘s PNUTS und MySQL. Die Arbeit beschreibt zu treffende Trade-Offs sowohl für Schreib- als auch für Lesevorgänge durch Vergleich von Latenz und Durability, Synchroner und Asynchroner Replizierung und Datenpartitionierung und führt im Anschluss entsprechende Benchmarks durch. Dabei wird das Verhalten bei Insert, Update, Read und Scanvorgängen erfasst. Die Ergebnisse werden anschließend beschrieben und grafisch dargestellt.

[44] praktiziert seine Tests in der Amazon Cloud und testet dabei die NoSQL DBMS Cassandra, HBase und MongoDB. Die Arbeit beschreibt die Konfiguration der Systeme und erläutert die mittels YCSB durchgeführten Workload Tests. Es wird die Architektur ausgehend von einem Datenknoten fortlaufend bis zu einem System mit 32 Knoten jeweils verdoppelt. Dabei wird ebenfalls die Anzahl der Clients erhöht. Die Ergebnisse der Durchsatztests werden anschließend grafisch und tabellarisch veranschaulicht.

[42] führt Benchmarks an den beiden Document Stores MongoDB und CouchDB durch und verwendet für die Tests eine eigens entwickelte und der Arbeit beiliegende Software. Die Arbeit testet die Performance der Datenbanken durch Ausführung von Queries, und verwendet hierfür automatisch generierte Daten. Nach einer allgemeinen Beschreibung von NoSQL Systemen werden die einzelnen Queries beschrieben und die Ausführungszeiten der Tests gemessen und aufgezeichnet. Im Anschluss werden die Latenzzeiten anhand von Grafiken beschrieben, und es wird darauf Bezug genommen, wieviel Speicherkapazität die beiden DBMS für die Verwaltung der Daten beanspruchen.

[43] beschreibt eine Gegenüberstellung von Amazon’s SimpleDB und Google’s Bigtable. Es wird auf die Charakteristika, Architektur und auf die Vorteile als auch auf die Limitationen der DBMS eingegangen. Der Fokus des Vergleiches liegt dabei auf der Anwendung der Datenbanken für Cloud Computing. Die unterschiedlichen Eigenschaften der Datenbanken werden dabei tabellarisch dargestellt.

7 Zusammenfassung

Diese Arbeit führte einen Vergleich von NoSQL DBMS durch und verglich dabei die Eigenschaften Performance, Durability und Consistency. Die Tests wurden mittels einer in Java programmierten Software auf den ausgewählten Datenbanken MongoDB, Couchbase und CouchDB durchgeführt. Die erste Messung wurde im Normalzustand durchgeführt, bei welchen die Cluster aus jeweils drei Nodes bestanden. Die zweite Messung wurde bei Erweiterung des Clusters um einen weiteren Node vorgenommen. Dabei wurde festgehalten, wie sich das System nach der durchgeführten Erweiterung verhält, ehe es wieder um einen Node reduziert wurde. Die darauffolgenden Tests befassten sich mit simulierten Systemfehler auf einzelnen Knoten. Dabei wurde einerseits der plötzliche Ausfall des Replika-Servers als auch der des Master-Servers untersucht. Jeder für die Performance ermittelte Messpunkt repräsentiert dabei den arithmetischen Mittelwert von drei mal zehntausend Schreibvorgängen. Die Latenzzeiten der MongoDB und Couchbase waren dabei auf gleichen Niveau, in starken Gegensatz zur CouchDB welche deutlich höhere Latenzzeiten aufwies. Für die Messungen der Konsistenz wurde die Latenzzeit der Replikation festgehalten, während desser sich die einzelnen Nodes des Clusters in einen inkonsistenten Zustand befinden. Weiters wurde ermittelt, wieviele Dokumente nach Abschluss von 30000 seriellen Schreiboperation noch nicht repliziert waren. Während bei CouchDB bereits im Normalzustand eine Synchronisierung der Dokumente unter fortlaufender Schreibvorgänge nicht mehr möglich war, blieb bei MongoDB und Couchbase mit einer Latenzzeit von zwei Sekunden das Delta konstant. Des Weiteren konnten bei der Couchbase auch nach einen simulierten Systemfehler eines Knoten die restlichen Knoten unter Last synchronisiert werden, was sowohl bei der CouchDB als auch bei der MongoDB nicht mehr möglich war. Hinsichtlich Datenverlust verhielten sich alle Datenbanken gleich. Dieser trat nur ein, wenn am Master Server ein Systemfehler simuliert wurde. Dabei gingen jene Dokumente verloren, welche bereits im Hauptspeicher ausgeführt aber noch nicht persistiert oder repliziert wurden. Die Anzahl an nicht persistierten Dokumenten war dabei bei CouchDB ungleich höher als jener bei MongoDB oder Couchbase.

