# Zusammenfassung zum Oberseminar Datenbanksysteme - Aktuelle Trends

NoSQL Datenbanken - Konzepte

Betreuer: Prof. Dr.-Ing. Thomas Kudraß

Christopher Hensel - 62427

20. Mai 2014

# Inhaltsverzeichnis

# 1 Einleitung

Zu Beginn soll eine kurze Erläuterung für den Begriff *NoSQL* stehen und ein kurzer Einblick in dessen Historie gegeben werden.

## 1.1 Begriffsdefinition

Der Begriff *NoSQL* wird oft missverstanden. Zerlegt man ihn in seine beiden offensichtlichen Bestandteile *No* und *SQL*, so versteht er sich als eine Ablehnung gegenüber der Anfragesprache *SQL*. Dies ist jedoch nicht für alle Systeme, die heute unter dem Begriff *NoSQL* geführt werden, tatsächlich der Fall. So hat sich auch die Bewegung hinter *NoSQL* dafür eingesetzt, *NoSQL* als *not only SQL* aufzufassen. Denn hinter *NoSQL* steht der Gedanke, neue Denkansätze abseits der klassischen relationalen Datenmodellen und Datenbanktechnologien zu etablieren.[EDL11]
Wie der Begriff sind auch die Eigenschaften, die ein *NoSQL*-System charakterisieren, nicht streng definiert. Übereinstimmend lassen sich jedoch Kernaussagen finden, von denen die meisten Systeme einige erfüllen. Zu diesen Kernaussagen zählen ein nicht relationales Datenmodell, der Verzicht auf ein Datenbankschema oder zumindest nur schwache Schemarestriktionen, die Auslegung für eine verteilte und horizontale Skalierbarkeit, einfache Replikation sowie ein alternatives Konsistenzmodell.[EDL11][WOL14]

Diese Grundeigenschaften lassen sich gut auf die Ursprünge der *NoSQL*-Datenbanken zurückführen, welche im Zeitalter des *Web 2.0* liegen. Aus diesem Grund soll im nächsten Abschnitt genauer auf diese Herkunft eingegangen werden.

## 1.2 Historie und die Ursprünge im Web 2.0

Bereits früh gab es Ansätze von Datenbanken, die ohne relationales Datenmodell oder eine SQL-API ausgelegt waren. Zu diesen zählen zum Beispiel die 1979 von Ken Thompson entwickelte *Key/Hash*-Datenbank *DBM* oder auch das Dokumentensystem *Lotus Notes*. Die NoSQL-Datenbanken, in ihrem Umfang und wie sie heute verstanden werden, setzten sich allerdings seit dem Jahr 2000 im Zuge des Web 2.0 durch. Das Web 2.0 und seine Begleiterscheinungen, wie die enorm anwachsende Datenmenge und -vielfalt, stellten immer mehr relationale Datenbanksystem vor große Schwierigkeiten. Ihre rein auf vertikale Skalierung ausgelegte Architektur machte es unmöglich, diese Datenmenge effizient und unter Einhaltung von Ausfallsicherheit und Reaktionszeit zu verarbeiten. Mit diesen Problemen konfrontiert arbeiteten immer mehr große Firmen, die große Web-Plattformen betreiben, an geeigneten Lösungsmöglichkeiten. Zu diesen Firmen zählen zum Beispiel Amazon, Facebook sowie Google. Vor allem Google hat mit seinen Entwicklungen wie *Map/Reduce*, *BigTable* oder auch dem Filesystem *GFS* eine Vorreiterrolle in Sachen *NoSQL* eingenommen. Auf dieser Basis entstanden ein Großteil der heutigen klassischen *NoSQL*-Systeme wie HBase, CouchDB, Cassandra, MongoDB, Redis und viele weitere.
Einen Überblick und die Kategorisierung der großen Anzahl an aktuellen *NoSQL*-Systemen, soll der nächsten Abschnitt bieten.

