NoSQL-Datenbanken

Markus Kramer

***Zusammenfassung*—NoSQL-Datenbanken sind zu einer interessanten Alternative zu herkömmlichen Datenbanken geworden. In dieser Arbeit werden die dahinter liegenden Konzepte erläutert, an zwei Bespielsystemen illustriert und es wird ein Vergleich zu den relationalen Datenbanken gezogen.**

I. Einleitung

Seit einigen Jahren gewinnt eine neue Klasse von Datenbanken stark an Verbreitung - die so genannten NoSQL-Datenbanken. Inzwischen wurden mehrere prominente Vertreter dieser Art von Datenbanken entwickelt. Dabei geht es nicht nur darum, dass kein SQL für Anfragen verwendet wird. Hinter dem Schlagwort NoSQL verbirgt sich eine Reihe von Technologien, womit sich diese Datenbanken grundsätzlich von den klassischen relationalen Datenbanken unterscheiden. Hierzu gehören oft eine Peer-to-Peer-Architektur, das Aufsetzen auf simplen Key-Value-Stores, die Speicherung unstrukturierter Daten, der Verzicht auf Transaktionen und die Idee von Eventual Consistency.

NoSQL-Datenbanken sind derzeit sehr in der Diskussion. Beispielsweise fand im Juni 2011 die Konferenz „Berlin Buzzwords 2011“ statt, auf der NoSQL ein Schwerpunkt war.

Diese Arbeit soll einen Überblick über das Thema der NoSQL-Datenbanken geben. Zuerst wird erläutert wieso es einen Bedarf an diesen neuen Datenbanken gibt. Anschließend werden die Kernkonzepte vorgestellt und an zwei Beispiel-Datenbanken verdeutlicht. Zum Schluss werden Stärken und Schwächen gegenübergestellt und das Potenzial des Art von Datenbank kritisch bewertet.

II. Grundlagen

Für ein Verständnis der Konzepte der NoSQL-Datenbanken ist es sinnvoll zuerst die klassischen relationalen Datenbanken zur charakterisieren, deren Probleme herauszuarbeiten und andere Grundlagen zu erläutern.

A. ACID

Relationale Datenbanken sind nach wie vor der mit Abstand am meisten verbreite Typus von Datenbanken. Ihren Erfolg verdanken sie einer sehr mächtigen Abfragesprache (SQL) und der Tatsache, dass sie dem Anwender viele praktische Funktionen und Garantien bieten. Hierzu zählen insbesondere die ACID-Eigenschaften. ACID bedeutet [1, S. 317-318]:

1) Atomarität (Atomicity): Jede Transaktion wird entweder komplett oder gar nicht ausgeführt.
2) Konsistenzerhaltung (Consistency): Jede Transaktion bewahrt die formale Korrektheit der Datenbank. Dies bedeutet, dass Daten die durch eine Transaktion erfolgreich geschrieben wurden, sofort für alle sichtbar sind. Die Datenbank wird nie veraltete Daten zurückgegeben.
3) Isolation (Isolation): Transaktionen sind von einander isoliert. Operationen können keine Änderungen aus einer anderen Transaktion sehen, die noch nicht abgeschlossen wurde.
4) Dauerhaftigkeit (Durablility): Die Garantie, dass die Änderungen einer abgeschlossenen Transaktionen dauerhaft erhalten bleiben.

Diese Garantien sind für den Anwender äußerst praktisch, denn sie müssen dann nicht mehr auf Applikationsebene sichergestellt werden. Damit erfüllen relationale Datenbanken eine wichtige Vorraussetzung von verlässlichen Systemen.

B. Skalierbarkeit

1) Definition: Unter Skalierbarkeit versteht man die Eigenschaft eines Systems, durch das Hinzufügen von Ressourcen einen entsprechenden Leistungszuwachs zu erzielen. Bei einem gut skalierbaren System bleibt das Verhältnis

zwischen hinzugefügten Ressourcen und erzieltem Leistungszuwachs über einen langen Zeitraum konstant.

Es wird außerdem zwischen horizontalem und vertikalem Skalieren unterschieden. Vertikales Skalieren (scale-up) bedeutet, dass derselbe Rechner durch mehr Hardware aufgerüstet wird (z.B. bessere CPU, mehr RAM oder mehr parallel arbeitende Festplatten). Vertikal zu skalieren ist einfach und auch relativ günstig, solange massenproduzierte Standardhardware eingesetzt wird.[2] Steigt der Leistungsbedarf jedoch weiter, wachsen die Kosten bald überproportional an. Daher hat die vertikale Skalierbarkeit Grenzen und eignet sich nicht für Anwendungsfälle mit sehr hoher oder stetig steigender Last.

