

## 0.1 ToDo

- MongoDB (neuartige Technologie), komplettes Kapitel
- Im Quelltextverzeichnis Problem mit dem Abstand zwischen Kapiteln, muss so wie bei dem Abbildungsverzeichnis sein...
- etc.

## 0.2 Nodes

- Ein verteiltes System ist in der Regel auch besser skalierbar als ein einzelner Computer, da man auf einfache Art und Weise durch Hinzufügen weiterer Rechner die Leistungsfähigkeit erhöhen kann.
- Ein häufig anzutreffendes Szenario ist natürlich auch die Bereitstellung von entfernten Ressourcen, wie es bei der Wikipedia der Fall ist. Außerdem werden verteilte Systeme zur Erhöhung der Ausfallsicherheit benutzt, indem bestimmte Funktionalitäten von mehreren Rechnern angeboten werden (Redundanz), so dass beim Ausfall eines Rechners die gleiche Funktionalität von einem weiteren Rechner angeboten wird.
- etc.

Weitere Gründe:

- Fernzugriff auf bestimmte Ressourcen (Drucker, ...)
  - Kooperation (Computer Supported Cooperative Work)
  - Lastverteilung
- In verteilten Speichersystemen werden Daten mehrfach über verschiedene Server repliziert, um die Verfügbarkeit der Daten zu erhöhen und die Zugriffzeiten zu verringern.

### 0.2.1 WICHTIG

#### ab - Apache HTTP server benchmarking tool

Siehe Details: <http://httpd.apache.org/docs/current/programs/ab.html>

Beispiel:

Listing 0.1: Monitoring mit ab

```
vlfa:scripts vlfa$ ab -n 100 -c 10 https://vlxxxfa.github.io
```

n ist die Anzahl von Anfragen

c ist die Anzahl der gleichzeitigen Bearbeitung von Anfragen

#### Benchmarking and Load Testing with Siege

Siehe Details: <http://blog.remarkablelabs.com/2012/11/benchmarking-and-load-testing-with-siege>

Listing 0.2: Monitoring mit siege

```
vlfa:scripts vlfa$ siege -b -c10 -t60S https://vlxxxfa.github.io
```

### 0.2.2 Monitoring

Was ist wichtig bei Monitoring <https://m.habrahabr.ru/company/oleg-bunin/blog/319526/>:

- Verfügbarkeit des Servers, läuft der noch..!
- Ressourcen zu wenig, Speicher, Prozessor etc.
- Fehler

Wie kann man das Ganze beobachten? Es gibt sehr gute verschiedene starke Tools zum Monitoring:

- Monit
- Zabbix
- Munin
- Nagios

Zum Fehler-Monitoring gibt es zwei gute Services:

- Rollbar
- Sentry

### 0.2.3 Web-Serviices

Amazon Elastic Compute Cloud (Amazon EC2) ist ein Web-Service, der anpassbare Rechenkapazität in der Cloud bietet. Der Service ist darauf ausgelegt, Cloud Computing für Entwickler zu erleichtern.

# Inhaltsverzeichnis

0.1	ToDo . . . . .	1
0.2	Nodes . . . . .	1
0.2.1	WICHTIG . . . . .	2
0.2.2	Monitoring . . . . .	2
0.2.3	Web-Serviices . . . . .	3
<b>1</b>	<b>Datenhaltungsschicht</b>	<b>1</b>
1.1	Allgemein . . . . .	1
1.1.1	ACID-Prinzip . . . . .	1
1.1.2	Skalierung . . . . .	2
1.2	NoSQL-Datenbanken . . . . .	4
1.2.1	Was ist NoSQL? . . . . .	4
1.2.2	Das CAP-Theorem . . . . .	5
1.2.3	BASE . . . . .	8
1.2.4	Arten von NoSQL-Datenbanken . . . . .	8
1.3	MongoDB . . . . .	12
1.3.1	Datensätze in Form von Dokumenten . . . . .	12
1.3.2	Die Nexus Architektur . . . . .	13
1.3.3	CRUD = IFUR . . . . .	14
1.3.4	Indizes . . . . .	15
1.3.5	Aggregation Framework . . . . .	19
1.3.6	Horizontale Skalierung (Sharding) . . . . .	19
1.3.7	Replikation (Replication) . . . . .	21
1.3.8	MongoDB mit Java . . . . .	26

<b>2 Prototyp</b>	<b>28</b>
2.1 Prototyp . . . . .	28
2.2 Fazit . . . . .	28
2.3 Apache Cassandra . . . . .	29

<b>Abbildungsverzeichnis</b>	<b>A</b>
------------------------------	----------

# Kapitel 1

## Datenhaltungsschicht

Jede Anwendung verarbeitet heutzutage sehr große Datenmengen. blabla

### 1.1 Allgemein

blablabla

#### 1.1.1 ACID-Prinzip

Fehlt noch warum ACID, wozu ACID im allgemeinen....., NUR FÜR RELATIONALE DATENBAN

ACID steht für Atomicity (Atomarität), Consistency (Konsistenz), Isolation (Isolation) und Durability (Dauerhaftigkeit) und beschreibt somit die Eigenschaften eines Datenbankmanagementsystems zur Sicherung der Datenkonsistenz bei Transaktionen.

- Atomicity (Atomarität): Die *Atomarität* einer Transaktion bedeutet, dass sie entweder ganz oder gar nicht ausgeführt wird. Falls eine Transaktion abgebrochen wird, werden alle im Laufe der Transaktion schon durchgeführte Änderungen rückgängig gemacht, was eigentlich zu einer sicheren Fehlerisolierung führt.

- **Consistency** (Konsistenz): Die *Konsistenz* besagt, dass vor und auch nach dem Ablauf einer Transaktion die Integrität und Plausibilität der Datenbestände gewährleistet werden. Die Integrität der Datenbank ist es möglich, beispielsweise mit Integritätsbedingungen<sup>1</sup> zu gewährleisten.
- **Isolation** (Isolation): Die *Isolation* dient zu Kapselung von Transaktionen, um unerwünschte Nebenwirkungen vermeiden zu können. Die Transaktionen müssen unabhängig voneinander ablaufen.
- **Durability** (Dauerhaftigkeit): Die *Dauerhaftigkeit* gewährleistet nach einer erfolgreichen Transaktion die Persistenz aller Datenänderungen. Im Falle eines Systemfehlers oder Neustarts müssen die Daten nichtsdestotrotz zur Verfügung stehen, dass sie in einer Datenbank dauerhaft gesichert sein müssen.

### 1.1.2 Skalierung

Der Begriff *Skalierung* beschreibt die Fähigkeit eines Systems, aufgrund der wachsenden Anforderungen, entweder die Leistung der vorhandenen Ressourcen zu verbessern oder zusätzlich die Neuen hinzufügen. In der Regel sind die verteilten Systeme besser skalierbar, da mit solchen System die Nebenläufigkeit von Prozessen realisiert wird und somit die Leistungsfähigkeit der verwendeten Ressourcen steigt. Bei der Skalierung sind zwei Arten zu unterscheiden, eine vertikale und horizontale Skalierung, die es zunächst näher zu erläutern gilt.

**Ergänzung:** GRAPH für vertikale und horizontale zeichnen und hinzufügen, schaue Beispiel im img-Ordner.....!

---

<sup>1</sup>Unter Integritätsbedingungen (Zusicherungen, Assertions) sind Bedingungen zu verstehen, die die Korrektheit der gespeicherten Daten sichern. Diese werden in SQL zum Beispiel mithilfe von CONSTRAINTS formuliert. Folgende CONSTRAINTS sind möglich: NULL, NOT NULL, PRIMARY KEY, FOREIGN KEY etc.

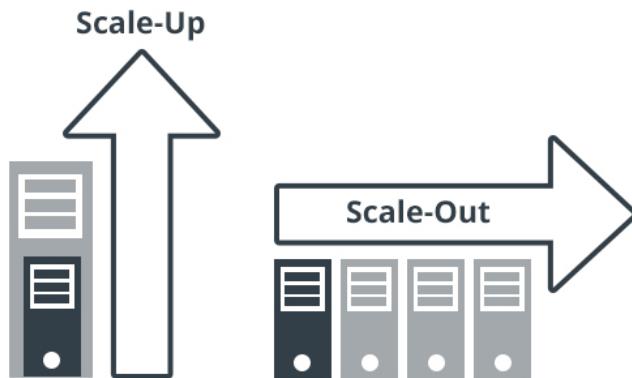


Abbildung 1.1: Skalierung<sup>2</sup>

### Vertikale Skalierung

Die vertikale Skalierung (*scale-up*) strebt die qualitative Steigerung der Leistungsfähigkeit an, bei der die schon eingesetzten Ressourcen beispielsweise durch die Speichererweiterung oder CPU-Steigerung einfach verbessert werden.