# 2 Unterschiedliche Kategorien in der Welt von NoSQL

Mit der immer weiter wachsenden Datenmenge wächst auch die Zahl an *NoSQL*-Datenbanken. Das *NoSQL-Archiv*, welches durch Stefan Edlich, den Autor von [EDL11], gepflegt wird, versucht eine strukturierte Übersicht über den aktuellen Markt an vorhandenen *NoSQL-Systemen* zu geben. Da es sich bei *NoSQL* um einen aktuellen Trend handelt, versuchen immer mehr Systeme auf den Zug aufzuspringen und in das *NoSQL-Archiv* aufgenommen zu werden. Dadurch steigt auch die Vielfalt an *NoSQL*-Systemen immer weiter an, weshalb es sich durchgesetzt hat, zwischen sogenannten *Kern-NoSQL-Systemen* und

den nachgelagerten *Soft-NoSQL-Systemen* zu unterscheiden.[EDL11]
Zu den aus [EDL11] entnommenen *Kern-NoSQL-Systemen* zählen hierbei:

- Key-Value-Datenbanken
- Large- oder auch Wide-Column-Datenbanken
- Dokumentendatenbanken
- Graphendatenbanken

Die ebenfalls aus [EDL11] entnommene Gruppe der nachgelagerten *Soft-NoSQL-Systemen* bilden:

- Objektdatenbanken
- XML-Datenbanken
- Grid-Datenbanken
- und weitere nichtrelationale Datenbankensysteme

Im folgenden richtet sich der Fokus jedoch ausschließlich auf die *NoSQL-Kernsysteme*, deren Konzepte in den folgenden Unterabschnitten eine kurze Erläuterung erfahren.

## 2.1 Key-Value-Datenbanken

*Key-Value-Datenbanken* besitzen ein Datenmodell vergleichbar mit einer *Map*, bei der einem Schlüssel, dem *Key*, ein Wert, der *Value*, zugeordnet wird. Das einfache Datenmodell ist die Stärke von *Key-Value-Datenbanken*. Es trägt zum einen zur einfachen Verständlichkeit bei und zum anderen ist dadurch eine schnelle und effiziente Datenverwaltung möglich. Auch eine horizontale Skalierung ist ohne Weiteres möglich, indem der Wertebereich der *Keys* aufgeteilt und auf unterschiedliche Server verteilt wird. Daher können *Key-Value-Datenbanken* ihre Stärke dort ausspielen, wo hohe Skalierbarkeit und häufige Lese- und Schreibzugriffe gefragt sind. Als Nachteile für das einfache Datenmodell ergibt sich die Beschränkung hinsichtlich der Komplexität von Strukturen, die es handhabbar abbilden kann. Komplexe Probleme und die damit verbundenen Datenmodelle können nur schwer verwaltet werden und führen schnell zu einer undurchsichtigen Applikationslogik, da dort alle Verbindungen hergestellt werden müssen. Aus diesem Grund sollten *Key-Value-Datenbanken* nur für schlanke Datenmodelle eingesetzt werden.[WOL14]
Es ist bei einigen *Key-Value-Systemen* zwar durchaus möglich, für die *Values* wiederum komplexere Werten wie *Listen*, *Sets* oder *Hashes* anstelle einfacher Zeichenketten zu verwenden, wodurch sie allerdings den *Large-Column-Systemen* ähneln.[EDL11]

Wie solche *Large-Column-Systeme* aufgebaut sind und warum deren Konzept besser für komplexere Datenmodelle geeignet ist, bildet das Thema des nächsten Abschnittes.

| KEY | VALUE |
|---|---|
| 23 | Venezia in der Hauptstr. 1 in Berlin |
| ... | ... |

Tabelle 1: Beispielhaftes Datenmodell für einen *Key-Value-Datenbanke*.

## 2.2 Large-Column-Datenbanken

Anders als in einer *Key-Value-Datenbank* können in einer *Large-Column-Datenbank* oder auch *Wide-Column-Datenbank* die Daten strukturierter abgelegt werden. Hierzu werden die Spalten (*Columns*) unter einem Zeilenschlüssel hinterlegt. Zueinander gehörende Spalten werden als sogenannte *Column-Family* zusammengefasst. Daher findet man *Large-Column-Datenbanken* häufig auch, wie in [EDL11], unter der Bezeichnung *Column-Family-Systeme*. Eine *Column-Family* ist mit dem Konzept einer Tabelle von relationalen Datenbanken vergleichbar, wobei jedoch im Gegensatz zu relationalen Tabellen zwischen *Column-Families* keine Beziehungen hergestellt werden können. Vorteil einer *Column-Family* hingegen ist, dass in jeder ihrer Zeilen verschiedene *Columns* gespeichert werden können.[WOL14]
Weitere Vertreter der *NoSQL-Datenbanken*, welche sich vor allem für semi-strukturierte Daten eignen, sind die im folgenden Unterabschnitt betrachteten *Dokumentendatenbanken*.

| | | | | | |
|---|---|---|---|---|---|
| Restaurant1 | name | street | city | zip | |
| | Venezia | Hauptstr. 1 | Berlin | 10115 | |
| Restaurant2 | name | street | city | zip | stars |
| | Haute Cuisine | Jahnstr. 210 | Hamburg | 21129 | 3 |
| Restaurant3 | city | | | | |
| | Leipzig | | | | |
| ... | ... | | | | |

Tabelle 2: Beispielhaftes Datenmodell für einen *Large-Column-Datenbanke*.