Unter horizontaler Skalierbarkeit (scale-out), versteht man, dass weitere Rechner hinzugefügt werden. Das heißt, es handelt sich um ein verteiltes System aus Rechnern. Dabei wird oft günstige, massenproduzierte Standardhardware eingesetzt.[3, S. 277] Hier kann praktisch beliebig skaliert werden, ohne dass die Kosten überproportional steigen. Unter anderem setzen große Unternehmen wie Google und Facebook auf diese Art der Skalierbarkeit. Aber da es ein verteiltes System ist, steigt die Komplexität der Software.

2) Horizontale Skalierbarkeit relationaler Datenbanken: Eine Möglichkeit der horizontalen Skalierbarkeit relationaler Datenbanken stellt die Master/Slave-Replikation dar. Hierbei werden alle schreibenden Transaktionen auf dem Master-Datenbank-Server durchgeführt. Dieser Server repliziert die Änderungen auf eine Anzahl so genannter Slaves. Auf Slaves kann dementsprechend nur lesend zugegriffen werden. Viele relationale Datenbanken bieten dies an.[4, S. 696] Diese Architektur funktioniert gut für Anwendungsfälle, in denen es viele lesende Zugriffe und relativ wenige Schreibzugriffe gibt. Jedoch bleibt der Master-Server ein Flaschenhals. Auch die Anzahl der Slaves ist begrenzt, da sonst der Replikationsaufwand auf dem Master zu hoch wird.

Eine weitere Möglichkeit zur horizontalen Skalierung stellt Sharding dar. Hierbei werden die Tabellen partitioniert und in einem shared-nothing Cluster verteilt gespeichert.[5, S. 10] Dadurch kann eine gute Lastverteilung erreicht werden. In der Praxis ist dies jedoch oft schwierig. Je nachdem anhand welches Kriteriums die Partitionierung erfolgte können die Abfragen äußerst teuer werden. Beispielsweise kann es für einen Join dann nötig sein, eine erhebliche Menge an Daten über das Netzwerk zu übertragen.[4, S. 692, 698]

Relationale Datenbanken garantieren außerdem die Atomarität der Transaktionen, die Konsistenz der Datenbank, die Isolation einzelner Transaktionen und die Dauerhaftigkeit aller bestätigten Transaktionen. Die Garantie dieser Eigenschaften führt in der Datenbank jedoch zu hoher Komplexität und hohem Aufwand, insbesondere wenn die Datenbank partitioniert über das Netzwerk verteilt ist. Die Konsistenz wird dann über das Two-Phase-Commit-Protokoll erreicht, welches aber hohen Koordinierungsaufwands bedarf, die Verfügbarkeit mindert und nicht beliebig skaliert.[4, S. 1008ff]

Letztendlich kann gesagt werden, dass sowohl die vertikale, als auch die horizontale Skalierbarkeit von relationalen Datenbanken sehr problematisch ist.[6, S. 13]

C. CAP-Theorem

Bei der Betrachtung der verschiedenen Techniken zur horizontalen Skalierbarkeit, sowohl für relationale als auch für NoSQL-Datenbanken, ist es wichtig zu verstehen, dass keine “perfekt” skalierbare Datenbank möglich ist. Im Jahr 2000 stellte Eric Brewer das CAP-Theorem auf. Es sagt aus, dass es für eine verteilte Datenbank unmöglich ist, alle drei der folgenden Garantien zu bieten [7, S. 1-2]:

- Consistency (dt. Konsistenz): Die Datenbank befindet sich immer in einem konsistenten Zustand, d.h. Leseoperationen auf zwei beliebigen Knoten ergeben zum selben Zeitpunkt immer das gleiche Ergebnis.
- Availability (dt. Verfügbarkeit): Das System ist immer verfügbar. Lese- und Schreiboperationen können immer durchgeführt werden. Dies wird üblicherweise durch Replikation der Daten erreicht. Replikate schützen einerseits vor Hardwareausfällen und beschleunigen andererseits die Leseoperationen.

- Partition Tolerance (dt. Partitionstoleranz): Das System kann trotz Kommunikationsabbruch zwischen einzelnen Knoten weiterarbeiten, selbst wenn das Gesamtsystem in zwei Teile getrennt wird (z.B. Aufgrund fehlerhafter Netzwerk-Hardware).