7.1 Future Work

Vorliegende Arbeit untersuchte die zu diesem Zeitpunkt populärsten Document Stores, jedoch befindet sich dieses Themenfeld derzeit noch in einer schnelllebigen Entwicklung. Gegenwärtig gibt es dreißig bis vierzig verschiedene Implementationen von JSON basierten Document Stores, welche sowohl von der OpenSource Gemeinde als auch von kommerziellen Unternehmen permanent weiterentwickelt werden. Die Anzahl der verfügbaren Systeme ist tendenziell steigend und die Verbreitung der jeweiligen Datenbank unterliegt starken Schwankungen. Die in dieser Arbeit verwendete Software bietet damit Potenzial, Schnittstellen für weitere Systeme zu erstellen und die dargelegten Ergebnisse zu erweitern. Des Weiteren befindet sich das YCSB Framework in permanenter Weiterentwicklung, wobei gegenwärtig nur Schnittstellen für die beiden Document Stores MongoDB und CouchDB vorhanden sind und damit ebenso Potenzial für Erweiterungen bietet.

Literaturverzeichnis

[1] Martin Fowler, Pramod J. Sadalage. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. ISBN: 978-0321826626, Addison-Wesley Professional (2012).

[2] Dan McCreary, Ann Kelly. Making Sense of NoSQL: A Guide for Managers and the Rest of Us. ISBN: 978-1617291074, Manning Pubn (2013).

[3] MC Brown. Developing with Couchbase Server. O'Reilly Media (2013)

[4] David Ostrovsky, Yaniv Rodenski. Pro Couchbase Server. ISBN: 978-1430266136, Apress (2014)

[5] WhyNoSQL. http://info.couchbase.com/rs/northscale/images/whyNoSQL.whitePaper.FINAL.pdf. (Letzter Zugriff am 20.12.2014)

[6] Robin Hecht, Stefan Jablonski. NoSQL Evaluation - A Use Case Oriented Survey. http://rogerking.me/wp-content/uploads/2012/03/DatabaseSystemsPaper.pdf. (Letzter Zugriff am 20.12.2014)

[7] Charakteristika und Vergleich von SQL- und NoSQL- Datenbanken. http://dbs.uni-leipzig.de/file/seminar_1112_tran_ausarbeitung.pdf. (Letzter Zugriff am 22.12.2014)

[8] Column family databases. http://curah.microsoft.com/58515/column-family-databases. (Letzter Zugriff am 22.12.2014)

[9] J. Chris Anderson, Jan Lehnardt, Noah Slater. CouchDB: The Definitive Guide. ISBN: 978-0596155896, O'Reilly & Associates (2010)

[10] DB-Engines Ranking von Graph DBMS. http://db-engines.com/de/ranking/graph+dbms. (Letzter Zugriff am 23.12.2014)

[11] E. Brewer. CAP twelve years later: How the "rules" have changed. IEEE Computer 45(2): 23 - 29 (2012)

[12] Daniel J. Abadi. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. IEEE Computer 45(2): 37-42 (2012)

[13] Innocent Mapanga, Prudence Kadebu. Database Management Systems: A NoSQL Analysis. International Journal of Modern Communication Technologies & Research 1(7) (2013)