### Horizontale Skalierung

Die horizontale Skalierung (*scale-out*), im Gegensatz zur vertikalen Skalierung verteilt die Daten auf verschiedenen Knoten im großen Cluster, wobei die quantitative Steigerung der Leistungsfähigkeit angestrebt wird. Somit können mehrere weniger leistungsfähigere, nicht so teure Rechner eingesetzt werden. Dadurch ist es möglich, sehr große horizontale Skalierbarkeit zu gewährleisten, da die Knoten somit nicht so stark, im Vergleich zur vertikalen Skalierung überlastet sind.

<sup>2</sup>Skalierung: <https://magazin.kapilendo.de/den-supergau-verhindern-so-bereiten-sie-ihre-website-auf-einen-besucheransturm-vor/>, zugegriffen am 15. Januar 2017

## 1.2 NoSQL-Datenbanken

Im Vergleich zu den relationalen Datenbanken, die sich als eine strukturierte Sammlung von Tabellen (den Relationen) vorstellen, in welchen Datensätze abgespeichert sind, eignen sich NoSQL-Datenbanken zur unstrukturierter Daten, die einen nicht-relationalen Ansatz verfolgen.

### 1.2.1 Was ist NoSQL?

Der Begriff NoSQL steht nicht für 'kein SQL', sondern für 'nicht nur SQL' (Not only SQL). Das Ziel von NoSQL ist, relationale Datenbanken sinnvoll zu ergänzen, wo sie Defizite aufzeigen. Entstanden ist dieses Konzept in erster Linie als Antwort zur Unflexibilität, sowie zur relativ schwierigen Skalierbarkeit von klassischen Datenbanksystemen, bei denen die Daten nach einem stark strukturierten Modell gespeichert werden müssen.<sup>3</sup> Dokumentdatenbanken gruppieren die Daten in einem strukturierten Dokument, typischerweise in einer JSON-Datenstruktur. Auch **MongoDB**, siehe dazu Abschnitt 1.3 verfolgt diesen Ansatz und bietet darauf aufbauend eine reichhaltige Abfragesprache und Indexe auf einzelne Datenfelder. Die Möglichkeiten der Replikation und des Shardings zur stufenlosen und unkomplizierten Skalierung der Daten und Zugriffe macht **MongoDB** auch für stark frequentierte Websites äußerst interessant.([**Hollosi.2012**], Kapitel 14, Seite 435)

Beispiele für NoSQL-Datenbanken....:

- CouchDB
- MongoDB
- Redis
- Google BigTable
- Amazon Dynamo
- Apache Cassandra
- Hbase (ApacheHadoop)
- Twitter Gizzard
- weitere...

---

<sup>3</sup>MySQL vs. MongoDB: <http://www.computerwoche.de/a/datenbanksysteme-fuer-web-anwendungen-im-vergleich,2496589>, zugegriffen am 3. Januar 2016

Jede NoSQL-Datenbanke verfolgt seine Ziele und welche sie genau verfolgen, beschreibt der Abschnitt 1.2.4, in dem verschiedene Kategorien von NoSQL-Datenbanken vorgestellt werden.

### 1.2.2 Das CAP-Theorem

Im Jahr 2000 hielt Brewer<sup>4</sup> die Keynote auf dem ACM Symposium on Principles of Distributed Computing (PODC)<sup>5</sup>, einer Konferenz über die Grundlagen der Datenverarbeitung in verteilten Systemen<sup>6</sup> (Principles of Distributed Computing). In seiner Keynote stellte Brewer sein **CAP**-Theorem vor, ein Ergebnis seiner Forschungen zu verteilten Systemen an der University of California [Kurowski.2012]. Brewer's Theorem wurde im Jahr 2002 von Seth Gilbert und Nancy Lynch formal bewiesen.

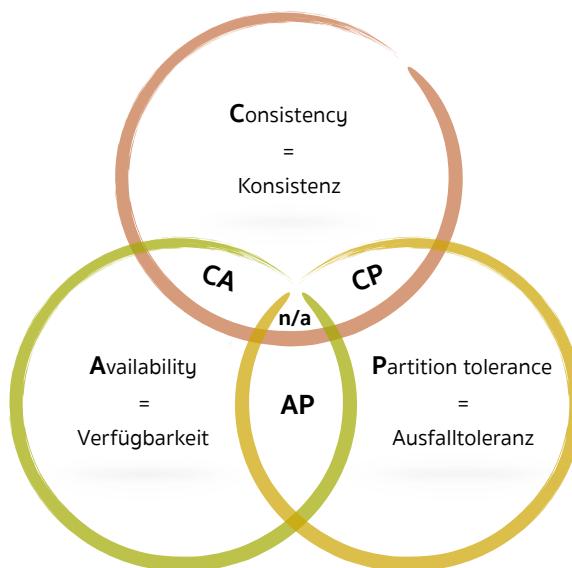


Abbildung 1.2: Anforderungen an verteilte Systeme gemäß dem **CAP**-Theorem

Das Akronym **CAP** steht für die englischsprachigen Begriffe **Consistency** (Konsistenz),

<sup>4</sup>Eric A. Brewer ist ein Informatik-Professor an der University of California, Berkeley und einer der Erfinder der Suchmaschine Inktomi

<sup>5</sup>PODC2000: <http://www.podc.org/podc2000/>, zugegriffen am 02.01.2017

<sup>6</sup>In einem verteilten System im Bereich Datenverarbeitung werden gespeicherte Daten mehrfach über mindestens zwei verschiedene Server repliziert und miteinander synchronisiert, um die Verfügbarkeit der Daten zu erhöhen und die Zugriffszeiten der User zu verringern.

**Availability** (Hochverfügbarkeit) und **Partition Tolerance** (Partitionstoleranz) und beschreiben die Anforderungen für die Skalierung an verteilte Systeme, die es zunächst näher zu erläutern gilt.

Was besagt eigentlich dieses **CAP**-Theorem? Das **CAP**-Theorem besagt, dass die verteilten Systeme, die mit großen Datenmengen zu tun haben, gleichzeitig die folgenden Anforderungen wie **Consistency** (Konsistenz), **Availability** (Hochverfügbarkeit) und **Partition Tolerance** (Partitionstoleranz) nicht erfüllen können.

- **Consistency** (Konsistenz): Die *Konsistenz* der Daten in einem verteilten System wird im Prinzip genauso geschätzt wie im Abschnitt 1.1.1 die schon besprochene **Atomicity** (Atomarität)-Eigenschaft. Bedeutet, dass alle replizierenden Knoten aus einem großen Cluster über die gleichen Daten verfügen. Falls ein Wert auf einem Knoten durch eine Transaktion per Schreiboperation geändert wird, muss der aktualisierte Wert auf Anfrage mit der Leseoperation von anderen Knoten zurückgeliefert werden können. Die Transaktion selbst ist eine atomare<sup>7</sup> Einheit in der Datenbank.
- **Availability** (Hochverfügbarkeit): Die *Hochverfügbarkeit* ist die weitere Anforderung, die besagt, dass immer alle gesendeten Anfragen durch User ans System beantwortet werden müssen und mit einer akzeptablen Reaktionszeit.
- **Partition Tolerance** (Partitionstoleranz): Die *Partitions- oder Ausfalltoleranz* bedeutet, dass der Ausfall eines Knoten bzw. eines Servers aus einem Cluster das verteilte System nicht beeinträchtigt und es fehlerfrei weiter funktioniert. Falls einzelne Knoten in so einem System ausfallen, wird deren Ausfall von den verbleibenden Knoten aus dem Cluster kompensiert, um die Funktionsfähigkeit des Gesamtsystems aufrecht zu halten.

Die graphische Darstellung für das Brewer's **CAP**-Theorem ist aus der Abbildung 1.2 zu entnehmen. Wie die Abbildung 1.2 erkennen lässt, können in einem verteilten System gleichzeitig und vollständig nur zwei dieser drei Anforderungen **Consistency** (Konsistenz), **Availability** (Hochverfügbarkeit), **Partition Tolerance** (Partitionstoleranz) erfüllt sein. Konkret aus der Praxis bedeutet das, dass es für eine hohe Verfügbarkeit und

---

<sup>7</sup>Eine atomare Transaktion bedeutet, dass sie entweder ganz oder gar nicht ausgeführt wird. Falls eine atomare Transaktion abgebrochen wird, werden alle im Laufe der Transaktion schon durchgeführte Änderungen rückgängig gemacht.

Partitions- oder Ausfalltoleranz notwendig ist, die Anforderungen an die Konsistenz zu lockern [Edlich.2011].

Die Anforderungen in Paaren klassifizieren gemäß dem **CAP**-Theorem bestimmte Datenbanktechnologien. Für jede Anwendung muss daher individuell entschieden werden, ob sie als ein **CA**-, **CP**- oder **AP**-System zu realisieren ist.