Alle drei Eigenschaften sind erstrebenswert. Es wurde aber gezeigt [7, S. 12], dass eine verteilte Datenbank maximal zwei dieser Eigenschaften bieten kann.

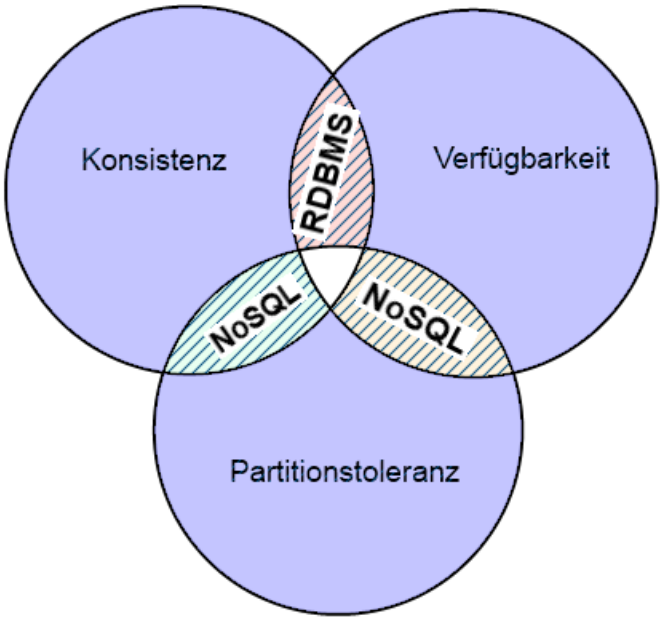

Abbildung 1. CAP Dreieck

Abbildung 1 veranschaulicht dies. Ein System, dass auf Partitionstoleranz verzichtet, kann Konsistenz und Verfügbarkeit erreichen. Dies ist bei den relationale Datenbanken der Fall, da sie ACID garantieren. Unter bestimmten Umständen, z.B. im Fall einer Trennung des Gesamtsystems in zwei Teile, kann das System vollständig ausfallen. Beispielsweise kann dann nicht mehr sichergestellt werden, dass Schreiboperationen einen konsistenten Zustand hinterlassen. Daher mindert die Garantie der ACID-Eigenschaften die Ausfallsicherheit von relationalen Datenbanken. Liegt der Fokus auf Konsistenz und Partitionstoleranz, kann nicht mehr sichergestellt werden, dass das System jederzeit verfügbar ist [8].

III. NoSQL-Datenbanken

Um die Probleme der relationalen Datenbanken zu lösen, gehen NoSQL-Datenbanken einen anderen Weg. Hier stehen Skalierbarkeit, Ausfallsicherheit und Verfügbarkeit im Fokus. Auf eine mächtige Abfragesprache und komfortable Funktionen wie Transaktionen und strenge Konsistenz wird verzichtet. Hierdurch können enorm große und flexible Datenbanken realisiert werden. Unter anderen setzen Amazon, Google und Facebook auf diese Technologie.[6, S. 12]

A. Definition

Datenbanken die andere Konzepte als die der relationalen Systeme verfolgen, gibt es schon seit Langem. Schon 1979 entwickelte Ken Thompson eine Key-Value-Datenbank. Der Begriff NoSQL wurde im Mai 2009 durch Eric Evans geprägt und steht inzwischen für „Not only SQL“.

Hinter dem Schlagwort NoSQL verbirgt sich eine Reihe von Konzepten, die von den NoSQL-Vertretern in unterschiedlichem Maße eingesetzt werden. Da es kein federführendes Gremium gibt, fällt eine Definition schwer. Folgende Definition ist stark an die Definition angelehnt, die Stefan Edlich in seinem Buch [3] vertritt:

1) Das Datenmodell ist nicht relational.
2) Fokus auf horizontaler Skalierbarkeit.
3) Im Wesentlichen schemafrei.
4) Unterstützung von Datenreplikation.
5) Das System bietet eine einfache API.
6) Nutzung eines weniger strengen Konsistenzmodells. Verzicht auf ACID.