## 2.3 Dokumentendatenbanken

In *Dokumentendatebanken* werden die Daten als schemalose Dokumente zusammen mit einer ID hinterlegt. Als Formate für die strukturierten Dokumente kommen zum Beispiel *JSON* (siehe Listing 1), *RDF* oder *XML* zum Einsatz. *Dokumentendatebanken* sind besonders flexibel, da sich die Dokumente in ihrer Struktur vollkommen unterscheiden können und auch Strukturänderungen, die sich im Laufe der Zeit ergeben, keinerlei Problem darstellen. Zusätzlich können bekannte Verfahren zur Indexierung und Volltextsuche auf die Dokumente angewandt werden.[WOL14]

```
{
        "_id" :          23
        "type": "T-Shirt",
        "size": "M",
        "color": "blue"
}

{
        "_id" : 42
        "type": "Shoe",
        "size": "10",
        "material": "leather",
        "color": "brown"
}
```

Listing 1: Beispiel Datensatz einer Dokumentendatenbank

### 2.4 Graphendatenbanken

Als letztes Kernkonzept von *NoSQL* sind die *Graphendatenbanken* zu nennen. Sie bilden Beziehungen zwischen Daten durch Knoten und Kanten ab (siehe Abbildung 1). Hierzu werden den Knoten und Kanten Eigenschaften zugeordnet und als Tripel abgespeichert. So miteinander vernetzte Daten können schnell traversiert werden. Als Vorteil gibt es im Bereich der Graphentheorie viele bekannte Algorithmen, die auf die Daten einer *Graphendatenbank* angewendet werden können.

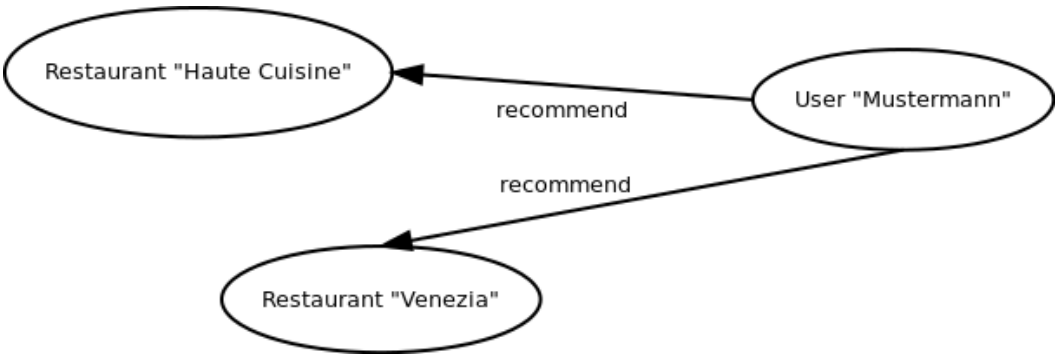


Abbildung 1: Datenmodell zu einer Graphendatenbank

## 3 Theoretische Grundlagen

Mit den neuen Anforderungen von *NoSQL-Systemen*, wie einfache horizontale Skalierung, Replikation und der Verarbeitung der damit verbundenen Datenmodelle, sind auch neue theoretische Ansätze betreffend der Konsistenz und Verarbeitungsmöglichkeiten notwendig. Eben diese theoretischen Grundlagen sind Gegenstand dieses Abschnittes.

Eine wichtige Fragestellung, die unmittelbar im Zusammenhang mit Replikation und Verteilung der Daten steht, ist die der Konsistenz.

### 3.1 CAP-Theorem

Um die Anforderung an Konsistenz in verteilten Systemen besser zu verstehen, hilft es zunächst einmal das *CAP-Theorem* von Eric Brewer näher zu betrachten. *CAP* steht für *Consistency* (Konsistenz), *Availability* (Verfügbarkeit) und *Partition Tolerance* (Ausfalltoleranz). Das Theorem besagt, dass es in einem verteilten System nicht möglich ist, alle der eben genannten drei Anforderungen zur selben Zeit zu erfüllen.[EDL11]
Die drei Größen sind für verteilte Datenbanken folgendermaßen zu verstehen:

- **Konsistenz**: Ein verteiltes Datenbanksystem ist dann in einem konsistenten Zustand, wenn nach einer Transaktion bei einem Lesezugriff jeder Knoten den selben aktuellen Wert zurückgibt. Bei replizierten Knoten würde dies bedeuten, dass erst dann wieder lesend zugegriffen werden kann, wenn alle replizierten Knoten auf dem aktuellen Stand sind. Dies kann unter Umständen bei einem großen Cluster mit hoher Last viel Zeit in Anspruch nehmen.[EDL11]
- **Verfügbarkeit**: Verfügbarkeit steht für eine annehmbare Reaktionszeit, welche mit unterschiedlichen Anwendungsszenarien variieren kann. Wird ein System als verfügbar bezeichnet, muss es auch unter Last eine vorher definierte Reaktionszeit einhalten.[EDL11]
- **Ausfalltoleranz**: Von einem ausfalltoleranten System ist dann die Rede, wenn weder der Ausfall eines Knotens noch eine Unterbrechung der Kommunikationsverbindung zwischen den einzelnen

Knoten zu einem kompletten Versagen des ganzen Systems führt. Es reagiert also trotz solcher Ausfälle weiterhin auf Anfragen, die von außen an das System gestellt werden. [EDL11]

Unter Berücksichtigung der oben genannten Größen kann es zu folgendem Szenario kommen: Besteht eine Replizierung der Daten auf mehrere Knoten in der verteilten Datenbank, so müssen die Knoten alle Änderungen stetig über das Netz austauschen.[WOL14]
Kommt es nun zu einem Ausfall der Kommunikationsverbindung zwischen den Knoten, muss sich ein Knoten, der nicht alle Änderungen hat, zwischen zwei Möglichkeiten entscheiden:

- Erstens: Er bearbeitet die Anfrage nicht, um die Herausgabe von inkonsistenten Daten zu vermeiden. Damit stellt er die Konsistenz über die Verfügbarkeit, da das ganze System solange still gelegt ist, bis wieder ein konsistenter Zustand erreicht ist.[WOL14]
- Die zweite Möglichkeit wäre mit dem veralteten Datenbestand zu arbeiten. Hiermit würde die Anforderung an die Konsistenz zugunsten der Verfügbarkeit und Ausfalltoleranz fallen gelassen werden. Das System wäre somit zu jeder Zeit erreichbar und würde eine Antwort liefern, auch wenn diese Antwort nicht zwangsläufig die selbe ist, wie die von allen anderen Knoten.[WOL14]

Als eine Lösung für den Konflikt setzen die meisten *NoSQL-Systeme* auf Letzteres und gehen Kompromisse in Sachen Konsistenz der Daten ein. Den Ansatz hierzu liefert das alternative Konsistenzprüfung *BASE*. Dieses Modell, welches konträr zum klassischen *ACID*-Modell steht, wird im folgenden näher beleuchtet.

## 3.2 BASE

Der Begriff *BASE* steht für *Basically Available, Soft-state, Eventually consistent* und enthält damit die Kernaussagen, für die dieses Konsistenmodell steht. Die erste, *Basically Available*, bezeichnet, dass das System grundsätzlich verfügbar ist. Der Zustand nach einer Transaktion kann jedoch „weich“ (*Soft-state*) sein, weil nicht zwingend Konsistenz gewährleistet ist und jeder Knoten eine unterschiedliche Antwort auf die selbe Anfrage liefern könnte. Die letzte Aussage *Eventually consistent* sagt allerdings aus, dass schlussendlich ein konsistenter Zustand erreicht wird.[WOL14] Wann dieser konsistente Zustand erreicht wird, ist dabei nicht festgelegt. Es ist nur festgelegt, dass irgendwann der Status der Konsistenz eintritt. Dazwischen liegt also eine Zeit, in der sich die verteilte Datenbank in einem inkonsistentem Zustand befindet. Wie lang dieser Zeitraum ist, hängt von verschiedenen Faktoren, wie Anzahl der Replikationen, durchschnittliche Reaktionszeit und durchschnittliche Last des Systems, ab.[EDL11] Vergleicht man diesen Ansatz nun mit dem *ACID-Modell*, welches in erster Linie Wert auf einen konsistenten Zustand der Daten nach jeder Transaktion legt, bleibt zu erwähnen, dass auch hier wie so oft gilt: Das Anwendungsgebiet entscheidet.

Neben den neuen Anforderungen an den Konsistenzbegriff, müssen auch andere bekannte Methoden, die zur Umsetzung für Datenbanken genutzt werden für die verteilten *NoSQL-Datenbanken* neu überdacht werden. Zu diesen Methoden zählen *Hashfunktionen*, welche mit *Consisten-Hashing* an diese Gegebenheiten angepasst wurden.