- **CA (Consistency und Availability):** Die klassischen relationalen Datenbankmanagementsysteme (RDBMS) wie Oracle, DB2 etc. fallen in **CA**-Kategorie, die vor allem **Consistency** (Konsistenz) und **Availability** (Hochverfügbarkeit) aller Knoten in einem Cluster hinzielt. Hierbei werden die Daten nach dem **ACID**-Prinzip verwaltet. Die relationalen Datenbanken sind für Ein-Server-Hardware konzipiert und vertikal skalierbar. Das bedeutet, dass solche Systeme mit hochverfügbaren Servern betrieben werden und **Partition Tolerance** (Partitionstoleranz) nicht unbedingt in Frage kommt.
  - keine Partitionstoleranz
  - (Relationale) Datenbank ermöglicht verteilte Transaktionen zur Konsistenzwahrung
  - Voraussetzung: funktionierendes Netzwerk (kein Nachrichtenverlust)
  - URL: [http://dbs.uni-leipzig.de/file/NoSQL\\_SS14\\_01\\_Intro.pdf](http://dbs.uni-leipzig.de/file/NoSQL_SS14_01_Intro.pdf)
- **CP (Consistency und Partition tolerance):** Ein gutes Beispiel für die Anwendungen, die zu der **CP**-Kategorie zu ordnen sind, sind Banking-Anwendungen. Für solche Anwendungen ist es wichtig, dass die Transaktionen zuverlässig durchgeführt werden und der mögliche Ausfall eines Knotens sichergestellt wird.
  - keine Verfügbarkeit
  - im Falle von Netzwerkpartitionierung werden Transaktionen blockiert
  - Vermeidung möglicher Konflikte bei Merge, dadurch Sicherstellung der Konsistenz
  - URL: [http://dbs.uni-leipzig.de/file/NoSQL\\_SS14\\_01\\_Intro.pdf](http://dbs.uni-leipzig.de/file/NoSQL_SS14_01_Intro.pdf)

- **AP** (**A**vailability und **P**artition tolerance): Für die Anwendungen, die in die **AP**-Kategorie fallen, rückt die Anforderung **C**onsistency (Konsistenz) in den Hintergrund. Beispiele für solche Anwendungen sind die Social-Media-Sites wie Twitter<sup>8</sup> oder Facebook<sup>9</sup>, da die Hauptidee der Anwendung dadurch nicht verfällt, wenn zum gleichen Zeitpunkt die replizierten Knoten nicht über die gleiche Datenstruktur verfügen.
- keine Konsistenz
- Writes stets möglich auch wenn keine Kommunikation mit anderen Knoten möglich (z.B. Synchronisation)
- Notwendigkeit der Auflösung inkonsistenter Daten, d.h. verschiedene Versionen des selben Datums an verschiedenen Knoten
- URL: [http://dbs.uni-leipzig.de/file/NoSQL\\_SS14\\_01\\_Intro.pdf](http://dbs.uni-leipzig.de/file/NoSQL_SS14_01_Intro.pdf)

### 1.2.3 **BASE**

**BASE** steht für **B**asically **A**vailable, **S**oft State, **E**ventually **C**onsistent und beschreibt den Gegenteil zu den strengen **ACID**-Kriterien aus dem Teilabschnitt 1.1.1. **BASE** ist wie **CAP**-Theorem (Teilabschnitt 1.2.2) auch für verteilte Datenbanksysteme formuliert, für die die *Konsistenz* nicht mehr im Vordergrund steht, sondern die *Verfügbarkeit* eines Systems. Bei solchen Systemen, die nach dem **BASE**-Prinzip gestaltet sind, ist eher wichtig, dass für alle Clients das System ständig verfügbar ist. Die Clients müssen nicht unbedingt zu dem gleichen Zeitpunkt die gleichen Daten sehen.

### 1.2.4 **Arten von NoSQL-Datenbanken**

Eigene Graphen als Abbildungen verwenden, diese sind nur als Beispiel drin.....! Der folgende Abschnitt stellt insgesamt vier Kategorien von NoSQL-Datenbanken dar, wie in der Abbildung 1.3 zu sehen ist.

---

<sup>8</sup>Twitter: <https://twitter.com/>

<sup>9</sup>Facebook: <https://www.facebook.com/>

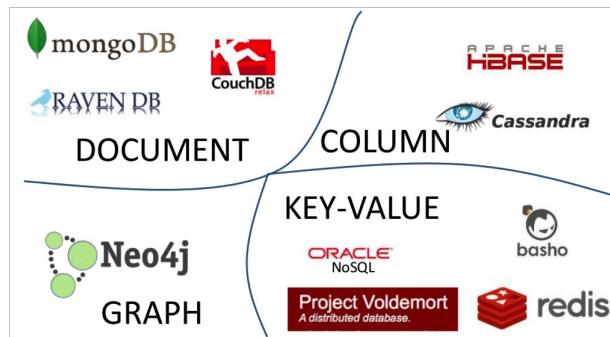


Abbildung 1.3: NoSQL-Datenbanken, verteilt in vier Gruppen<sup>10</sup>

Die **MongoDB** wird im Abschnitt 1.3 detailliert beschrieben. BlaBlaBla

### Key-Value-Datenbanken

Eine Key-Value-Datenbank (*Key-Value Store*) ist eine Datenbank, in der die Daten in Form von Schlüssel-Werte-Paaren abgespeichert werden. Der Schlüssel verweist dabei auf einen eindeutigen (meist in Binär- oder Zeichenketten-Format vorliegenden) Wert<sup>11</sup>. Value kann oft beliebiger Datentyp wie Arrays, Dokumente, Objekte, Bytes etc. sein. Siehe dazu die Abbildung 1.4 mit einem Beispiel:

Example : Key-Value Store	
Key	Value
Mahesh	{"Mathematics, Science, History, Geography"}
Uma	{"English, Hindi, French, German"}
Paul	{"Computers, Programming"}
Abraham	{"Geology, Metallurgy, Material Science"}

Abbildung 1.4: Key-Value-Datenbank als Beispiel<sup>12</sup>

<sup>10</sup>NoSQL-Datenbanken: <http://bigdata-blog.com/key-value-database>, zugegriffen am 17. Januar 2017

<sup>11</sup>NoSQL: Key-Value-Datenbank Redis im Überblick: <https://www.heise.de/developer/artikel/NoSQL-Key-Value-Datenbank-Redis-im-Ueberblick-1233843.html>, zugegriffen am 17. Januar 2017

<sup>12</sup>NoSQL: A Silver Bullet for handling Big Data?: <https://www.linkedin.com/pulse/nosql-silver-bullet-handling-big-data-shashank-dhaneshwar>, zugegriffen am 17. Januar 2017

## Spaltenorientierte Datenbanken

In einer spaltenorientierten Datenbank (*Column Store*), wie der Name vermuten lässt, werden die Datensätze spalten- statt zeilenweise abgespeichert. Durch die spaltenorientierte Abspeicherung der Daten wird der Lesezugriff stark beschleunigt, da keine unnötigen Informationen mehr gelesen werden, stattdessen nur diejenigen, die wirklich benötigt wurden. Dadurch wird der Schreibprozess aber erschwert, falls die schreibenden Daten aus mehreren Spalten bestehen werden, auf die entsprechend zugegriffen werden muss. Der Schreibprozess wird sich in diesem Fall etwas verlangsamen.

Abbildung==?????

## Graphen-Datenbanken

Eine Graphen-Datenbank (*Graph database*) ist weitere Kategorie aus der NoSQL Gruppe, in der die Daten anhand eines Graphen dargestellt und abgespeichert werden.

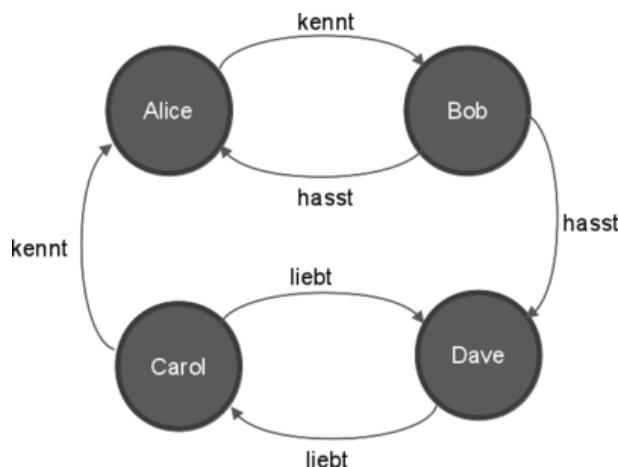


Abbildung 1.5: Beispiel für die Darstellung der Daten in einer Graphen-Datenbank<sup>13</sup>

Wie der Abbildung 1.5 zu entnehmen ist, bestehen Graphen grundsätzlich aus Knoten (*Node*) und Kanten (*Edge*). Dabei stellen die Kanten die Verbindungen zwischen den einzelnen Knoten dar.