B. Kernkonzepte

Inzwischen existiert eine Vielzahl von Datenbanken, die als NoSQL bezeichnet werden. Hierbei wird zwischen Column-Stores (z.B. Cassandra), Document-Stores (z.B. CouchDB, MongoDB), Key-Value-Stores (z.B. Redis) und Graphdatenbanken (z.B. neo4j) unterschieden. [3, S. 6] Im Folgenden werden Kernkonzepte vorgestellt, die NoSQL-Datenbank von relationalen Datenbanken unterscheiden. Aufgrund der Vielfalt an NoSQL-Datenbanken werden diese Konzepte jedoch nicht von jeder Datenbank im selben Maße eingesetzt.

1) Keep It Simple: Als erstes kann man feststellen, dass auf bestimmte Funktionen, die typisch für relationale Datenbanken sind, bei NoSQL-Datenbank bewusst verzichtet wurde.

Hierzu gehören insbesondere Transaktionen. Die meisten NoSQL-Systeme kennen das Konzept einer Transaktion nicht. Der Gedanke hinter dieser Entscheidung ist, dass Transaktionen maßgeblich für die schlechte horizontale Skalierbarkeit von relationalen Datenbanken verantwortlich sind. Es wird angenommen, dass Transaktionen, insbesondere bei Web-Applikationen, meistens nicht benötigt werden. Entweder, weil sowieso nicht mehrere zusammenhängende schreibende Operationen innerhalb einer Benutzeranfrage ausgelöst werden, oder weil die Applikation eventuelle Probleme bei der Datenintegrität verkraften kann, da sie robust genug entwickelt wurde.[2]

Außerdem wird auf Joins verzichtet. Ein Join lässt sich sehr schlecht auf einer großen verteilten Datenbank durchführen. Stattdessen sollen Anwendungen so entwickelt werden, dass sie ohne diese auskommen bzw. es soll ein Datenmodell verwendet werden welches Joins unnötig macht.

2) Schemalos: Im Gegensatz zu relationalen Datenbanken ist es nicht notwendig ein festes Schema zu konfigurieren, dessen Einhaltung dann von der Datenbank erzwungen wird. Dies führt zu einer höheren Flexibilität. Somit ist es möglich, im laufenden Betrieb Erweiterungen oder Änderungen am Datenmodell durchzuführen. Relationale Datenbanken müssen für diesen Zweck heruntergefahren werden. Ein kompletter Neustart des Datenbank-Clusters und die damit verbundene Downtime ist für Firmen die hohe Verfügbarkeit garantieren wollen (z.B. Amazon) ein erhebliches Problem. [3, S. 3]

3) Eventual Consistency: Eines der wesentlichen Markenzeichen von NoSQL-Systemen, welches auch in der obigen Definition enthalten ist, ist die Abkehr von ACID und der Forderung nach strenger Konsistenz. Wie im letzten Kapitel erläutert, verursachen diese Forderungen erhebliche Skalierbarkeitsprobleme. Bei NoSQL-Datenbanken wird im CAP-Dreieck meistens der Schwerpunkt von Konsistenz auf Verfügbarkeit und Partitionstoleranz verlagert.

Als wichtiges Schlagwort ist hier die Idee von Eventual Consistency zu nennen. Unter strenger Konsistenz (engl.: strong consistency) versteht man, dass ein Datum, welches durch eine schreibende Operation verändert wurde, unmittelbar danach von allen lesenden Operationen korrekt zurückgeben wird. Das Ergebnis von Lese- und Schreiboperationen ist also immer konsistent und es werden nie veraltete Daten zurückgegeben. [3, S. 34]. Offensichtlich verursacht dies im Hintergrund einen erheblichen Aufwand. Wie bereits vorher erwähnt, wird hierfür das aufwändige Two-Phase-Commit-Protokoll eingesetzt.

Bei NoSQL-Datenbanken weicht man die Forderung nach strenger Konsistenz auf. Man spricht nun von Eventual Consistency. D.h. es wird lediglich garantiert, dass irgendwann alle Änderungen propagiert wurden und damit die Datenbank schließlich wieder konsistent ist. Wie lange es dauert die Konsistenz wiederherzustellen ist natürlich System und insbesondere lastabhängig. Meistens aber nach menschlichen Maßstäben immer noch fast sofort. Dies ist insbesondere für viele Web-2.0-Applikationen wie Facebook, Twitter oder Amazon vollkommen ausreichend. [8] Hier liegen zwischen den einzelnen Benutzeraktionen, die wiederum Datenbankoperationen anstoßen, ohnehin meist mehrere Sekunden. Hat der Benutzer beispielsweise einen Artikel in seinen Warenkorb genommen, bleiben dem System einige Sekunden die Änderung vollständig zu propagieren, bevor die nächste Anfrage des Benutzers zu erwarten ist.