## 3.3 Consistent-Hashing

Hashfunktionen dienen im Allgemeinen dazu, einen Wert aus einer Quellmenge auf einen Hashwert aus einer kleineren Menge abzubilden. Vorteil einer solchen Hashfunktion ist, dass sie in konstanter Zeit ein Ergebnis liefert. Anwendung finden sie zum Beispiel in der Prüfsummenberechnung, der Kryptographie oder auch, um auf schnelle Art und Weise den Speicherort eines bestimmten Objektes auszumachen. Eben für das Berechnen eines solchen Speicherortes ist *Consistent Hashing* ein Lösungsversuch für verteilte

Systeme.[EDL11]
Im Vornherein ist es meist nicht möglich den genauen Bedarf an Speicherplätzen auszumachen. Die benötigte Anzahl an Speicherorten kann sich bei einer laufenden Anwendung immer wieder in beide Richtungen verändern. Bei einer Bedarfsänderung an Speicherorten kann es jedoch dazu kommen, dass ein Großteil der gespeicherten Daten umstrukturiert werden muss. Allerdings sind in verteilten Systemen die Datenmengen so umfangreich und die Transfergeschwindigkeiten über das Netz vergleichsweise langsam, dass solche Umstrukturierungen praktisch nicht machbar sind.[EDL11]
Zur Vermeidung solcher großen Datenbewegungen müssen folgende Kriterien erfüllt sein:

- Bei Hinzufügen eines neuen Knotens zur verteilten Datenbank soll nur ein geringer Prozentsatz der bisher vorhandenen Daten umkopiert werden müssen.[EDL11]
- Wird ein Server aus dem Verbund entfernt, so soll es genügen nur die dort abgelegten Daten neu zu verteilen.[EDL11]
- Vorteilhafterweise berücksichtigt das Hashverfahren unterschiedliche Speicherkapazitäten der einzelnen Knoten und verteilt die Daten gleichmäßig auf ihnen.[EDL11]

*Consistent Hashing* erfüllt eben diese Kriterien. Hierzu werden beim *Consistent Hashing* die Zielwerte der Hashfunktion nicht als Liste, sondern als Ring angeordnet. In diesem Ring werden zunächst alle Server anhand eines Hashwertes einer ihrer eindeutigen Merkmale wie IP-Adresse oder Servername eingeordnet. Die Objekte werden dann wiederum anhand ihres Hashwertes eingefügt. Welcher Server das Objekt aufnimmt, ergibt sich daraus, wessen Hashwert im Uhrzeigersinn der am nächstgelegene ist.[EDL11]
Durch dieses Verfahren hat auch das Hinzufügen oder Entfernen von Servern nur Einfluss auf die Objekte die logisch auf dem Ring in unmittelbarer Nähe des Hashwertes dieses Servers liegen. Ein neu hinzugefügter Knoten würde also alle Objekte aufnehmen, die zwischen dem Hashwert seines Vorgängerknotens und seinem eigenen liegen. Wird ein Server hingegen aus dem Verbund entfernt, so werden all seine Objekte auf seinen logischen Nachfolger im Ring kopiert.[EDL11]
Um nun noch das Kriterium der Berücksichtigung der unterschiedlich großen Speicherkapazitäten der Server zu erfüllen, wird ein weiterer Trick angewandt. Je größer die Kapazität des Servers, um so mehr Hashwerte werden für ihn berechnet. Verschiedene Hashwerte für den selben Server können einfach dadurch generiert werden, dass eine laufende Nummer an das verwendete Unterscheidungsmerkmal angefügt wird. Die so dem Ring als eine Art „virtuelle Server“ hinzugefügten Werte, führen zu einer statistisch besseren Verteilung von Objekten zu Servern.[EDL11]
Ebenfalls denkbar ist es im Verfahren von *Consistent Hashing* analog zu den virtuellen Servern mehrere Replikate des selben Objektes im Ring zu hinterlegen. Dies verbessert die Ausfallsicherheit des kompletten Systems und die Verfügbarkeit der einzelnen Objekte.[EDL11]
Das soeben vorgestellte Verfahren ermöglicht es in einer verteilten Architektur das effiziente Hashverfahren für den Zugriff auf ein Objekt zu nutzen.

Bei einem solchen Zugriff auf Daten, welcher von mehreren Nutzern gleichzeitig geschehen kann ist es wichtig die semantische Integrität der Daten über verschiedene Schreib- und Lesezugriffe hinweg sicherzustellen. Eine Strategie für die Behandlung solcher konkurrierenden Zugriffe bietet das im nächsten Abschnitt vorgestellte Verfahren *Multiversion Concurrency Control*, kurz *MVCC*.

