Ausarbeitung Anwendungen 1 - WiSe 2011

Andreas Winschu

NoSQL -Hintergruende und Anwendungen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Inhaltsverzeichnis

1 Einführung in das Themengebiet

Es ist nicht selten der Fall, dass wenn man einen etwas weit in der Zeit zurückliegenden Artikel auf dem Gebiet der Informatik liest, einfach ein wenig lächeln muss. Nicht etwa, weil der Inhalt in irgendeiner Weise komisch oder nicht tiefgründig genug ist, sondern weil wegen rasant fortschreitender Entwicklung die Relevanz des betreffenden Problems sich einfach aufgelöst hat. Es gibt jedoch auf der anderen Seite einige Konzepte in der Computerwissenschaft, die die Ewigkeit zu überdauern scheinen. Oft nicht Mal wegen ihrer Perfektion oder des Anspruchs auf die einzig richtige Lösung, sondern schlicht aus dem Grund, dass Industriestandards in ihren Implementierungen darauf setzen und eine Umstellung auf ein neues Paradigma verhindern.

Zu einem solcher Meisterwerke zählen die Relationalen Datenbank Management Systeme (RDBMS). Das Konzept tauchte in den 70-ger Jahren auf, und löste einen immens großen Hype aus. Es war verglichen mit damalig vorherrschenden hierarchischen System, die auf low-level Daten basierten, deutlich einfacher zu verstehen. Dafür dass schließlich aus dem Hype eine Standardtechnologie wurde, sorgte nicht nur das simple Paradigma, Daten in Spalten und Zeilen von Tabellen abzulegen, sondern nicht zuletzt auch die neue Abfragesprache SQL. Diese erlaubte verglichen zu damaligen Mechanismen mit deutlich weniger Zeilen Code das gewünschte Ergebnis auszudrücken. Ein ausführliche Darstellung dieser Historie findet sich in Anthes (2010).

Als in den 80-gern die Objektorientierung das Paradigma der Softwareentwicklung wurde, verbreiteten sich ebenfalls Anstrengungen, das gleiche Paradigma in die Welt der Datenbanken zu übertragen. Dieses Mal kam es aber bei Weitem nicht zu dem gleichen Erfolg. Bei dem Verfahren Objektorientierte Daten in RDBMS dauerhaft zu persistieren, wird oft folgende Metapher verwendet: Ës ist als würde man ein Fahrzeug in seine Bestandteile auseinander nehmen, jedes Mal bevor man es in die Garage abstellt, und wieder zusammenbauen wenn man es benutzen möchte". Dennoch obwohl die objektorientierten Datenbanksysteme gute Konzepte lieferten um auf den manuellen Auseinander- un Zusammenbau auf Softwareebene zu verzichten, blieben die Entwickler zwecks Aufwandminimierung bei den altbewährten Mechanismen. Und so gilt es bis heute, dass Jeder, angefangen bei dem Ersteller eines Webforums oder Blogs bis hin zu den großen ECommerce Konzernen, auf die gleiche Lösung - RDBMS - für die Datenspeicherung zurückgreift.

Aktuell wird erneut die Autorität der RDBMS in der Wissenschaft auf die Probe gestellt. Motiviert ist diese Auseinadersetzung hauptsächlich von neuen datenhungrigen Webanwendungen wie sozialen Netzwerken oder den Synchronisationsanwendungen zwischen mobilen und üblichen Endgeräten. Diesen Anwendungen ist stets gemein, dass ihre Daten auf mehrere physikalische Endknoten verteilt werden müssen. Im Falle der Social-Webanwendungen zwecks schnellerer Antwortzeiten bei hoher Last und parallelen Zugriffen, und bei dem Synchronisationsszenario zwecks Verfügbarkeit gleicher Daten auf beiden Geräten. Eine zweite Gemeinsamkeit besteht in den toleranten Anforderungen dieser Applikationen an die Korrektheit und Aktualität dieser Daten. Während RDBMS im Gegensatz dazu stets die Korrektheit und Aktualität der Daten im Fokus haben, haben Sie per Definition eine Schwachstelle bei der Partitionierung der Daten auf mehrere physikalische Instanzen. Diese Schwachstelle machte sich in den oben genannten Anwendungen bemerkbar und führte zu Entstehung sog. NoSQL Datenbanksysteme.

Diese Arbeit beschäftigt sich mit der Schwachstelle der RDBMS bei den oben angedeuteten Szenarien, zeigt wie NoSQl Systeme verspechen dieses Problem zu lösen und zeigt shließlich Anwendugnsszenarien auf, an denen die NoSQl Paradigemen weiterhin erforscht werden können.

2 Hauptteil

2.1 RDBMS

. . . Dieser Abschnitt stellt zunächst ein kurzes Resume des bekannten Aufbaus relationaler Datenbaken. Ferner werden Problematiken angedeuted, die bei der Partitionierung relationaler Datenbanken auftreten.