4) Peer-to-Peer-Architektur: Aus den Forderungen nach hoher horizontaler Skalierbarkeit und Verfügbarkeit folgt auch der Schluss, dass es keinen Single-Point-of-Failure mehr geben darf. Einige NoSQL-Vertreter setzen daher auf eine Peer-to-Peer-Architektur. Dies bedeutet, dass jeder Knoten der Datenbank gleichberechtigt ist. Das System besitzt einen Mechanismus zur Datenreplikation zwischen den einzelnen Knoten. Hierdurch kann die Last gut auf alle Knoten verteilt werden, was die Verfügbarkeit und Ausfallsicherheit erhöht.

Eine solche Verteilung der Daten in einem Peer-to-Peer-System ist deshalb möglich, weil meist ein besonders einfaches Datenmodell verwendet wird. Das Datenmodel vieler NoSQL-Vertreter ist im Wesentlichen ein Key-Value-Store. Da Joins und andere komplexe Operationen aus der SQL-Welt nicht unterstützt werden, können die Daten relativ einfach in einem Peer-to-Peer-System verteilt werden.

Bewährte Verfahren aus dem Bereich der Peer-to-Peer-Systeme können verwendet werden

um diese Systeme umzusetzen. Beispielsweise das strukturierte Peer-to-Peer-System Chord und Verfahren wie das Consistent Hashing. [3, S. 36-36] Hierdurch wird ermöglicht, dass beim Hinzufügen oder beim Wegfall eines Knotens nicht alle Objekte neu verteilt werden müssen, sondern jeweils nur ein kleiner Teil umkopiert wird, um das System optimal auszulasten. Am Beispiel von Cassandra werden die Vorteile einer Peer-to-Peer-Architektur später noch verdeutlicht.

C. Beispiel: CouchDB

CouchDB ist ein Vertreter der dokumentorientierten Datenbanken. Es wird derzeit als Top-Level-Projekt bei der Apache Software Foundation weiterentwickelt.

1) Datenmodel: Das Datenmodell ist schemafrei. Ein Dokument kann beliebige Daten enthalten und beliebig komplex strukturiert sein. Gespeichert werden Dokumente im JSON-Format. Eine CouchDB-Datenbank ist eine Sammlung solcher JSON-Dokumente, die über eine ID referenziert werden. Die Idee hinter einer dokumentorientierten Datenbank ist, dass die Domain-Objekte nicht wie bei einem relationalen Datenmodel in eine Vielzahl an Relationen zerstückelt werden, sondern zusammengehörige Informationen auch zusammen in einem Dokument ablegt werden. Während man bei relationalen Datenbanken massiv aufwendige Joins einsetzen muss um diese Daten wieder in Verbindung zu bringen, stehen sie in CouchDB im selben Dokument.

Ein Beispiel für ein Dokument in dem ein komplettes Benutzerprofil gespeichert ist:

```
{
   "_id": "1000371",
   "_rev": "1-e539134ee01ec1d4127ce2...",
   "username": "MaxMuster",
   "pictures": [
       {
           "filename": "001.jpg",
           "comment": "im urlaub"
       },
       {
           "filename": "002.jpg",
       }
   ]
}
```

Für gewisse Anwendungsfälle kann dies von großem Vorteil sein. Ein gutes Bespiel ist ein Content-Management-System (CMS). Die Inhalte (Dokumente) die in einem CMS verwaltet werden, bestehen oft aus zahlreichen Objekten, beispielsweise einem Wurzelobjekt, verschiedenen Versionen, Seiten, Kapitel, Absätze und Multimedia-Inhalte, die aber alle zu demselben Dokument gehören. In einer relationalen Datenbank müsste eine große Menge an Joins durchgeführt werden, um ein Dokument des CMS darzustellen. In einer dokument-orientierten Datenbank können all diese Informationen in einem einzelnen JSON-Dokument abspeichert werden. Die InfoPark AG, ein Hersteller von Content-Management-Systemen, ist daher von ihrer früheren relationalen Lösung auf CouchDB umgestiegen (bzw. BigCouch, welches eine kommerzielle Variante ist). [9, S. 27]