## 3.4 Multiversion Concurrency Control

Greifen mehrere Anwender zur selben Zeit auf den gleichen Datenbestand zu und verändern ihn dabei, kann es leicht zu einer Verletzung der semantischen Integrität der manipulierten Daten kommen. Der naive Ansatz, den Datensatz während der Manipulation komplett zu sperren, funktioniert in jedem Fall;

allerdings kann es zu nicht unerheblichen Verzögerungen von parallelen Lesezugriffen kommen, wenn es häufig und lange zu Sperrungen kommt und das Beziehen einer solchen Sperre mit hohem Kommunikationsaufwand verbunden ist. Hier kommt *MVCC* ins Spiel, welches den Zugriff nicht mehr versucht mittels einer Sperre zu kontrollieren. Stattdessen hält es „[...] mehrere unveränderliche Versionen eines Datensatzes in einer zeitlichen Reihenfolge organisiert [...]“,[EDL11] vor. Eine neue Version entsteht durch jeden Schreibvorgang. Sie besteht aus einer eindeutigen Identifikationsnummer, den veränderten Datensatz und einen Verweis auf die Vorgängerversion, welche manipuliert wurde. Wie die Identifikationsnummer zusammengesetzt ist, kann sich von System zu System unterscheiden. Denkbar ist die Verwendung von Transaktionsstartzeitpunkt oder eine laufende Nummer aller Transaktionen.[EDL11]
Lesende Zugriffe werden somit nicht mehr blockiert, sondern können zu jedem Zeitpunkt mittels einer älteren Version bedient werden. Konflikte bei parallelen Schreibzugriffen können dadurch erkannt und behandelt werden, indem bei einer Transaktion die zu Beginn eingelesene Vorgängerversion mit der am Ende der Transaktion aktuellen Version verglichen wird. Sollten diese beiden Versionen nicht übereinstimmen, wird die Transaktion abgebrochen, alle Änderungen zurückgenommen und der Vorgang eventuell neu gestartet.[EDL11]
Bei genauerer Betrachtung des Verfahrens stellt man schnell fest, dass deutlich mehr Speicherplatz und Rechenzeit im Gegensatz zu einfachen Sperren benötigt wird. Weiterhin müssen nicht mehr in Verwendung stehende Datensatzversionen von Zeit zu Zeit beseitigt werden. Trotz dieses Mehraufwandes amortisieren sich die Kosten dafür jedoch schnell, insbesondere da die Datenbanken für den Fall eines Systemabsturzes oder Hardwareausfalls sowieso ein Protokoll über die vorgenommenen Änderungen führen und dieses für das *MVCC*-Verfahren genutzt werden kann.[EDL11]

Nachdem bisher einige Verfahren betrachtet wurden, die sich vor allem mit der Anpassung an die veränderten Bedienungen der Datenhaltung in der *NoSQL*-Welt beschäftigen, soll nun in Form von *Map/Reduce* eine Verarbeitungsmethode für die mit *NoSQL* einhergehende wachsende Datenmenge und -vielfalt erläutert werden.

## 3.5 Map/Reduce

Der Grundgedanke hinter dem Programmiermodell *Map/Reduce* ist Parallelität, welche es ermöglicht, große Datenmengen effizient zu verarbeiten. Im Wesentlichen bezeichnet Map/Reduce nicht nur einen Algorithmus, sondern auch ein Framework, welches zur Umsetzung des Verfahrens benötigt wird. Im Kontext von *NoSQL* wird das Verfahren von einigen Datenbank, wie CouchDB, MongoDB, Riak oder HBase zur Abfrage ihrer Einträge angewandt.[EDL11]
Konkret gliedert sich das *Map/Reduce*-Verfahren in die beiden im Namen enthaltenen Phasen *Map* und *Reduce*. Der Ablauf des Verfahrens lässt sich am Besten anhand des in Abbildung 2 dargestellten Datenflussmodells erläutern.
Zu Beginn werden in der *Map-Phase* die Eingabedaten mittels einer Funktion *Map* verarbeitet. Die hierbei gewonnenen Zwischenergebnisse dienen wiederum als Eingabe für die *Reduce*-Funktion. Sie fasst in der zweiten Phase alle Zwischenergebnisse zum Endergebnis zusammen.