Bücher

Buch-ID	Autor	Verlag	Verlagsjahr	Titel	Datum
1	Hans Vielschreiber	Musterverlag	2007	Wir lernen SQL	13.01.2007
2	J. Gutenberg	Gutenberg und Co.	1452	Drucken leicht gemacht	01.01.1452
3	G. I. Caesar	Handschriftverlag	-44	Mein Leben mit Asterix	16.03.1944
5	Galileo Galilei	Inquisition International	1640	Eppur si muove	1641
6	Charles Darwin	Vatikan-Verlag	1860	Adam und Eva	1862
7	Christian Schade	KVG	2008	SQL für Dummies	04.06.2007

BücherNutzer

Nutzer-ID	Buch-ID
10	1
10	2
10	3
12	5
12	6

Nutzer

Nutzer-ID	Vorname	Nachname
10	Hans	Vielleser
11	Jens	Mittelleser
12	Erich	Wenigleser

Abbildung 1: Relationales Schema

Bekannterweise handhaben relationale Datenbanken ihre Daten in festen Tabellen Schemata. (siehe Abb. 1). Jede Zeile in so einer Tabelle stellt ein Datensatz dar und jede Spalte ein Datumsattribut. Datensätze verfügen über einen eindeutigen Primärschlüssel (e.g Bücher-ID für Tabelle Bücher) und können in anderen Tabellen über diesen referenziert werden um Beziehungen zwischen den Daten abzubilden (siehe Tabelle BücherNutzer in Abb. 1).

Um diese Beziehungsreferenzen zusammenzuführen und vollständige Datensätze anzeigen zu können (e.g. Jedes Buch mit allen Personen die es gelesen haben) werden sog. JOIN Ope-

rationen zwischen den einzelnen Tabellen anhand der Primärschlüssel und deren Referenzen (Fremdschlüssel) ausgeführt. Dieses ist der in der Einleitung erwähnte Zerlegungs- und Zusammensetzungsmechanismus welcher für eine redundanzfreie Speicherung der Daten oder Normalisierung, wie es in RDBMS Sprache heißt, essentiel ist. Der Vorteil wird logischerweise durch CPU Rechenzeit für die JOIN Operationen erkauft. Werden Tabellen auf mehrere Instanzen partitioniert, so liegt es auf der Hand, dass der Overhead für verteilte JOIN's in die Höhe steigt.

Bei der Datenverwaltung sorgt ein RDBMS immer dafür, dass die Daten trotz Ausfälle oder parallelen Zugriff korrekt bleiben. Hierfür wendet es ein Transaktionen Konzept an, wobei mehrere Datenveränderungen zu einer Transaktion zusammengefasst werden können. Dabei werden jeder Transaktion sog. ACID Eigenschaften zugeschrieben. Grob gesagt sorgen diese Eigenschaften für eine stetige Konsistenzerhaltung - starke Konsistenz. Z.B. garantieren entsprechende Mechanismen bei einer Geldüberweisung vom Konto A auf ein Konto B, dass stets die Operationen „Abheben von A“ als auch „Gutschreiben auf B“ gemeinsam ausgeführt werden, falls diese in einer Transaktion zusammengefasst werden.

Solange die gesamte Datenbank sich auf einer physikalischer Maschine befindet, sind die ACID Garantien leicht einzuhalten. [Abadi (2010)]zeigt weshalb es zu schwierigen Problemen kommen kann, sobald die Daten auf mehrere physikalische Insatnzen verteilt werden.

Angenommen, es soll eine Geldüberweisung auf mehrere Konten stattfinden, die alle auf verschiedenen Nodes einer Datenbank gespeichert sind. Damit so eine Operation niemals zu inkonsistenten Daten führt müssen alle Teilnehmer dieser Transaktion über den Erfolg oder Misserfolg der einzelnen Teiloperationen einig sein. Misslingt eine Teiloperation, so muss die gesammte verteilte Transaktion abgebrochen werden (Atomicity). Um dies zu garantieren wird ein sog. verteiltes 2-Phasen-Commit Protokoll verwendet, bei dem ein Coordinator alle Teilnehmer der Transaktion in der ersten Phase vorbereitet und in einer anschließenden Phase einen Feedback von allen Teilnehmern einholt, um über den Gesamtzustand der verteilten Transaktion zu entscheiden(Abb. 2).

Andererseits wird durch ACID ebenfalls garantiert, dass nebenläufige Transaktionen nichts von einander bemerken. In dem obigen Beispiel dürfte also kein Besitzer des jeweiligen Kontos den neuen Kontostand sehen bevor nicht die gesamte Transaktion abgeschlossen ist(Isolation). Dies bedeutet, dass Locks über die gesamte Dauer der verteilten Transaktion erhalten werden müssen. Da es oft der Fall ist, dass eine verteilte Transaktion aus mehreren leichtgewichtigen Transaktionen zusammengesetzt ist und durchaus mehrere Netzwerkroundtrips für den Kommunikationsoverhead benötigt werden, kann es dazu kommen, dass Locks für einen längeren Zeitraum erhalten bleiben als die einzelne Transatktion tatsächlich benötigen würde um die ihre gesperrten Items zu aktualisieren, was zu „skyrocketing“ Lock Kämpfen zwischen den Transaktionen führen kann [vgl.Abadi (2010)].