Besonders interessant ist die Art, wie Abfragen in CouchDB umgesetzt werden, da sich das Konzept grundlegend von SQL unterscheidet. CouchDB setzt auf Map/Reduce um die gewünschten Informationen zu ermitteln. Hierzu wird eine View definiert, die die beiden Map/Reduce-Funktionen als JavaScript enthält. Map/Reduce wird im Rahmen dieser Arbeit nicht weiter erläutert. Vereinfacht ausgedrückt erlaubt es, im ersten Schritt durch die Map-Funktion auf den Dokumenten eine Selektion durchzuführen und, falls gewünscht, die Elemente im Anschluss durch die Reduce-Funktion zu aggregieren.[3, S. 111]

Gegeben sei wie zuvor eine Datenbank mit Benutzerprofilen. Soll ein Benutzer mit einem bestimmten Benutzernamen gefunden werden, ist folgende View sinnvoll:

```
Map Function: function(doc){
  emit(doc.username.toLowerCase(), doc._id);
}
```

Diese Funktion ist der Map-Teil der View. Eine Reduce-Funktion ist nicht notwendig. Diese Funktion erzeugt eine View, in der der Benutzername auf die ID des Benutzer-Dokuments referenziert. Für einen gegebenen Benutzernamen lässt sich nun sehr schnell das dazugehörige Benutzer-Dokument ermittelt. Für die Map- und Reduce-Funktionen steht die komplette Mächtigkeit von Javascript zur Verfügung. Hier wird beispielsweise

die toLowerCase()-Methode verwendet, um Groß- und Kleinschreibung zu ignorieren. Mit if und for Konstrukten können sehr komplizierte Abfragen umgesetzt werden. Je nach Anwendungsfall kann dies einfacher zu verstehen sein, als eine komplexe SQL-Abfrage. Die View wird von CouchDB automatisch aktuell gehalten.[10]

2) Architektur: CouchDB kann einfach auf mehrere Knoten verteilt werden. Ein Replikationsmechanismus ist elementarer Bestandteil von CouchDB. Die Replikation von Daten kann zwischen jedem beliebigen Knoten durchgeführt werden. Es gibt keinen "Master"-Knoten dem besondere Bedeutung zukommt, wie bei der klassischen Master-Slave-Replikation. Man kann hier also auch von einem Peer-to-Peer-System sprechen, obwohl die Daten nicht wie bei einem Chord verteilt werden.[3, S. 112]

Wie die Replikation konkret stattfindet, ist dem Anwender überlassen. Daten könnten zum Bespiel nur in eine Richtung von Knoten A zu Knoten B repliziert werden (unidirektional) oder auch in die andere Richtung (bidirektional). Dies muss jedoch vom Anwender jeweils selbst für alle Knoten des Clusters eingerichtet werden. Da dies sehr bald zu einer hohen Komplexität führt, wird in der Praxis oft eine einfache Master-Slave-Replikation eingesetzt.

Da Dokumente zwischen zwei beliebigen Knoten repliziert werden können, ist ein Konfliktmanagement nötig. CouchDB erkennt Konflikte und wählt deterministisch eine gewinnende Version aus. Die verlierende Kopie wird als eine alte Version gespeichert. Der Anwender kann CouchDB auf aufgetretene Konflikte hin abfragen und dann die automatische Entscheidung der Datenbank gegebenenfalls korrigieren. Hierbei sieht man, dass auch die ACID-Eigenschaft "Dauerhaftigkeit" nicht mehr komplett garantiert wird, da eine von der Datenbank bestätigte Änderung eventuell zu einem späteren Zeitpunkt durch die Konfliktbehebung überschrieben werden kann.

Dieser Replikationsmechnismus ist die Grundlage zur horizontalen Skalierbarkeit von CouchDB. Funktionen wie Partitionierung und Sharding werden jedoch noch nicht standardmäßig unterstützt [3, S. 113]. Außerdem steigt der Administrationsaufwand mit der Größe des Clusters stark. Es ist aber zu erwarten, dass sich dies durch neue Tools in Zukunft verbessert, da CouchDB auf einer soliden Architektur zur horizontalen Skalierbarkeit aufsetzt.