<sup>13</sup>Beispiel für die Darstellung der Daten in einer Graphen-Datenbank: <https://de.wikipedia.org/wiki/Graphdatenbank>, zugegriffen am 17. Januar 2017

## Dokumentenorientierte Datenbanken

Eine Datenbank, in der die Daten in Form von Dokumenten abgespeichert werden, ist als eine dokumentenorientierte Datenbank (*Document Store*) zu definieren. In diesem Zusammenhang ist ein Dokument als eine Zusammenstellung bestimmter Daten zu verstehen, das mit einem eindeutigen Identifikator angesprochen werden kann. Da die Daten in der dokumentenorientierten Datenbank nicht in Form von Tabellen, sondern in Form von Dokumenten abgespeichert werden, ergibt sich daraus keinen Strukturzwang. Als Beispiel ist aus der Abbildung 1.6 zwei Dokumente im **JSON**-Format zu entnehmen, die sich voneinander gänzlich unterscheiden.

```
{  
    "city" : "FISHERS ISLAND",  
    "loc" : [  
        -72.017834,  
        41.263934  
    ],  
    "pop" : 329,  
    "state" : "NY",  
    "_id" : "06390"  
}  
  
{  
    "_id" : ObjectId("50b1aa983b3d0043b51b2c52"),  
    "name" : "Nexus 7",  
    "category" : "Tablets",  
    "manufacturer" : "Google",  
    "price" : 199  
}
```

Abbildung 1.6: Zwei Dokumente im JSON Format

Möchte man ein bestimmtes Dokument erweitern, so kann man es einfach tun, da eine dokumentenorientierte Datenbank strukturfrei ist. Weitere Datenformate sind beispielsweise YAML<sup>14</sup> (angelehnt an XML) oder XML<sup>15</sup> selbst.

<sup>14</sup>YAML: <http://www.yaml.org/start.html>

<sup>15</sup>XML: <https://www.xml.com/>

## 1.3 MongoDB

**MongoDB** ist eine schemalose, dokumentenorientierte Open-Source-Datenbank und gehört somit zu einer der im Teilabschnitt 1.2.4 besprochenen Arten von NoSQL-Datenbanken. Der Name stammt von dem englischen Begriff *huMONGous*, ins Deutsche als *gigantisch* oder *riesig* übersetzen lässt. Die genannte NoSQL-Datenbank macht mit seinem effizienten dokumentenorientierten Ansatz, einfacher Skalierbarkeit und hoher Flexibilität dem bewährten MySQL<sup>16</sup>-System zunehmend Konkurrenz.<sup>17</sup>

Die Unterschiede zu den Konzepten der relationalen und nicht-relationalen Datenbanken konkret von **MongoDB** stellt die Tabelle 1.1 dar.

Relational	MongoDB
Database	Database
Table	Collection
Row	Document
Index	Index
Join	Lookup
Foreign Key	Reference
Multi-table transaction	Single document transaction

Tabelle 1.1: Konzepte

### 1.3.1 Datensätze in Form von Dokumenten

Die NoSQL-Datenbank **MongoDB** verwendet für die Dokumentenspeicherung und den Datenaustausch das sogenannte **BSON**<sup>18</sup>-Format, das eine binäre Darstellung von **JSON**-ähnlichen Dokumenten bietet. Die Dokumente selbst werden von **MongoDB** in sogenannten Kollektionen (*Collections*) gespeichert. Wie es schon aus dem Abschnitt 1.2.4 bekannt

<sup>16</sup>MySQL: <https://www.mysql.com>

<sup>17</sup>MySQL vs. MongoDB: <http://www.computerwoche.de/a/datenbanksysteme-fuer-web-anwendungen-im-vergleich,2496589>, zugegriffen am 19. Januar 2017

<sup>18</sup>BJSON: <http://www.bjson.org>

ist, kann jedes Dokument eine beliebige Anzahl an Feldern besitzen, unabhängig voneinander.

Zudem dürfen Dokumente auch innerhalb eines Dokuments gespeichert werden<sup>19</sup>. Die Speicherung von Daten in Form von Dokumenten bietet den Vorteil, dass sowohl strukturierte, als auch semi-strukturierte und polymorphe Daten gespeichert werden können. Dokumente, die jedoch das gleiche oder ein ähnliches Format haben, sollten zu einer Kollektion (*Collection*) zusammengefasst werden<sup>20</sup>.

### 1.3.2 Die Nexus Architektur

Die MongoDB konzentriert sich auf die Kombination blabla

Zu den wichtigsten Eigenschaften, die für einen Einsatz von **MongoDB** sprechen, gehören:

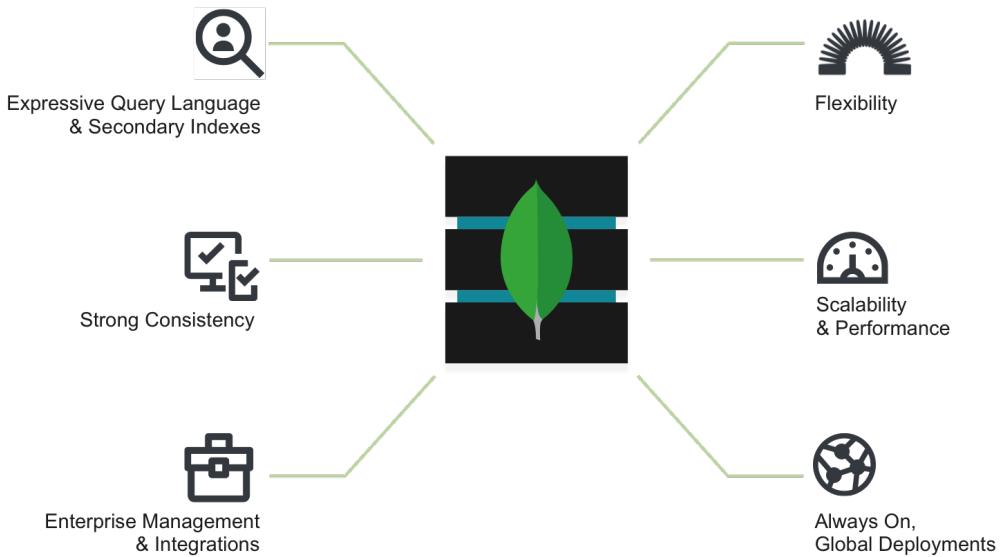
- Hochverfügbarkeit: Auch bei Ausfall einer Datenbankinstanz soll die Applikation weiterhin verfügbar bleiben, d. h. nahtlos – ohne manuellen Eingriff – müssen redundante Instanzen bei einem Ausfall einspringen
- Skalierbarkeit: Mit transparentem Sharding, siehe Teilabschnitt ?? – einem Verfahren zur horizontalen Skalierung – kann die Infrastruktur vergleichsweise einfach wachsenden Anforderungen angepasst werden
- Performanz: Vom Ansatz her haben dokumentenorientierte DBMS hier einen Vorteil, weil die Daten nicht erst aus mehreren Tabellen zusammengeführt werden<sup>21</sup>

Die Vorteile relationaler und NoSQL-Datenbanken sind aus der Abbildung 1.7 zu entnehmen.

<sup>19</sup>Einführung in MongoDB: <https://www.iks-gmbh.com/assets/downloads/Einfuehrung-in-MongoDB-iks.pdf>, zugegriffen am 19. Januar 2017

<sup>20</sup>MongoDB: <http://www.moretechnology.de/mongodb-eine-dokumentenorientierte-datenbank/>, zugegriffen am 21. Januar 2017

<sup>21</sup>MongoDB Eigenschaften: <https://entwickler.de/online/datenbanken/mongodb-erfolgreich-ein-dokumentenorientiertes-datenbanksystem-einfuehren-115079.html>, zugegriffen am 12. Dezember 2016

Abbildung 1.7: MongoDB Architektur<sup>22</sup>

### 1.3.3 CRUD = IFUR

**CRUD**-Operationen heißen in MongoDB ***I*nser**, ***F*ind**, ***U*pdate** und ***R*emove**.

Bei **MongoDB** ist es nicht einmal notwendig, eine Datenbank oder eine Collection zu definieren, bevor etwas gespeichert werden wird. Datenbanken und Collections werden zur Laufzeit beim ersten Einfügen eines Dokuments von MongoDB erzeugt.