Fazit: RDBMS definieren ein festes Datenmodell, sorgen für Redundanzvermeidung duch Fremddschlüssel Beziehungen und garantieren vor Allem starke Konsistenz der Daten durch das transaktionelle Modell. Jedoch wachsen die Schwierigkeiten diese Garantien einzuhalten mit zunehmender Partitionierung der Daten auf verteilte Datenbanknodes.

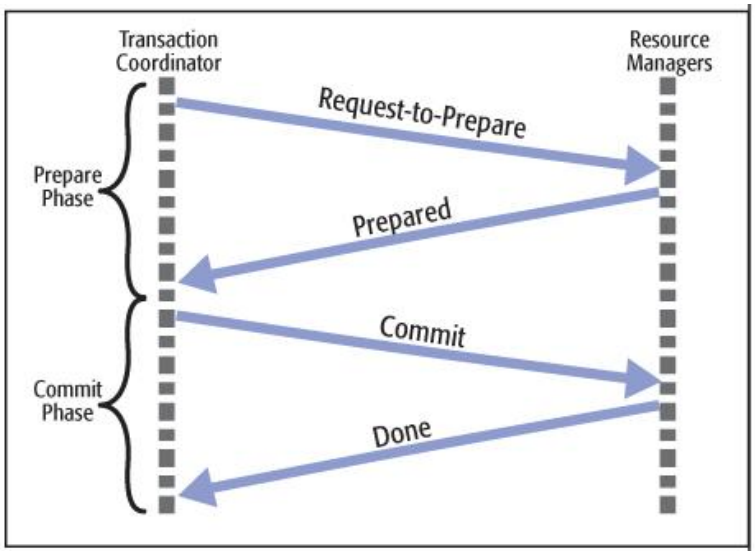

Abbildung 2: 2-Phasen-Commit Protokoll [Kaye (2004)]

2.2 CAP Theorem

Der vorherige Abschnitt deutete Schwierigkeiten in verteilten RDBMS an. In diesem Anschnitt geht es um das CAP-Theorem. Dieses geht eine Stufe weiter und besagt, das RDBMS wegen ihres Strebens nach starker Konsistenz bei Partitionierung ihre Garantien überhaupt nicht einhalten können.

Eric Brewer, ein Professor an der University of California, machte die Behautptung, dass verteilte Datenbanken niemals alle drei, sondern nur zwei der folgenden Garantien einhalten können:

- Consitency - Es wird eine totale Ordnung der Operationen eingehalten und eine Menge von Operationen wird als Ganzes auf ein Mal ausgeführt. (gemeint sind die ACID Eigenschaften von Transaktionen)
- Availabiliity - Jede Afrage wird beantwortet.
- Partition Tolerance - Operationen führen zu einem korrektem Ergebnis, auch wenn einzelen Komponenten des System's nicht verfügbar sind. (als Partitions sind nicht die einzelnen Partitionen der Datenbank, sondern die Fehler in deren Kommunikation gemeint)

Es bilden sich also drei Klassen von Datenbanksystemen mit jeweils zwei der CAP Eigenschaften - CA, CP und AP (siehe Abb. 3). Logischerweise, wenn es also um Verteilung der Daten über mehrere Knoten geht, muss entweder Consistency oder Availibility aufgegeben werden. Dies können RDBMS nicht leisten, denn sie fallen per Definition in die CA Klasse. Eine ausführliche Auseinadersetzung mit dieser Problematik findet sich in [Pritchett (2008)].

Das CAP Theorem gilt Allgemein als bewiesen. Ein formaler Beweis findet sich im Paper von [Gilbert und Lynch (2011)]. Logisch stützt sich der Beweis vereinfacht auf etwa folgende Argumentation:

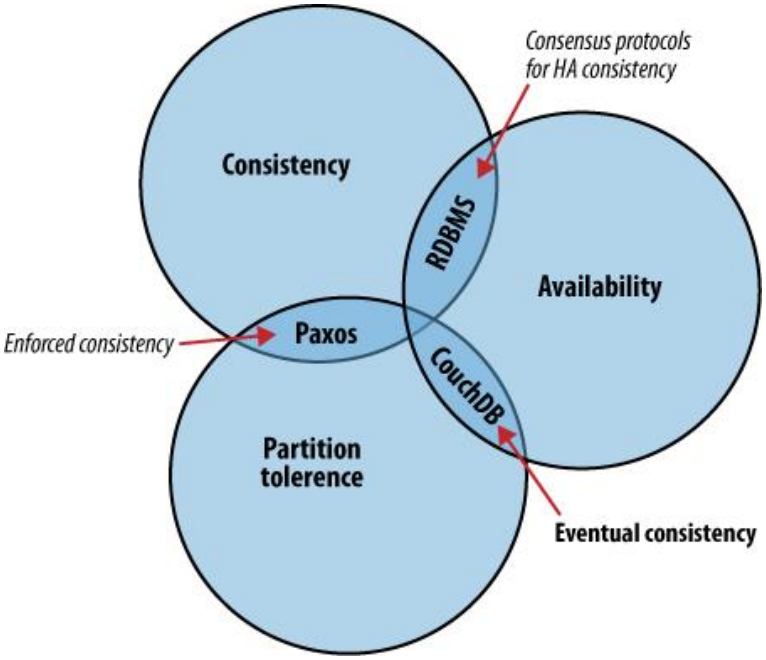

Abbildung 3: CAP-Theorem [Andersen (2010)]