D. Beispiel: Cassandra

Cassandra ist eine NoSQL-Datenbank, die ursprünglich von Facebook im Jahr 2007 entwickelt wurde. Dort wurde sie für die Nachrichtensuche in den Posteingängen der Facebook-Nutzer verwendet. Die Datenbank ist inzwischen Open-Source und seit 2010 ein Top-Level-Projekt der Apache Software Foundation. Cassandra ist eine spaltenorientierte Datenbank und wurde stark von Amazons Dynamo beeinflusst. [11, S. 24]

1) Datenmodel: Das spaltenorientierte Datenmodel von Cassandra ist etwas schwieriger zu verstehen. Auf der untersten Ebene kann man Cassandra als einfachen Key-Value-Store sehen. Allerdings können diese verschachtelt eingesetzt werden. Es werden ein oder mehrere Column-Families definiert. Diese enthalten Zeilen. Eine Zeile wird durch einen Row Key referenziert (vergleichbar mit dem Primärschlüssel einer Zeile in einem RDBMS). Der Inhalt einer Zeile wird durch Columns abgebildet, die wiederum ein Key-Value-Paar sind. [11, S. 43]

Folgendes Beispiel verdeutlicht, wie man die bekannte Twitter-Applikation mit Cassandra umsetzen könnte.

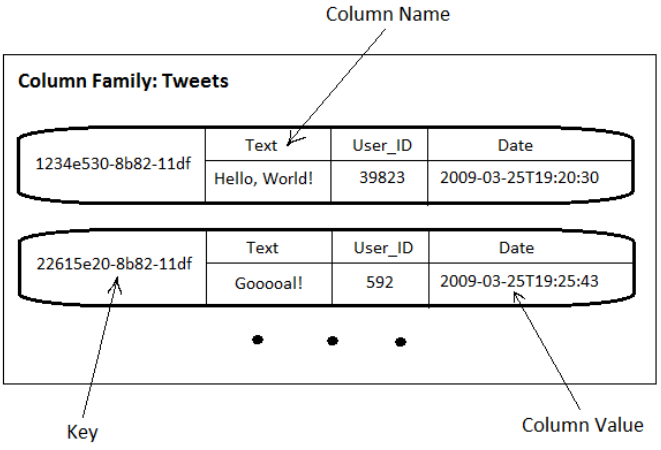

Abbildung 2. Column Family: Tweets. Quelle: http://maxgrinev.com/2010/07/09/a-quick-introduction-to-the-cassandra-data-model

Wie Abbildung 2 zeigt, werden in der ColumnFamily "Tweets" die Kurznachrichten gespeichert. Jede Zeile hat einen Key. Die

Columns einer Zeile speichern die zugehörigen Informationen wie Text und User_ID. Die ID eines Tweets ist kein künstlicher Schlüssel, sondern enthält eine Semantik. Hier enthält der Key die Uhrzeit des Tweets. Cassandra bietet selbst keine Abfragesprache wie SQL an. Der Zugriff auf Daten kann lediglich über die entsprechenden Keys erfolgen. Sollen alle Tweets eines Benutzers angezeigt werden, muss hierfür eine extra ColumnFamily (Abbildung 3) angelegt werden.

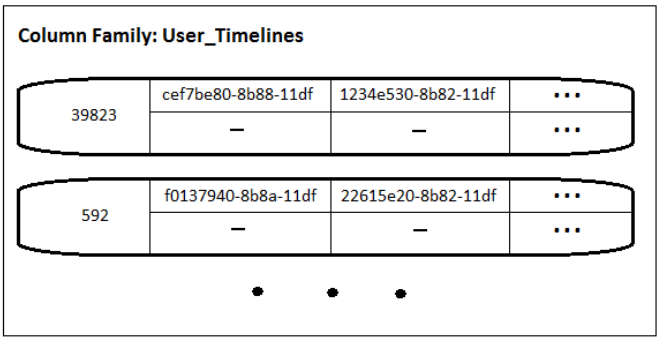

Abbildung 3. Column Family: User_Timelines

Im Prinzip ist dies ein Sekundärindex. Somit können alle Tweets eines Benutzers gefunden werden. Hervorzuheben ist, dass der Column Name direkt die ID des entsprechenden Tweet enthält und der Wert der Column selbst leer ist. Zeilen können also unterschiedliche Columns haben und die Column Names selbst Informationen speichern, im Gegensatz zu Tabellen in relationalen Datenbanken. Cassandra ist ein sortierter Key-Value-Store. Wie zuvor erwähnt, wurde in der ID eines Tweets ein Zeitstempel gespeichert. Somit enthalten die Column Names diese Zeitstempel und werden von Cassandra entsprechend sortiert gespeichert. Die Tweets eines Benutzers können hierdurch in der korrekten zeitlichen Reihenfolge abgerufen werden. Die Tatsachen, dass Column Names selbst Informationen speichern können und eine Sortierreihenfolge gewahrt wird, sind wesentliche Unterscheidungsmerkmale zwischen dem Datenmodell von Cassandra und dem relationalen Datenmodell.