#### Create/Insert

blablablabla

##### Listing 1.1: Dokument(e) speichern

```
> db.collection.insert(..)
> db.collection.insertMany(..)
> db.collection.insertOne(..)
```

<sup>22</sup>MongoDB Architektur: <https://www.mongodb.com/mongodb-architecture>, zugegriffen am 28. Januar 2016

## Read/Find

blablablala

### Listing 1.2: Dokument(e) finden

```
> db.collection.find(..)
> db.collection.findOne(..)
> db.collection.findOneAndDelete(..) etc.
```

## Update/Update

blablablala

### Listing 1.3: Dokument(e) aktualisieren

```
> db.collection.update(..)
> db.collection.updateMany(..)
> db.collection.updateOne(..)
```

## Delete/Remove

blablablala

### Listing 1.4: Dokument(e) löschen

```
> db.collection.remove(..)
```

## 1.3.4 Indizes

Indizes in MongoDB werden als Binär-Baum<sup>23</sup> in einer vordefinierten Sortierreihenfolge abgelegt, siehe Abbildung 1.8

<sup>23</sup>Binär-Baum: [http://www.hs-augsburg.de/mebib/emiell/entw\\_inf/lernprogramme/baeume/gdi\\_kap\\_4\\_1.html](http://www.hs-augsburg.de/mebib/emiell/entw_inf/lernprogramme/baeume/gdi_kap_4_1.html), zugegriffen am 23. Dezember 2016

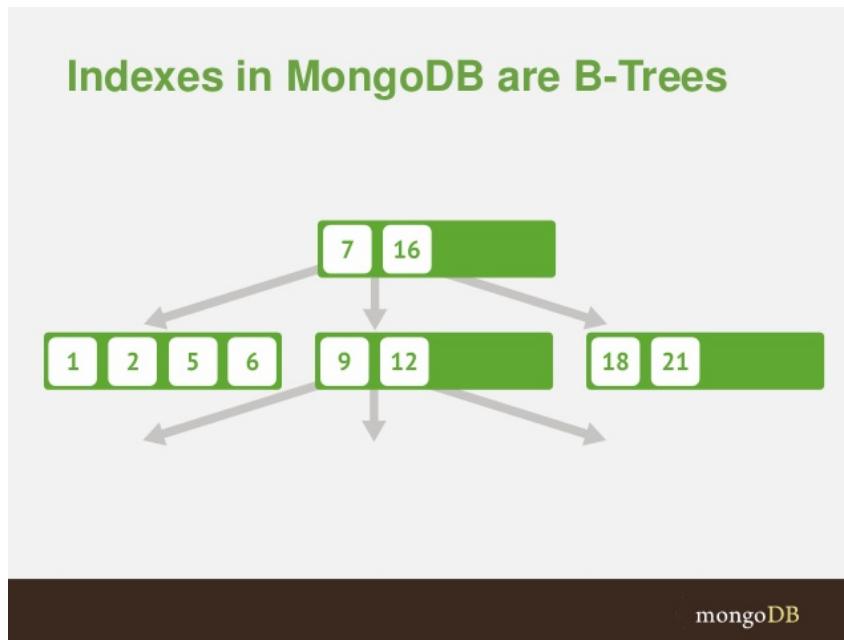


Abbildung 1.8: Indizes in **MongoDB** als Binär-Baum<sup>24</sup>

Der Index wird beim Erstellen, Updaten und Löschen eines Dokumentes auch aktualisiert. Zu viele Indizes machen schreib Operationen langsam und die Größe der Indizes steigt an, deswegen sollte ein Index nur auf Felder zeigen, auf die auch Query-Operationen angewendet werden. Beim Anlegen der Indizes ist die Sortierreihenfolge zu gunsten einer schnelleren Suche wichtig, da sie schon vorsortiert vorliegen und nicht erst bei der Liveabfrage sortiert werden müssen.<sup>25</sup>

Es gibt drei Möglichkeiten der Sortierung:

- Aufsteigend(1),
- Absteigend (-1),
- geospatial (2d).

Index hilft, Datenbanken zu optimieren. Die nachfolgende ?? demonstriert

<sup>24</sup>Indizes in MongoDB als Binär-Baum: <http://www.slideshare.net/mongodb/indexing-strategies-to-help-you-scale>, zugegriffen am 23. Dezember 2016

<sup>25</sup>Indizes: [http://wikis.gm.fh-koeln.de/wiki\\_db/MongoDB/Indizes](http://wikis.gm.fh-koeln.de/wiki_db/MongoDB/Indizes), aufgerufen am 24. Dezember 2016

Neben dem obligatorischen Primär-Index auf dem Feld `_id`, das in jedem Dokument existieren und pro Collection eindeutig sein muss, können Sie in MongoDB bis zu 63 weitere Sekundär-Indizes pro Collection anlegen, um Suchanfragen zu beschleunigen. Ein Sekundär-Index kann auf einem einzelnen Feld oder einer Gruppe von Feldern angelegt werden.

Which optimization will typically have the greatest impact on the performance of a database?

Adding appropriate indexes on large collections so that only a small percentage of queries need to scan the collection.

## Creating Indexes

blabla

### Listing 1.5: Mongo-Shell: Something else

```
> db.students.createIndex();
```

blabla

### Listing 1.6: Mongo-Shell: Something else

```
> db.students.explain().find();
```

Quiz: Please provide the mongo shell command to add an index to a collection named `students`, having the index key be `class, student_name`. Neither will go in the 1"direction..

### Listing 1.7: Something else

```
> db.students.createIndex({student\_name:1, class:1});
```

### Listing 1.8: Mongo-Shell: Something else

```
> db.students.dropIndex({student_name:1});
```

## Multikey Indexes

blabla

### Index Creation Option, Unique

für jedes attribut kann man Unique definieren, d.h. doppelte Werte dürfen nicht vorkommen

#### Listing 1.9: Mongo-Shell: Something else

```
> db.students.createIndex({student_id : test}, {unique:true});
```

Please provide the mongo shell command to create a unique index on student\_id, class\_id, ascending for the collection students.

#### Listing 1.10: Mongo-Shell: Something else

```
> db.students.createIndex({student_id:1, class_id:1}, {unique:true});
```

### Index Creation, Sparse

Im Fall, wenn ein Attribut nicht in allen Dokumenten vorkommt, aber für dieses ein Unique Index definiert werden soll, muss Folgendes verwendet werden:

#### Listing 1.11: Something else

```
> db.students.createIndex({cell:1}, {unique:true, sparse:true});
```

blabla, siehe den Shellbefehl, blabla

#### Listing 1.12: Something else

```
> db.students.createIndex({student_id:1, class_id:1}, {unique:true});
```

siehe Codeauszug

### 1.3.5 Aggregation Framework

Seit der Version 3.2 der dokumentenorientierten NoSQL-Datenbank **MongoDB**, die am 8. Dezember 2015 erschienen ist, ist *Aggregation Framework* mit zwei grundlegenden Features ergänzt worden, wie *Fetch Joins* und *Schema-Validierung*. Bislang waren die (*Collections*) völlig isoliert voneinander, was die **CRUD**-Operationen (Teilabschnitt 1.3.3) erschwert hat. Mit *Joins* ist es möglich geworden, Dokumente aus anderen (*Collections*) mitzuladen.

Über Schema-Validierung noch schreiben...

Ein Beispiel für Joins dazu hinzufügen, in dem ein eingebettetes Dokument gezeigt wird...!

How good is it? Mapping between SQL and Aggregation Um die Nutzung der Aggregation Framework in MongoDB zu ermöglichen, stellt MongoDB Java Driver zur Verfügung.

### 1.3.6 Horizontale Skalierung (Sharding)

Um eine kostengünstige Lösung für eine Steigerung der Leistung von Systemen zu ermöglichen, ermöglicht das Datenbanksystem von **MongoDB** eine horizontale Skalierung, die allgemein im Teilabschnitt 1.1.2 schon diskutiert ist. Die horizontale Verteilung der Daten erfolgt bei **MongoDB** auf Ebene der *Collections*, siehe Abbildung 1.9, mithilfe von sogenannten *Sharding-Keys*. Die *Sharding-Keys* dienen dazu, um später Zugriffe auf einzelne Dokumente zu ermöglichen.