[14] Charles Roe. The Question of Database Transaction Procesing: An ACID, BASE, NoSQL Primer. DataVersity (2013)

[15] Seth Gilbert, Nancy A. Lynch. Perspectives on the CAP Theorem. IEEE Computer 45(2): 30-36 (2012)

[16] Sibo Mohammad, Sebastian Breß, Eike Schallehn. Cloud Data Management: A Short Overview and Comparison of Current Approaches. http://ceur-ws.org/Vol-850/paper_mohammad.pdf. (Letzter Zugriff am 26.12.2014)

[17] Armando Fox, Eric A. Brewer. Harvest, Yield, and Scalable Tolerant Systems. https://github.com/sholiday/papers/blob/master/Distributed%20Foundations/CAP%20-%20Harvest%20Yield%20and%20Scalable%20Tolerant%20Systems.pdf. (Letzter Zugriff am 26.12.2014)

[18] Henry Robinson. CAP Confusion: Problems with 'partition tolerance'. http://blog.cloudera.com/blog/2010/04/cap-confusion-problems-with-partition-tolerance/. (Letzter Zugriff am 26.12.2014)

[19] Divyakant Agrawal, Sudipto Das, Amr El Abbadi. Data Management in the Cloud: Challenges and Opportunities (Synthesis Lectures on Data Management). ISBN: 978-1608459247. Morgan & Claypool Publishers (19. Dezember 2012)

[20] Andrew S. Tanenbaum, Maarten van Steen. Distributed Systems: Principle and Paradigms. ISBN: 978-0136135531. Prentice Hall; Auflage: 2nd rev. ed. (23. Oktober 2006)

[21] E. Dede, M.Govindaraju, D.Gunter, R. Canon, L. Ramakrishnan. Performance Evaluation of a MongoDB and Hadoop Platform for Scientific Data Analysis. Science Cloud '13 Proceedings of the 4th ACM workshop on Scientific cloud computing: 13-20 (2013)

[22] BSON, http://bsonspec.org (Letzter Zugriff am 27.12.2014)

[23] MongoDB Architecture Guide, http://info.mongodb.com/rs/mongodb/images/MongoDB_Architecture_Guide.pdf (Letzter Zugruff am 27.12.2014)

[24] MongoDB Glossary, http://docs.mongodb.org/manual/reference/glossary/ (Letzter Zugriff am 27.12.2014)

[25] Write Concern Reference, http://docs.mongodb.org/manual/reference/write-concern (Letzter Zugriff am 27.12.2014)

[26] Getting Started with Java Driver, http://docs.mongodb.org/ecosystem/tutorial/getting-started-with-java-driver/#getting-started-with-java-driver (Letzter Zugriff am 27.12.2014)

[27] Couchbase Server Architecture and Capabilities, http://www.couchbase.com/nosql-databases/about-couchbase-server (Letzter Zugriff am 27.12.2014)

[28] Couchbase vs. Apache CouchDB, http://www.couchbase.com/couchbase-vs-couchdb (Letzter Zugriff am 27.12.2014)

[29] About Couchbase, http://www.couchbase.com/about (Letzter Zugriff am 27.12.2014)

[30] Couchbase Server Under the Hood - An Architectural Overview, http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/Couchbase_Server_Architecture_Review.pdf (Letzter Zugriff am 27.12.2014)

[31] Enum PersistTo, http://www.couchbase.com/autodocs/couchbase-java-client-1.0.3/net/spy/memcached/PersistTo.html (Letzter Zugriff am 27.12.2014)

[32] Enum ReplicateTo, http://www.couchbase.com/autodocs/couchbase-java-client-1.0.3/net/spy/memcached/ReplicateTo.html (Letzter Zugriff am 27.12.2014)

[33] .NET SDK, http://docs.couchbase.com/couchbase-sdk-net-1.3/ (Letzter Zugriff am 27.12.2014)