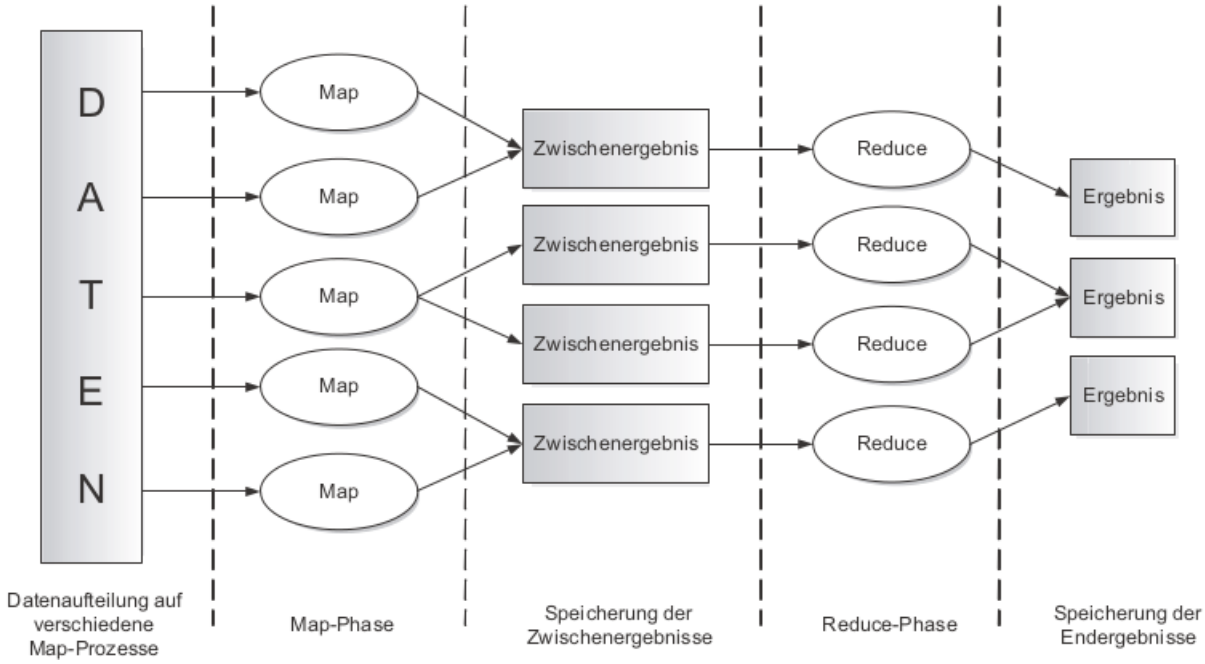


Abbildung 2: Datenflussmodell für das *Map/Reduce*-Verfahren, Quelle: [EDL11]

Beide Funktionen werden in den zwei hintereinander ablaufenden Phasen auf allen Knoten des verteilten Systems und auf unterschiedliche Eingabedaten gleichzeitig ausgeführt. Dadurch ergibt sich eine massive Parallelität, welche ein hochperformantes Vorgehen ermöglicht.[EDL11]
Mit diesem Vorgehen lassen sich Anwendungen wie, verteiltes Suchen, das Zählen von Wörtern und noch viele weitere Beispiele effizient Implementieren.[EDL11]

## 3.6 REST

*REST*, die Abkürzung für *Representational State Transfer*, bezeichnet einen Architekturstil. Erstmals wurde er von Roy Fielding im Rahmen seiner Disseration eingeführt. Hierzu leitete er den Achritekturstil anhand der bestehenden Architektur des Webs ab.[FIE00]
Als anerkanntes und erfolgreiches Entwurfsmuster für verteilte Architekturen, die hohe Anforderungen an eine horizontale Skalierbarkeit besitzen, setzte es sich auch als Ansatz für *NoSQL*-Systeme durch. *NoSQL*-Datenbanken, die den REST-Entwurfsansatz folgen, sind zum Beispiel *CouchDB*, *Riak* oder *Neo4J*.[EDL11]
Da eine volle Erläuterung von *REST* den Rahmen dieser Arbeit überschreiten würde, soll sich im Folgenden auf die grundlegenden Kernaussagen beschränkt werden, die für das Verständnis in Verbindung mit *NoSQL*-Systemen wichtig sind. So soll sich die Beschreibung von *REST* in dieser Arbeit auf die Umsetzung mit HTTP beschränken. Auch die im folgenden als Beispiel dienende *NoSQL*-Datenbank mit *REST* Implementierung *CouchDB* ermöglicht einen Komplettzugriff über HTTP/REST.[EDL11] Eine erste wichtige Aussage von *REST* ist die Statuslosigkeit. Jede Anfrage wird unabhängig ausgeführt, wodurch sie sich ohne Probleme auf mehrere Server verteilen lassen. Weiterhin sollen die drei Grundkomponente von *REST*, *Ressourcen*, *Operationen* und *Links*, näher betrachtet werden und wie diese für die *NoSQL*-Datenbank *CouchDB* umgesetzt wurden.[EDL11]
Eine *Ressource* steht in *REST* für einen adressierbaren Endpunkt, über den mit dem System interagiert werden kann. *Ressourcen* können dabei vielgestaltig sein, vom bekannten *HTML*-Dokument über Bilder bis hin zu Prozesschritten ist alles denkbar. Die Ressource ist über einen eindeutigen Identifizierer, den sogenannten Uniform Ressource Identifier (URI), ansprechbar. Spricht ein Nutzer eine Ressource an, so erhält er immer eine Repräsentation von ihr. Dabei ist es durchaus denkbar, dass eine Ressource mehrere