Man geht davon aus, dass das System erstens verteilt ist. Dies bedeutet, dass Daten mittels verteilter Hashfunktionen auf mehrere Nodes gestreut werden. Zweitens soll gelten, dass Partition Tolerance garantiert werden soll. Dies wiederum heißt, dass einzelne Nodes zusätzlich repliziert werden müssen, damit die Daten trotz Netzwerkausfälle verfügbar sind (Siehe auch [Henderson (2006), Chapter 9] zum Thema „Scalierung von Datenbanken“).

Nun treten die Netzwerkfehler(Partitions) auf. Möchte das System gleichzeitig Availability garantieren (AP), so bedeuted dies, dass es trotz ausgefallener Nodes und unbeendeter Replikationsvorgänge immer eine Antwort liefern muss. Somit lautet die logische Schlussfolgerung, dass diese Antwort auch nicht aktuelle Daten enthalten kann. Consitency kann demnach nicht gleichzeitig eingehalten werden. Man stelle sich hierfür ein kollektives Nachrichtensystem, bei dem verschiedene Benutzer parallel an unterschiedlichen Standorten - A und B - neue Beiträge eintragen können. Die Datenbank ist also auf Standort A und B verteilt. Bei einem Netzwerkfehler in der Kommunikation zwischen A und B sowie einer gleichzeitiger Anfrage nach alle neuen Beitragen in A, können nicht alle neuen Nachrichten angezeigt werden, da unbekannt ist ob neue Updates inzwischen in B stattgefunden haben. Da jedoch das System immer verfügbar ist, antwortet es nur mit den in A bekannten Daten, welche somit inkonsistent sind.

Gleiche Argumentation gilt für die CP Klasse. Wird neben Partition Tolerance gleichzeitig Consistency garantiert, so kann das System nicht immer verfügbar sein. Denn um immer konsistente Antworten liefern zu können, müssen alle Synchronisationsvorgänge beendet sein, was bei Netzwerkfehlern nicht sicher gestellt werden kann. Somit können einzelne Anfragen nicht beantwortet werden - Availability wird aufgegeben. Bei dem obigen Beispiel bedeutet es also, dass das System im Falle eines Netzwerkfehlers zwischen A und B nicht antworten darf, da alle Benutzer

stets alle aktuellen Nachrichten sehen müssen. Es ist demnach solange nicht verfügbar bis eine synchronisation zwischen A und B stattgefunden hat.

Brewer benutzte das CAP Theorem um ein gegensätzliches Paradigma zu dem ACID Model zu definieren und nannte es BASE. Beide Akronyme sind künstlich, so dass nicht jeder einzelne Buchstabe analysiert werden muss. Im Allgemeinen kann gesagt werden, dass während ACID für starke Konsistenz und pessimistische Annahmen („Operation wird möglicherweise scheitern“) steht, BASE für schwache Konsistenz (auch Eventual Consistency) und optimistische Annahmen („Versuch dein Bestes“ Metapher) eintritt. Siehe auch [Pritchett (2008)].

Heutzutage durchlebt das BASE Paradigma eine Wiedergeburt. Es finden sich Anwendungen, welche auf starke Konsistenz von relationalen Datenbanken verzichten können, jedoch auf mehrere physikalische Nodes zwecks schneller Antwortzeiten scaliert werden müssen. Und dies nicht nur im LAN sondern evtl. über das Internet oder über die Cloud, wie es modern genannt wird. Um diesen Anwendungen gerecht zu werden, entstehen momentan mehrere nichtrelationale Datenbanksysteme, welche unter dem Oberbegriff NoSQL zusammengefasst werden. Das nächste Kapitel beschäftigt sich mit diesen Systemen.

2.3 NoSQL

Der vorherige Abschnitt argumentierte mit dem CAP Theorem für einen Bedarf nach nicht relationalen Datenbanken, welche unter dem Oberbegriff NoSQL zusammengefasst werden. Nun sollen diese Systeme und ihre Stellungen zu dem CAP Theorem betrachtet werden.

Überblick

Der Begriff „NoSQL“ wurde erstmals 1998 durch Carlo Strozzi verwendet. Jedoch war von Ihm eine relationale Datenbank gemeint, welche tatsächlich auf SQL als Abfragesprache verzichtete. „NoSQL“ stand also für „No to SQL“. In 2009 wurde der Begriff von Johan Oskarsson, damals Entwickler von Last.fm, aufgegriffen. Nun wurden damit aber alle Datenbanksysteme angesprochen, welche auf das relationale Modell und besonders auf die ACID Eigenschaften verzichteten. „NoSQL“ wurde also im Sinne von „Not only SQL“ verwendet.