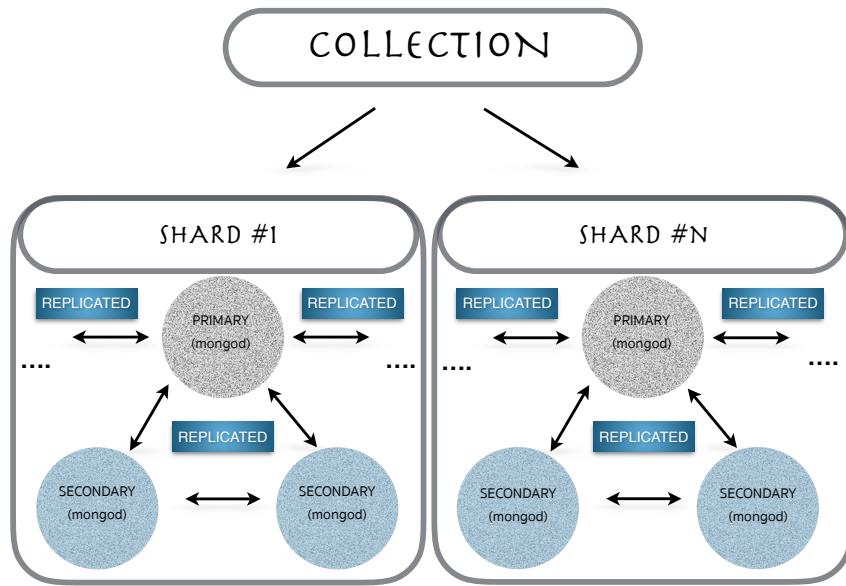


Abbildung 1.9: Ein Beispiel für Verteilung einer *Collection* auf mehreren *Shards*

Die Aufteilung der *Collections* erfolgt in Blöcken, auch als *Chunks* genannt. Ein *Chunk* ist ein Teil einer bestimmten *Collection*. Gespeichert werden *Chunks* auf Servern, die in diesem Zusammenhang als *Shards* bezeichnet werden. Was es unter *Shards* zu verstehen ist, erläutert der Teilabschnitt 1.3.7 zur Replikation.

Um die Aufteilung der *Collections* in *Chunks* auf *Shards* realisieren zu können, verwendet **MongoDB** folgende Komponenten:

- *shards*: Die *Shards* enthalten letztendlich die Daten. In einer *Shard* ist es möglich, Replikationsgruppen zu verwenden, näher dazu im Teilabschnitt 1.3.7.
- *mongos*: *mongos* gilt als ein *RoutingService*, der die Abfragen der Anwendungsschicht verarbeitet und diese an eine entsprechende *Shard* weiterleitet, die die nötigen Daten enthält.
- *config servers*: Die *Config Servers* speichern die Metadaten für einen Sharded-Cluster.

Die folgende Abbildung 1.10 veranschaulicht die Interaktion von den genannten Komponenten innerhalb eines Sharded-Cluster:

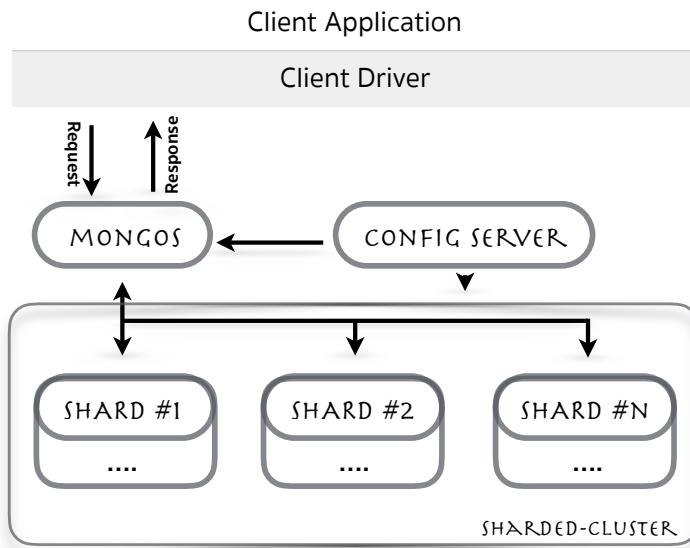


Abbildung 1.10: Horizontale Skalierung (*Sharding*)

Das Ziel des Ganzen ist die horizontale Skalierbarkeit an Datenmengen, um die Performance des Datenbanksystems zu steigern.

### 1.3.7 Replikation (Replication)

Manchmal kann es dazu kommen, dass ein Server ausfällt und die Schreib- und Lesezugriffe dadurch auf eine kurze Zeit nicht mehr möglich sind. Um Schreib- und Lesezugriffe auch im Fall eines Serverausfalls ständig ermöglichen zu können, hat **MongoDB** einen Replikationsmechanismus entwickelt. Der Replikationsmechanismus dient zur Replikation bzw. zum Spiegeln der Daten auf mehreren Servern und funktioniert nach einem *Master-n-Slaves-Prinzip*.

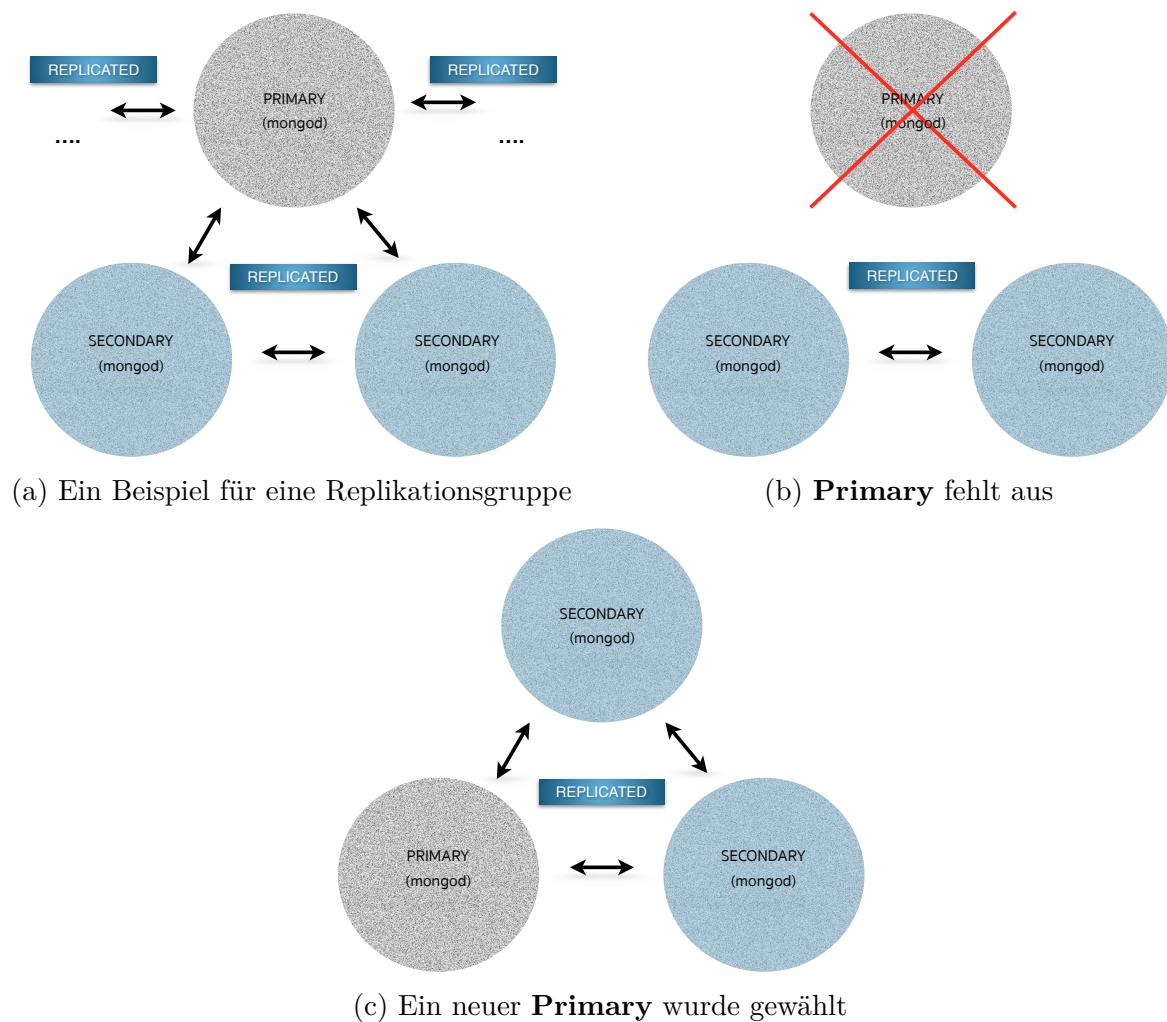


Abbildung 1.11: Szenario für eine Replikationsgruppe mit drei Servern in einer *Shard*

Ein *Master*, auch ein *Primary* genannt, besitzt Schreib- und Leserechte. Dieser repliziert die Daten auf *n-Slaves*, die auch als *Secondaries* bezeichnet werden. Ein *Primary* mit *n-Secondaries* bilden gemeinsam eine *Shard*. Eine *Shard* kann aus mind. einem Server bestehen. Falls eine *Shard* aus mehreren Servern besteht, so kann **MongoDB** die Server in Replikationsgruppe (*Replica set*) anordnen, damit bei Ausfall eines Servers die Verfügbarkeit der Datenbank trotzdem gewährleistet ist. Mit Replikationsgruppen will **MongoDB** die Ausfallsicherheit sicherstellen. Die Abbildung 1.11 veranschaulicht ein Szenario für eine Replikationsgruppe mit drei Knoten. Jeder Knoten aus der Gruppe ist als einen eigenen Server vorzustellen. Das *Master-n-Slaves-Prinzip* besagt, dass in einer Replikationsgruppe nur ein Master und n-Slaves existieren können, um eine strenge Konsistenz gewährleisten

zu können.