[34] CouchDB – A Database for the Web, http://couchdb.apache.org (Letzter Zugriff am 27.12.2014)

[35] Fsync(2) – Linux Man Page, http://linux.die.net/man/2/fsync (Letzter Zugriff am 28.12.2014)

[36] Fujitsu Primergy Server Raid Controller Performance White Paper, http://globalsp.ts.fujitsu.com/dmsp/Publications/public/wp-raid-controller-performance-ww-en.pdf (Letzter Zugriff am 28.12.2014)

[37] CouchDB API Basics, http://docs.couchdb.org/en/latest/api/basics.html (Letzter Zugriff am 28.12.2014)

[38] Tcp(7) – Linux Man Page, http://linux.die.net/man/7/tcp (Letzter Zugriff am 28.12.2014)

[39] Changes Feed, https://couchdb.readthedocs.org/en/1.4.x/changes.html (Letzter Zugriff am 28.12.2014)

[40] Linux Programmer's Manual – Signal (7), ist bei MongoDB und Couchbase der Defaultwert (Letzter Zugriff am 29.12.2014)

[41] Brian F.Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, Russel Sears. Benchmarking Cloud Serving Systems with YCSB, SoCC: 143-154 (2010)

[42] Robin Henricsson. Document Oriented NoSQL Databases – A comparison of performance in MongoDB and CouchDB using a Pyhton interface. http://www.bth.se/fou/cuppsats.nsf/all/32737dee280f07ddc12578b200454a24?OpenDocument (Letzter Zugriff am 31.12.2014)

[43] Shalini Ramanathan, Savita Goel, Subramanian Alagumalai. Comparison of Cloud Database: Amazon's SimpleDB and Google's Bigtable. ReTIS (2011)

[44] Benchmarking Top NoSQL Databases – A Performance Comparison for Architects and IT Managers, http://www.datastax.com/wp-content/uploads/2013/02/WP-Benchmarking-Top-NoSQL-Databases.pdf (Letzter Zugriff am 31.12.2014)

[45] Yahoo! Cloud Serving Benchmark, https://github.com/brianfrankcooper/YCSB/wiki (Letzter Zugriff am 31.12.2014)

[46] CAP Theorem: Explained. http://robertgreiner.com/2014/06/cap-theorem-explained/ (Letzter Zugriff am 8.1.2014)

[47] MongoDB Architecture. http://siliconangle.com/blog/2013/08/21/10gen-boosts-hadoop-ecosystem-with-upgraded-connector/mongodb-architecture/ (Letzter Zugriff am 8.1.2014)

[48] DB Ranking. http://db-engines.com/en/ranking (Letzter Zugriff am 9.1.2014)

Abbildungsverzeichnis

Tabellenverzeichnis

Beispielverzeichnis

Abkürzungsverzeichnis

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BASE	Basic Availability, Soft state, Eventual consistency
BSON	Binary JSON
CAP	Consistency, Availability, Partition Tolerance
CMS	Content Management System
DBMS	Datenbank Management System
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
KVM	Kernel-based Virtual Machine
NoSQL	Not Only SQL
Oracel RAC	Oracle Real Application Clusters
PACELC	Definition: If there is a partition (P), how does the system trade off availability and consistency (A and C); else (E), when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C)?"
QEMU	Quick Emulator
RDBMS	Relational DBMS
REST	Representational state transfer
SDK	Software Development Kit
SQL	Structured Query Language
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Anhang A: Java Test Programm

Der Code wird auf Github verwaltet: https://github.com/ic12b001/DocumentStoreAnalysis
Das Programm wurde vollständig in Java programmiert und umfasst 1030 Zeilen.