Die Art und Weise, wie einzelne Entwickler konkret „NoSQL“ umsetzen variiert stark. Es soll eine überschaubare Charakteresierung der bis dato entstanden Lösungen folgen.

Folgende Eigenschaften sind in den meisten NoSQL Systemen vertreten:

- Schemafreiheit - Es gibt keine feste Datenstruktur, die vor der Speicherung der Daten definiert ist. Der Entwickler bestimmt diese zur Laufzeit. Ferner gilt meist der Ansatz „Embed over Reference“. Dabei soll nach Möglichkeit auf Auslagern von Datenstrukturen in separate Dokumente wie in RDBMS verzichtet werden. Enthaltene Unterdokumente werden ähnlich wie bei Objektorientierung in das Hauptdokument eingebunden.

- „Einfache“ Schnittstellen - In den Meisten Fällen wird in der Tat auf SQL als Abfrage Mechanismus. Der Zugriff erfolgt über REST Schnittstellen oder senden von kompakten Query Hash Objekten mit kurzen Query Strings.
- Horizontale Scalierbarkeit(Shards) - alle Hersteller werben mit automatischer Funktion zu Verteilung der Daten auf mehrere physikalische Computer Nodes (Shards).
- Replikation - einzelne Shards werden kontinuierlich gespiegelt
- Map/Reduce - dies ist ein Paradigma um Suchanfragen auf verteilten Nodes parallel auszuführen (Map) und zu einem Gesamtergerbins zusammenzufügen (Reduce). Siehe dazu [Dean und Ghemawat (2010)]
- Partition Tolerance - alle NoSQL Systeme behaupte das „P“ aus dem CAP Theorem einzuhalten.

Bei der Art und Weise, wie diese Systeme ihre Daten organisieren, lassen sich diese Systeme in drei Kategorien unterteilen:

- Key-Value Stores - Eine einfache verteilte Hashtabelle mit Zugriff über den Schlüssel. Der Schlüsselwert ist von der Datenbank uninterpretierbares binäres Objeket. (Vertreter: Voldemort, Redis, Tokio Cabinet)
- Column Based Stores - Dateneinträge in Tabellenspalten (im Gegensatz zu zu Tabellenzeilen in RDBMS).
- Document Stores - ähnlich wie Key-Value Stores mit dem Unterschied, dass der der Wert ein von der Datenbank interpretierbares, durchsuchbares Dokument mit Baumstruktur darstellt.(Vertreter: CouchDB, MongoDB)

Eine ausführliche Übersicht über alle vertretenen Lösungen liefert [Cartell (2011)].

NoSQL und CAP

Es wurden typische Eigenschaften der NoSQL Datenbanksystem erläutert, wobei festgestellt wurde, dass diese die „P“ Eigenschaft des CAP Theorems garantieren. In diesem Abschnitt soll verdeutlicht werden, was es bedeutet, auf eine der beiden restlichen Eigenschaften zu verzichten.

Da Partition Tolerance garantiert wird, müssen NoSQL Systeme entweder Consistency (AP Class) oder Availablity(CP Class) aufgeben. Die meisten Vertreter entscheiden sich für die AP Klasse. Konsistenz aufgeben, bedeutet, wie zuvor erwähnt, sich im Zweifelsfall für nicht aktuelle Daten zu entscheiden. Man möchte dabei jedoch Consistency nicht absolut aufgeben, sondern den bestmöglichen Grad an Konsistenz einhalten. Diese Anstrengung wird unter dem Begriff „Eventual Consistency“ zusammengefasst.

Folgende Eigenschaften können einem System mit Eventual Consistency zugesprochen werden:

- Keine Locks - das System errichtet keine Speeren für Lese oder Schreibzugriffe auf Entitäten.

- Local Consistency - das System sorgt, eine einzelne Node immer mit ihren aktuellsten Daten antwortet.
- Multi Version Concurrency Control - das System speichert bei Updates eine neue Version der Entity, so dass parallel laufende Lesezugriffe durch zwischenzeitliche Updates nicht gestört werden, also während des gesamten Zugriffs die alte Version lesen.
- „Read your own Writes“ - Eigene Schreibzugriffe können sofort gelesen werden.

Das ganze soll an einem anschaulichen Beispiel betrachtet werden (siehe Abb. 4).