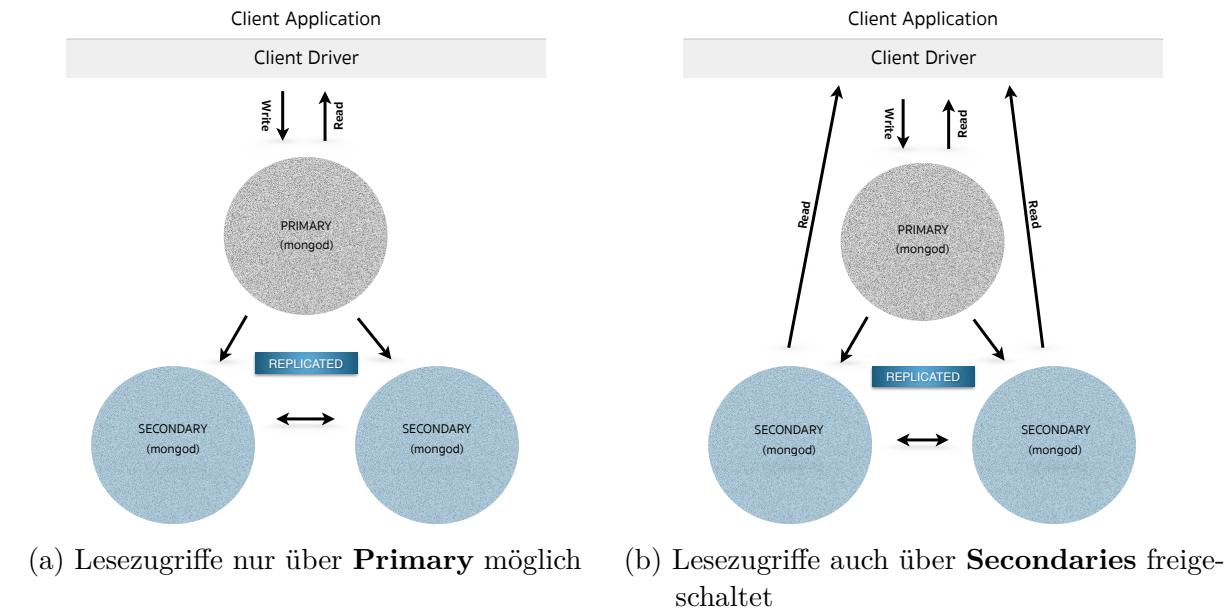


Abbildung 1.12: Freischaltung der Lesezugriffe

Im Gegenteil zu dem Primary sind bei Secondaries die Schreib- und Leserechte von Anfang an nicht möglich. Falls der Kontext der Anwendung verlangt, können nur die Leserechte entsprechend freigeschaltet werden.

### Eine Replikationsgruppe erzeugen

In diesem Teilabschnitt wird eine Replikationsgruppe mit insgesamt drei Servern erzeugt. Jeder Schritt der Konfiguration wird in diesem Teilabschnitt nachgespielt.

Um eine Replikationsgruppe effizienter erzeugen zu können, wird ein Skript geschrieben. In dem Skript aus der Abbildung 1.13 werden erstmals die Ordner zum Speichern der Daten angelegt. Demnächst ....

## Listing 1.13: Skript fürs Erstellen einer Replikationsgruppe

```
#!/usr/bin/env bash

mkdir -p /data/rs1 /data/rs2 /data/rs3

// Start von drei lokalen mongod-Instanzen als Replikationsgruppe

mongod --replSet m101 --logpath "1.log" --dbpath /data/rs1 --port 27017
--oplogSize 64 --fork --smallfiles
mongod --replSet m101 --logpath "2.log" --dbpath /data/rs2 --port 27018
--oplogSize 64 --smallfiles --fork
mongod --replSet m101 --logpath "3.log" --dbpath /data/rs3 --port 27019
--oplogSize 64 --smallfiles --fork
```

Das Skript mit dem Inhalt aus Listing 1.13 ist mit dem aus Listing 1.14 auszuführen:

## Listing 1.14: Erstellen einer Replikationsgruppe anhand eines Skriptes

```
vlfa:scripts vlfa$ bash < create_replica_set.sh
```

Bei der Ausführung des Skriptes kann zu den Problemen führen. Um aktuelle Prozesse mit mongo anschauen und stoppen zu können, muss man folgenden Befehl angeben. The problem was that I have runned mongod without any parameters before I started launching the nodes. First kill all the mongo, mongod and mongos instances to guarantee the environment is clear.<http://stackoverflow.com/questions/25839559/mongodb-server-is-not-running-with-replset>

Find the mongodb process PID by typing: lsof -i:27017 assuming your mongodb is running on port 27017 Type kill <PID>, replace <PID> by the value you found the previous command.

## Listing 1.15: Auflistung aktueller mongo(s,d)-Prozesse

```
vlfa:scripts vlfa$ ps -ef | grep 'mongo'

oder besser mit

lsof -i:27017
```

Danach ist wichtig, Prozesse zu stoppen. Dafür muss man nach dem Befehl kill die ProzessID eingeben, siehe Listing 1.16. Dann wird die Möglichkeit fürs Erstellen eigener Replikationsgruppe ermöglicht, siehe dazu Listings 1.13 und 1.14.

#### Listing 1.16: mongo(s,d)-Server zwingend stoppen

```
// konkreten mongo(s,d)-Server zwingend stoppen  
vlfa:scripts vlfa$ kill 'PID'  
  
// alle mongo(s,d)-Server zwingend stoppen  
vlfa:scripts vlfa$ killall mongo(s,d)
```

Damit ist die Konfigurationsgruppe mit 3 Servern angelegt. Zum Anschauen einer log-Datei;

#### Listing 1.17: 1.log-Inhalt

```
2016-12-19T14:58:11.637+0100 I CONTROL [initandlisten] MongoDB starting :  
pid=25626 port=27017 dbpath=/data/rs1 64-bit host=vlfa.fritz.box  
// irrelevant  
2016-12-19T14:58:11.639+0100 I CONTROL [initandlisten] options:  
{ net: { port: 27017 }, processManagement: { fork: true }, replication:  
{ oplogSizeMB: 64, replSet: "m101" }, storage: { dbPath: "/data/rs1",  
mmapv1: { smallFiles: true } }, systemLog: { destination: "file", path: "1.log" } }  
// irrelevant
```

Die Replikationsgruppe starten.....blabla

#### Listing 1.1: Skript zum Start der Replikationsgruppe

```
config = { _id: "m101", members:[  
    { _id : 0, host : "localhost:27017", priority:0, slaveDelay:5},  
    { _id : 1, host : "localhost:27018"},  
    { _id : 2, host : "localhost:27019"} ]  
};  
  
rs.initiate(config);  
rs.status();
```

Die Server aus Listing 1.1 nehmen nun Kontakt miteinander auf, gründen die Gruppe und wählen den Primary-Server aus. Wie im Skript aus Listing 1.1 zu entnehmen ist, kann der Zustand der Replikationsgruppe mit `rs.status()` geprüft werden. Bei Ausfall des Primary-Servers wählen die Secondaries untereinander entsprechend einen neuen Primary-Server. Damit wird die Ausfallsicherheit des Servers erreicht. Die Mindestanzahl an Servern in einer Replikationsgruppe liegt bei drei.

#### Listing 1.18: Skript ausführen

```
vlfa:scripts vlfa$ mongo --port 27018 < init_replica.js
```

Die Priorität '0' teilt mit, wer Primary Member in der Replikationsgruppe ist. Korrigieren, Stimmt nicht....<https://docs.mongodb.com/v3.2/core/replica-set-priority-0-member/>

### 1.3.8 MongoDB mit Java

#### Listing 1.1: Verbindungsauflaufbau

```
1 public static void main(String[] args) {
2
3     MongoClient mongoClient = new MongoClient("localhost", 27017);
4     MongoDatabase db = mongoClient.getDatabase("test");
5     MongoCollection<Document> collectionOfZips = db.getCollection("zips");
6
7     // weitere CRUD-Operationen mit der ausgewählten Kollektion
8 }
```

#### Listing 1.2: Skript zur Initialisierung der Replikationsgruppe

```
1 public static void main (String[] args) throws InterruptedException {
2     MongoClient client = new MongoClient(asList(
3             new ServerAddress("localhost", 27017),
4             new ServerAddress("localhost", 27018),
5             new ServerAddress("localhost", 27019)));
6
7     // weitere CRUD-Operationen
8 }
```

**Listing 1.19: Simulation des Server-Ausfalls 'PRIMARY'**

```
m101:PRIMARY> rs.stepDown()

Result:

2016-12-19T21:24:12.739+0100 I NETWORK  [thread1]
trying reconnect to 127.0.0.1:27018 (127.0.0.1) failed
2016-12-19T21:24:12.760+0100 I NETWORK  [thread1]
reconnect 127.0.0.1:27018 (127.0.0.1) ok
m101:SECONDARY>
```

Der aktuelle MongoDB Java Treiber ist in Version 3.4.0 verfügbar und kann bequem als Maven Dependency geladen werden, siehe Listing 1.3.