Es existieren folgende Klassen:

- Main: Hier werden die Testspezifikationen festgelegt, und die untenstehenden Klassen MongoDB, Couchbase und CouchDB instanziert.
- Result: Helper Klasse für die Klassen MongoDB, Couchbase und CouchDB. Diese wird verwendet um Performancewerte aufzuzeichnen und am Ende ein Messprotokoll auszugeben.
- MongoDB: Diese Klasse spezifiziert die für die MongoDB Datenbank durchzuführenden Tests, und verwendet hierfür den MongoDB Java Driver.
- Couchbase: Spezifiziert die Tests, welche auf der Couchbase Datenbank durchzuführen sind. Die Klasse verwendet hierfür den Couchbase Java Driver.
- CouchDB: Spezifiziert die für die CouchDB Datenbank durchzuführenden Tests. Dabei wird auf die Java HTTP-Library zurückgegriffen und die Operationen mittels REST-Interface durchgeführt.

Anhang B: CouchDB Cluster Frameworks

Dieser Anhang gibt eine Auflistung der bekanntesten und verbreitesten Cluster Frameworks für CouchDB. Allen gemein ist, dass diese derzeit nicht weiter entwickelt werden. Dieser Rückschluss wurde aus der Commit Statistik des jeweiligen Projekts auf Github gezogen.

- BigCouch: Hochverfügbare, fehlertolerante, geclusterte Version von CouchDB
  Homepage:http://bigcouch.cloudant.com/
  Github Project:https://github.com/cloudant/bigcouch
- Lounge: Proxy basierende geclusterte und partitionsbasierendes Framework
  Homepage:http://tilgovi.github.io/couchdb-lounge/
  Github Project: https://github.com/tilgovi/couchdb-lounge
- Pillow: CouchDB Shard Manager
  Homepage: Projekt verfügt über keine Homepage
  Github Project: https://github.com/khellan/Pillow

Anhang C: CouchDB Konflikt Management

Als Demonstrationszweck wird hier zuerst ein Konflikt erzeugt, indem an einem Dokument auf unterschiedlichen Servern unterschiedliche Änderungen gemacht werden. Im Anschluss wird dieser Konflikt durch den Client gelöst. Diese Prozedur wurde in der LinuxShell durchgeführt.

1. Exportieren von Hostnamen als Variablen:
   [markus@localhost ~]$ HOST1=http://192.168.122.10:5984
   [markus@localhost ~]$ HOST2=http://192.168.122.11:5984
2. Anlegen der Source Datenbank:
   [markus@localhost ~]$ curl -X PUT $HOST1/db → {"ok":true}
3. Anlegen der Replica Datenbank:
   [markus@localhost ~]$ curl -X PUT $HOST2/db-replica → {"ok":true}
4. Einfügen eines JSON Dokuments mit Inhalt "count":1 auf der Source Datenbank:
   [markus@localhost ~]$ curl -X PUT $HOST1/db/test -d '{"count":1},
   → {"ok":true,"id":„test","rev":"1-74620ecf527d29daaab9c2b465fbce66„
5. Replizieren des Dokuments von der Source auf die Replica Datenbank:
   [markus@localhost ~]$ curl -X POST $HOST1/_replicate -d
   '{"source":"db","target":"http://192.168.122.11:5984/db-replica"}‘
   → { "ok":true, ... }
6. Durchführung einer Änderung am Dokument auf der Source Datenbank, der Inhalt wird dabei auf "count":2 geändert:
   [markus@localhost ~]$ curl -X PUT $HOST1/db/test -d
   '{"count":2,"_rev":"1-74620ecf527d29daaab9c2b465fbce66"}'
   → {"ok":true,"id":„test","rev":"2-de0ea16f8621cbac506d23a0fbbde08a"}
7. Durchführung einer Änderung am Dokument auf der Replica Datenbank, der Inhalt wird dabei auf "count":3 geändert:
   [markus@localhost ~]$ curl -X PUT $HOST2/db-replica/test -d
   '{"count":3,"_rev":"1-74620ecf527d29daaab9c2b465fbce66"}'
   → {"ok":true,"id":„test","rev":"2-7c971bb974251ae8541b8fe045964219"}
8. Die Versionen der Dokumente unterscheiden sich nun und sind nicht mehr konsistent.
9. Nun wird der Inhalt von der Source Datenbank auf die Replica Datenbank repliziert:
   [markus@localhost ~]$ curl -X POST $HOST1/_replicate -d
   '{"source":"db","target":"http://192.168.122.11:5984/db-replica"},
   { "ok":true, ... }
10. Im nächsten Schritt erfolgt die Replizierung von der Replica Datenbank auf die Source Datenbank:
    [markus@localhost ~]$ curl -X POST $HOST1/_replicate -d