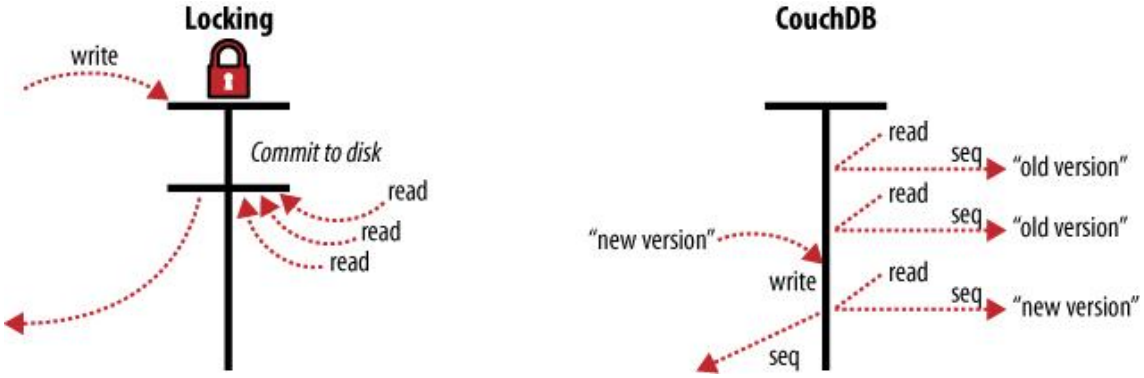

Abbildung 4: Eventual Consistency mit MVCC [Andersen (2010)]

Angenohmmen eine Menge von Requests will auf ein Entity zugreifen. Der erste Request liest das Dokument. Während dies geschieht, verändert ein weiteres Request das Dokument. In relationalen Datenbanken, dürfte ein Schreibzugriff nicht ausgeführt werden, bevor er nicht ein Lock auf der Enitity einrichten kann. Lesezugriffe, die zeitgleich mit dem Schreibzugriff eintreten, müssen abwarten bis die Sperre aufgehoben wird (siehe links in der Abb 4). Dieses Abwarten spricht aber entweder gegen Availabilty oder gegen Partition Tolerance. Daher darf ein AP System keine Locks einrichten. Aus diesem Grund wird MVCC angewendet. Lesezugriffe während des Schreibzugriffs werden erlaubt. Damit aber die Konsistenz auf der lokalen Maschniene zwischen den einzelnen Unteroperation eines Lesezugriffs nicht gestört wird, wird immer die Version zurückgegeben, die ein Request bei seinem Start kannte (siehe rechts in der Abb 4 am Beispiel von CouchDB [Andersen (2010), Chapter 2].

Eventual Consistency heißt grob gesagt also, dass wenn keine neuen Updates auftreten, jeder Zugriff eventuell den aktuellen Wert liefert. In der Realität kann diese Aussage durch mehrere der Stärke nach unterschiedliche Klassen verfeinert werden. Siehe dazu [Vogels (2008), Chapter 9] für einen theoretischen Einstieg und [10gen (2010)] für einen mehr pragmatischeren Überblick am Beispeil der MongoDB Datenbank.

Die CP Klasse, welche auf Availability im Vorzug von Einhaltung der Konsistenz trotz Netzwerkfehler, verzichtet wird selten vertreten. MongoDB ist nach meiner Recherche die einzige Datenbank, welche diesen Ansatz verfolgt (alternativ kann diese Datenbank auch im AP Modus betrieben werden). Im Allgemeinen bedeutet CP, dass ein Master Server entscheidet, auf wie viele Replicas eines Shards Schreib und auf wie viele Lesezugriffe gemacht werden (siehe Abb 5).

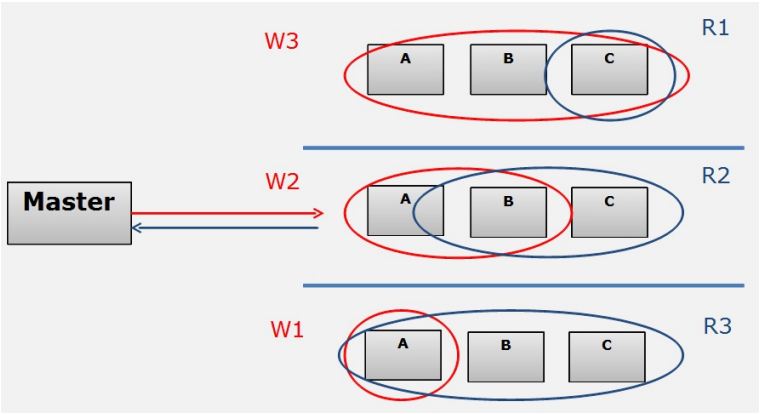

Abbildung 5: CP Klasse

Angenohmmen es gibt 3 Replicas und Updateoperationen(Writes) werden immer auf allen 3 Instanzen ausgeführt. Somit reicht das Lesen nur von einem Replica aus, um konsistente Daten zu erhalten, auch wenn zwischenzeitlich einzelne Replicas ausfallen. Es gilt also, dass Daten immer dann konsistent sind, wenn die Anzahl der Write-Instanzen summiert mit der Anzahl der Read-Instanzen größer ist als die Gesamtanzahl der Instanzen. Für die Availabilty beduetet es wiederum eine Entscheidungsmöglichkeit, wie stark das System für Lese- oder Schreiboperationen verfügbar sein kann, falls Netzwerkfehler auftreten. Diese Vorgehensweise wird in [Strauss10 (2010)] verdeutlicht.