unterschiedliche Repräsentationen besitzt. Im Falle von *CouchDB* ist die Ressource ein *JSON* kodiertes Dokument, welches unter einer ID im System abgelegt ist.[EDL11]
Um mit *Ressourcen* arbeiten zu können, sind *Operationen* von Nöten, welche es erst ermöglichen das System zu benutzen. Dafür gibt es unter *HTTP* die bekannten Standardmethoden *GET*, *HEAD*, *PUT*, *POST* und *DELETE*. Durch diese Methoden werden die grundsätzlichen Methoden zum Erstellen, Abrufen, Ändern und Löschen von Ressourcen umgesetzt. So stellt auch *CouchDB* diese Methoden mittels seiner *HTTP/REST*-API dem Benutzer zur Verfügung.[EDL11]
Als letzter hier vorgestellter Baustein von *REST* sind die *Links* zu nennen. *Links* dienen der Verknüpfung von Ressourcen zu einem Netz. *HTTP* selbst bietet kein direktes Konzept für *Links*, sie werden als Inhalte in den *Ressourcen* abgebildet. Die dokumentenorientierte Datenbank *CouchDB* bietet sogar keinerlei Unterstützung für *Links* an. Jedoch besteht die Möglichkeit sie als entsprechende Felder mit passenden Verweisen auf andere *Ressourcen* zu modellieren.[EDL11]

Mit *REST* wurde der letzte Punkt des Abschnittes zu den Theoretischen Grundlagen behandelt, woraufhin eine kurze Zusammenfassung den Abschluss bilden soll.

# 4 Zusammenfassung

Zusammenfassend lässt sich sagen, dass mit *NoSQL* eine passende Antwort auf die veränderten Bedürfnisse des Web 2.0 geschaffen wurde. Zu diesen zählen hohe horizontale Skalierbarkeit, die Verarbeitungen von großer Datenmenge und -vielfalt und das, ohne Verzicht auf hohe Performance. Wie auch der Begriff nicht starr definiert ist, so finden sich auch viele verschiedene Kategorien von *NoSQL*-Systeme, wie *Key/Value-Datenbanken*, *Large-Column-Datenbanken*, *Dokumenten- und Graphendatenbanken*, die diese Herausforderungen angehen. Um die Anforderungen erfolgreich zu erfüllen, setzen sie nicht nur auf neue Datenmodelle abseits des relationalen Modells, sondern es kommen auch Verfahren wie das alternative Konsistenzmodell *BASE*, *Map/Reduce*, *Consistent Hashing*, *MVCC* und *REST* zum Einsatz. Nichts desto trotz bleibt dem Anwender darauf zu achten, die passende Datenbank für das eigene Anwendungsgebiet zu wählen. Denn neben den Eigenschaften von hoher Skalierbarkeit und einfacher Replikation bieten gerade alternative Datenmodelle wie Graphen- oder Dokumentendatenbanken ganz neue Herangehensweise zur Verarbeitung der Daten und daraus resultierende Vorteile in der Entwicklung.

# Tabellenverzeichnis



# Abbildungsverzeichnis



# Listings



# Literatur

[EDL11] Stefan Edlich, et al.: NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken (2. Auflage), Hanser Verlag, München, 2011.

[FIE00] R. Fielding: Architectural Styles andthe Design of Network-based Software Architectures, UNIVERSITY OF CALIFORNIA, IRVINE, http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (letzter Zugriff: 18.05.2014).

[WOL14] Eberhard Wolff, et al.: "NoSQL-Architekturen: Wie sich die neuen Datenbanken auswirken", OBJEKTspektrum, Heft Nr. 3, Mai/Juni 2014, 34-39.