'{"source":"http://192.168.122.11:5984/db-replica","target":"db"},
→ { "ok":true, ... }

11. Nachdem Schritt 9 und 10 erfolgt ist, haben beide Datenbanken den gleichen Datenbestand und aufgrund der Replizierung in beide Richtungen wurde auch auf beiden Datenbanken der Konflikt erzeugt.
12. Mit Spezifizierung des Flags conflicts=true kann überprüft werden ob sich Dokumente in Konflikt befinden.
13. Wir überprüfen dies auf der Source Datenbank und stellen wie erwartet einen Konflikt fest:
    [markus@localhost ~]$ curl -X GET
    'http://192.168.122.10:5984/db/test?conflicts=true' {"_id":„test","_rev":"2-de0ea16f8621cbac506d23a0fbbde08a","count":2,
    "_conflicts":["2-7c971bb974251ae8541b8fe045964219"]}
14. Wir überprüfen nun auch die Replica Datenbank und stellen ebenfalls den erwarteten Konflikt fest:
    [markus@localhost ~]$ curl -X GET 'http://192.168.122.11:5984/db-replica/test?conflicts=true' {"_id":„test","_rev":"2-de0ea16f8621cbac506d23a0fbbde08a","count":2,
    "_conflicts":["2-7c971bb974251ae8541b8fe045964219"]}
15. Dabei haben beide Datenbanken aufgrund der automatisch angestoßenen Prozedur die gleiche Gewinnerrevision gewählt. Diese ist in diesem Beispiel jenes Dokument mit dem Inhalt "count":2. Das Dokument mit dem Inhalt "count":3 wird im Array "_conflicts" hinterlegt.
16. Es wird nun gezeigt wie die im Conflict Array hinterlegte Revision als die gültige ausgewählt wird. Dazu wird dieses Dokument erneut aus einen der beiden Knoten geschrieben:
    [markus@localhost ~]$ curl -X PUT $HOST1/db /test -d
    '{"_id":„test","_rev":"2-7c971bb974251ae8541b8fe045964219","count":3},
    → {"ok":true,"id":„test","rev":"3-5d0319b075a21b095719bc561def7122"}
17. Im Anschluss wird das Dokument mit Inhalt "count":2 gelöscht:
    [markus@localhost ~]$ curl -X DELETE $HOST1/db /test?rev=2-de0ea16f8621cbac506d23a0fbbde08a
    → {"ok":true,"id":„test","rev":"3-bfe83a296b0445c4d526ef35ef62ac14„
18. Bei einer erneuten Überprüfung sehen wir nun, dass der Konflikt gelöst wurde:
    [markus@localhost ~]$ curl -X GET
    'http://192.168.122.10:5984/db/test?conflicts=true'
    → {"_id":“test","_rev":"3-5d0319b075a21b095719bc561def7122","count":3}
19. Nun wird dieser Datenbestand noch auf die Replica Datenbank repliziert:
    [markus@localhost ~]$ curl -X POST $HOST1/_replicate -d '{"source":"db","target":"
    http://192.168.122.11:5984/db-replica "}'

20. Wir sehen nun auch auf der Replica Datenbank, dass der Konflikt gelöst wurde:
    [markus@localhost ~]$ curl -X GET 'http://localhost:5984/db/test?conflicts=true'
    {"_id":“test","_rev":"3-5d0319b075a21b095719bc561def7122","count":3}
21. Beide Datenbanken erreichen somit mit dem vom Client als gültig ausgewählten Dokument (mit dem Inhalt "count":3) einen konsistenten und konfliktfreien Zustand.