3 Schluss

Die Arbeit spiegelte die aktuell stattfindende Auseinadersetzung für die Gerechtfertigung der Existenz von nicht relationalen Datenbanksystemen. Dabei wurden unterschiedliche Ansätze vorgestellt, welche sich bezüglich des CAP Theorems entweder in die AP oder CP Klasse platzieren. Abschließend sollen konkrete Szenarien gezeigt werden, an denen ein praktischer Einsatz von NoSQL erforscht werden kann.

3.1 Anwendungen

AP Klasse:
Viele moderne Webanwendungen gerade im Unterhaltungsbereich (z.B. kollektive Blogs) sind zwecks Performance Optimierung gezwungen ihre Anwendung und Daten auf mehrere Nodes zu scalieren. Dabei können diese auf eine starke Konsistenz der Daten verzichten. Viel wichtiger ist jedoch, dass eine ständige Verfügbarkeit garantiert wird. Es ist bespielsweise nicht wichtig, dass jeder Benutzer beim Besuch eines kollektiven Blogs immer die neuesten Beiträge sieht. Entscheidend ist, dass jeder Benutzer immer irgendeine Version der Seite zu sehen bekommt, damit etwa ständig Werbeeinnahmen generiert werden können. Bekannte Anwendungen hier sind Facebook, Twitter, Google Blogs. Beinahe jede Informations oder Unterhaltugnwebseite mit

dynamischen Inhalt, kann auf die ACID Garantien verzichten und mithilfe von NoSQL ohne Aufwand auf der Implementierungsebene fehlertolerant scaliert werden.

Eine zweite Einsatzmöglichkeit ergibt sich aus dem MVCC Feature dieser Systeme. Die Multiversionierung der Daten erlaubt Synchronisationsanwendungen zwischen mehreren Geräten wiederum ohne hohen Aufwand auf der Applikationsebene. Ein Ansatz findet sich in [Andersen (2010)]. Beschrieben wird ein Anwendungszenario bei dem Musik Playlists zwischen einem Mobilen- und einem Desktopgerät synchronisiert werden. Auf den beiden Geräten befindet sich eine Replica der gleichen Datenbank. Editierungsprozesse können aber unabhängig auf beiden Geräten stattfinden. Es muss also hohe Availabiltiy vorliegen. Dabei sind beide Geräte die meiste Zeit nicht miteinander in Verbindung. Eine Partition (Netzwerkunterbrechung) ist demnach beinahe ein Dauerzustand. Wird die Kommunikation zwischen den Geräten hergestellt, synchronisiert sich die Datenbank von selbst ohne das Updates auf gleichen Musikstücken verloren gehen. Denn die Datenbank ist Partition tolerant. Die Eintragungen liegen nun in den Unterschiedlichen Versionen vor. Nun liegt es an dem Entwickler entweder ein Standartverfahren oder eine Benutzerschnittstelle für die Auswahl der aktuellen Version zu implementieren.

CP Klasse: Wie erwähnt finden sich wenige Beispiele für CP Anwendugen und die meisten NoSQL Systeme verfolgen eher den AP Weg. Ein Szenario wäre aber bei verteilten nicht zu 100% replizierten Datenbanken denkbar. Man stelle sich vor, es besteht ein Filialennetz, wobei alle Filialen möglichst auf alle Daten zugreifen wollen. Dabei sorgt man, dass Updates auf lokalen Daten immer mindestens in der eigenen Filialen Datenbank landen. Fällt eine Filiale aus, so bleiben sowohl Reads als auch Writes auf eigene Daten möglich und das konsistent.

Ein Auflistung möglicher Einsatzszenarien für NoSQL durch Recherche in diversen Webressourcen nach deren Verwendung zeigt weiterhin [Hoff (2010)].

3.2 Ausblick

Es wurden Szenarien gezeigt, welche von den NoSQL Entwicklern für den Einsatz dieser Systeme oft diskutiert werden. Ein eher unerforschtes Gebiet für diese Systeme bildet die Echtzeitanalyse von für Geschäftsentscheidungen relevanten Daten. Diese Art von Einsatz wird auch mit Begriffen wie Buisness Intelligence oder Data Warehousing in Verbindung zusammengebracht.

So behauptet z.B. [Stonebraker und Hong (2009) [in einem Artikel, dass NoSQL Column Stores herkömmliche RDBMS auf diesem Gebiet bei Faktor 50 in Performance übertreffen können. Das von Google vorgestellte Map/Reduce Verfahren, welches in vielen NoSQL Systemen implementiert ist, erlaubt zusätzlich auf einfache Weise komplexe Abfragen zu parallelisieren und somit in Realtime auszuführen. Diese Technik macht sich der Online Shop Gilt Europe zu Nutze. Die Technologie wird von diesem Unternehmen auch einzeln unter dem Namen Hummingbird Vertrieben. Mit Hilfe von der NoSQL Datenbank MongoDB verspricht das Produkt Benutzerverhalten von Online Shop Usern in Echtzeit zu monitoren. Die Motivation besteht darin, sofort auf Benutzerverhalten im Web ähnlich wie im echten Leben zu reagieren, um etwa der Produkte die dem Kaufverhalten entsprechen verstärkt anzubieten.