**Listing 1.3: MongoDB Java Treiber in Version 3.4.0 als Maven Dependency**

```
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongo-java-driver</artifactId>
    <version>3.4.0</version>
</dependency>
```

Um die Sicherung der Zugehörigkeit der Mitglieder zu konkreter Replikationsgruppe festzustellen, siehe Listing 1.4, Zeilen 6-8...

**Listing 1.4: Sicherung der Zugehörigkeit zu konkreter Replikationsgruppe**

```
1  public static void main (String[] args) throws InterruptedException {
2      MongoClient client = new MongoClient(asList(
3          new ServerAddress("localhost", 27017),
4          new ServerAddress("localhost", 27018),
5          new ServerAddress("localhost", 27019)),
6          MongoClientOptions.builder()
7              .requiredReplicaSetName("m101")
8              .build());
```

# Kapitel 2

## Prototyp

Listing 2.1: Alle mongod-Prozesse zwingend stoppen

```
> killall mongod
```

### 2.1 Prototyp

Kommentare zu den Fotos hinzufügen, Eingebettete Kommentare, siehe <http://ezproxy.bib.fh-muenchen.de:2125/doi/pdf/10.3139/9783446431225.014>

### 2.2 Fazit

Siehe Listing 2.1

Doch wie der Begriff Not only SQL (NoSQL) andeutet, stehen beide Datenbanksysteme nicht unbedingt in direkter Konkurrenz zueinander, sondern können sich gegenseitig ergänzen. Dennoch, wenn es um die persistente Datenspeicherung bei Web-Anwendungen geht, stellen relationale Datenbanken nicht mehr die einzige Alternative dar. Bei eigenen Projekten wären Entwickler heute also gut beraten, die Vor- und Nachteile der beiden Systeme gegenüberzustellen und entsprechend den eigenen Anforderungen und Prioritäten zu bewerten. Muss das System mit großen Datenmengen effizient umgehen können? Werden hohe Anforderungen an Skalierbarkeit und Flexibilität der Datenbank gestellt?

Sollen sich die Daten über mehrere Server verteilen lassen? Sind häufige Änderungen an der Datenstruktur in Zukunft zu erwarten? Wenn Sie die meisten dieser Fragen mit "Ja" beantworten, dann sollten Sie sich MongoDB zumindest näher anschauen.

Daten in MongoDB verfügen über ein flexibles Schema. Kollektionen (=Collections) erzwingt keine Struktur.

## 2.3 Apache Cassandra

Cassandra<sup>1</sup> zählt, neben MongoDB<sup>2</sup>, zu den derzeit populärsten NoSQL-Datenbanken. Cassandra war ursprünglich eine proprietäre Datenbank von Facebook und wurde 2008 als Open-Source-Datenbank veröffentlicht. Beispiele für weitere NoSQL-Datenbanken sind SimpleDB<sup>3</sup>, Google Big Table<sup>4</sup>, Apache Hadoop<sup>5</sup>, MapReduce<sup>6</sup>, MemcacheDB<sup>7</sup> und Voldemort<sup>8</sup>. Unternehmen, die auf NoSQL setzen, sind unter anderem Netflix<sup>9</sup>, LinkedIn<sup>10</sup> und Twitter<sup>11, 12</sup>.

Cassandra ist als skalierbares, ausfallsicheres System für den Umgang mit großen Datensätzen auf verteilten Systemen (Clustern) konzipiert. Sie ist die beliebteste spaltenorientierte NoSQL-Datenbank und im Gegensatz zu MongoDB (C++) in Java geschrieben. Aufgrund ihrer architektonischen Eigenschaften wird Cassandra häufig in Big-Data-Projekten

---

<sup>1</sup>Apache Cassandra: <http://cassandra.apache.org>, zugegriffen am 16. Dezember 2016

<sup>2</sup>MongoDB: <https://www.mongodb.com>, zugegriffen am 16. Dezember 2016

<sup>3</sup>SimpleDB: <https://aws.amazon.com/de/simpledb/>, zugegriffen am 16. Dezember 2016

<sup>4</sup>Google Big Table: <https://research.google.com/archive/bigtable.html>, zugegriffen am 16. Dezember 2016

<sup>5</sup>Apache Hadoop: <http://hadoop.apache.org>, zugegriffen am 16. Dezember 2016

<sup>6</sup>MapReduce: <http://hortonworks.com/apache/mapreduce/>, zugegriffen am 16. Dezember 2016

<sup>7</sup>MemcacheDB: <http://memcachedb.org>, zugegriffen am 16. Dezember 2016

<sup>8</sup>Voldemort: <http://www.project-voldemort.com/voldemort/>, zugegriffen am 16. Dezember 2016

<sup>9</sup>NetFlix: <https://www.netflix.com/de-en/>

<sup>10</sup>LinkedIn: <https://www.linkedin.com/feed/>

<sup>11</sup>Twitter: <https://twitter.com/?lang=en>

<sup>12</sup>NoSQL: <http://www.searchenterprisesoftware.de/definition/NoSQL>, zugegriffen am 16. Dezember 2016

eingesetzt, kann in Zusammenarbeit mit einem Applikations-Server/Framework aber auch gut für komplexe Webanwendungen verwendet werden.

# Abbildungsverzeichnis

1.1	Skalierung . . . . .	3
1.2	CAP-Theorem . . . . .	5
1.3	NoSQL-Datenbanken . . . . .	9
1.4	NoSQL: A Silver Bullet for handling Big Data? . . . . .	9
1.5	Beispiel für die Darstellung der Daten in einer Graphen-Datenbank . . . . .	10
1.6	Zwei Dokumente im JSON Format . . . . .	11
1.7	MongoDB Architektur . . . . .	14
1.8	Indizes in <b>MongoDB</b> als Binär-Baum . . . . .	16
1.9	Ein Beispiel für Verteilung einer <i>Collection</i> auf mehreren <i>Shards</i> . . . . .	20
1.10	Horizontale Skalierung (Sharding) . . . . .	21
1.11	Szenario für eine Replikationsgruppe mit drei Servern in einer <i>Shard</i> . . . . .	22
1.12	Freischaltung der Lesezugriffe . . . . .	23

# Quelltextverzeichnis

0.1	Monitoring mit ab . . . . .	2
0.2	Monitoring mit siege . . . . .	2
1.1	Dokument(e) speichern . . . . .	14
1.2	Dokument(e) finden . . . . .	15
1.3	Dokument(e) aktualisieren . . . . .	15
1.4	Dokument(e) löschen . . . . .	15
1.5	Mongo-Shell: Something else . . . . .	17
1.6	Mongo-Shell: Something else . . . . .	17
1.7	Something else . . . . .	17
1.8	Mongo-Shell: Something else . . . . .	17
1.9	Mongo-Shell: Something else . . . . .	18
1.10	Mongo-Shell: Something else . . . . .	18
1.11	Something else . . . . .	18
1.12	Something else . . . . .	18
1.13	Skript fürs Erstellen einer Replikationsgruppe . . . . .	24
1.14	Erstellen einer Replikationsgruppe anhand eines Skriptes . . . . .	24
1.15	Auflistung aktueller mongo(s,d)-Prozesse . . . . .	24

---

1.16 mongo(s,d)-Server zwingend stoppen . . . . .	25
1.17 1.log-Inhalt . . . . .	25
1.1 Skript zum Start der Replikationsgruppe . . . . .	25
1.18 Skript ausführen . . . . .	26
1.1 Verbindungsaufbau . . . . .	26
1.2 Skript zur Initialisierung der Replikationsgruppe . . . . .	26
1.19 Simulation des Server-Ausfalls 'PRIMARY' . . . . .	27
1.3 MongoDB Java Treiber in Version 3.4.0 als Maven Dependency . . . . .	27
1.4 Sicherung der Zugehörigkeit zu konkreter Replikationsgruppe . . . . .	27
2.1 Alle mongod-Prozesse zwingend stoppen . . . . .	28