Ähnliche Motivation strebt ein Hamburger Energie Unternehmen an. Es fanden bereits erste Gespräche mit der Hochschule für Angewandte Wissenschaften Hamburg , wobei ein Forschungsprojekt angestrebt wird, bei dem untersucht werden soll, wie mithilfe von Buisness Intelligence Software Techniken auf das Energiebrauchsverhalten in echtzeit reagiert werden kann um so effizient Stromressourcen verteilen zu können. Es drängt sich demnach die Fragestellung auf, ob dieses mithilfe von NoSQL erreicht werden kann und ob dabei Performance Vorteile im Vergleich zu einer konvenitionellen RDBMS Lösung festegestellt werden können.

3.3 Zusammenfasssung

RDBMS bieten starke Garantien für Datenkonsistenz und Verfügbarkeit durch das transaktionalle Konzept. Dieses Paradigma ist unschlagbar, wenn es sichere Finanzprozesse oder ERP Abrechnungsysteme geht. Jedoch wird die Lösung nach dem Motto - „One size fits All“ - auch in den meisten anderen Fällen angewendet. Dies weist sich nachteilig bei der Scalierung der Datenbank auf mehrere physikalische Nodes auf, da keine Toleranz gegenüber Netzwerkzusammenbrüchen besteht. NoSQL Systeme konzentrieren sich auf Gewährleistung dieser Toleranz. Dafür müssen aber wiederum einige der RDBMS Garantien aufgegeben werden. Je nach dem, welche Eigenschaften aufgegeben werden, werden unterschiedliche Systeme angeboten. Es gilt das Motto, dass für jedes Szenario, eine unterschiedliche Datenbank gewählt werden kann.

Literatur

[Strauss10 2010] : *MongoDB Day - Austin*. URL http://blip.tv/file/3605033, 2010

[10gen 2010] 10GEN: *Manual - MongoDB*. Webseite. 2010. – URL http://blog.mongodb.org/post/475279604/on-distributed-consistency-part-1. – Letzter Aufruf am 27. Feb 2011

[Abadi 2010] ABADI, Daniel: *The problems with ACID, and how to fix them without going NoSQL*. Webseite. 2010. – URL http://dbmsmusings.blogspot.com/2010/08/problems-with-acid-and-how-to-fix-them.html. – Letzter Aufruf am 27. Feb 2011

[Andersen 2010] ANDERSEN, Slater: *CouchDB: The Definitive Guide*. O'Reilly, 2010. – ISBN 978-0-596-15589-6

[Anthes 2010] ANTHES, Gary: Happy Birthday, RDBMS! In: *Commun. ACM* 53 (2010), May, S. 16–17. – URL http://doi.acm.org/10.1145/1735223.1735231. – ISSN 0001-0782

[Cartell 2011] CARTELL, Rick: *Scaleable SQL and NoSQL Datastores*. 2011. – URL http://www.cattell.net/datastores/Datastores.pdf. – Letzter Aufruf am 27. Feb 2011

[Dean und Ghemawat 2010] DEAN, Jeffrey ; GHEMAWAT, Sanjay: MapReduce: a flexible data processing tool. In: *Commun. ACM* 53 (2010), January, S. 72–77. – URL http://doi.acm.org/10.1145/1629175.1629198. – ISSN 0001-0782

[Gilbert und Lynch 2011] GILBERT, Seth ; LYNCH, Nancy: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. (2011). – URL http://www.mongodb.org/display/DOCS/Manual. – Letzter Aufruf am 27. Feb 2011

[Henderson 2006] HENDERSON, Cal: *Building Scalable Web Sites*. O'Reilly, 2006. – ISBN 0-596-10235-6

[Hoff 2010] HOFF, Todd: *What the heck are you actually using NoSQL for?* Webseite. 2010. – URL http://highscalability.com/blog/2010/12/6/what-the-heck-are-you-actually-using-nosql-for.html. – Letzter Aufruf am 27. Feb 2011

[Kaye 2004] KAYE, Doug: *Web-Services Transactions*. Webseite. 2004. – URL http://xml.sys-con.com/node/43755. – Letzter Aufruf am 27. Feb 2011

[Pritchett 2008] PRITCHETT, Dan: BASE: An Acid Alternative. In: *Queue* 6 (2008), May, S. 48–55. – URL http://doi.acm.org/10.1145/1394127.1394128. – ISSN 1542-7730

[Stonebraker und Hong 2009] STONEBRAKER, Michael ; HONG, Jason: Saying good-bye to DBMSs, designing effective interfaces. In: *Commun. ACM* 52 (2009), September, S. 12–13. – URL http://doi.acm.org/10.1145/1562164.1562169. – ISSN 0001-0782

[Vogels 2008] VOGELS, Werner: Eventually Consistent. In: *Queue* 6 (2008), October, S. 14–19. – URL http://doi.acm.org/10.1145/1466443.1466448. – ISSN 1542-7730

Abbildungsverzeichnis