Für eine zusätzliche Kapselung stehen Super-Columns zur Verfügung. Diese werden hier aber nicht weiter betrachtet.

2) Architektur: Cassandra hebt sich außerdem von anderen Datenbanken aufgrund seiner Peer-to-Peer-Architektur ab. In einem Cassandra-Cluster gibt es keinen Master-Knoten und somit auch keinen Single-Point-of-Failure. Alle Key-Value-Paare werden über den Cluster verteilt gespeichert.

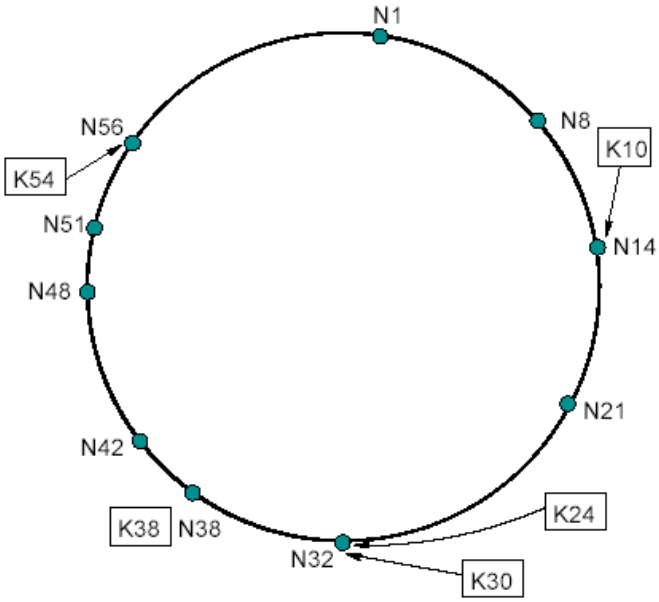

Abbildung 4. Cassandra Chord

Cassandra partitioniert Daten durch Consistent Hashing auf einem Chord (Hash Ring). Siehe Abbildung 4. Jeder Knoten (in der Abbildung mit „N“ für „node“ gekennzeichnet) ist verantwortlich für die Schlüssel (bezeichnet mit „K“ für „key“) zwischen ihm und seinem Vorgänger. Er ist somit der Koordinator für diesen Teil des Ringes.

Das Hinzufügen oder Wegnehmen (z.B. durch Hardwareausfall) eines Knotens hat nur Auswirkungen auf die direkten Nachbarn. Die Notwendigkeit Daten neu zu verteilen wird dadurch minimiert. Abhängig von den eigenen Anforderungen kann konfiguriert werden, wie viele Replikate erzeugt werden. Der Koordinator stellt sicher, dass genug Replikate für die in seinem Verantwortungsbereich liegenden Daten existieren.[12, S. 2,3]

In der Praxis können neue Knoten sehr einfach in einen bestehenden Cluster aufgenommen werden. Cassandra benötigt lediglich die Adresse eines anderen Knotens des Clusters und integriert den neuen Knoten dann komplett selbständig.

Cassandra sorgt auch dafür, dass der neue Knoten Daten enthält, durch die er andere besonders belastete Knoten entlastet.[12, S. 2] Daher ist der Administrationsaufwand eines Cassandra-Clusters viel niedriger als bei anderen Lösungen wie beispielsweise CouchDB.

Cassandra setzt sehr stark auf Verfügbarkeit und Partitionstoleranz, was durch Eventual Consistency erkauft wird. In einen Cassandra-Cluster kann immer geschrieben werden, auch wenn die zuständigen Knoten gerade nicht erreichbar sind (das so genannte Hinted Handoff Feature).[11, S. 93] Für Lese- und Schreiboperationen kann unabhängig die gewünschte Konsistenzgarantie gewählt werden. Bei einer schreibenden Operation bedeutet dies, dass festgelegt werden kann, auf wie viele Knoten das Datum mindestens erfolgreich geschrieben werden muss. Somit könnte sogar strenge Konsistenz erreicht werden, was jedoch kaum sinnvoll wäre, da dies dem Sinn der Peer-to-Peer-Architektur widerspricht.

Zusammenfassend kann gesagt werden, dass Cassandra eine sehr mächtige Datenbank ist, obwohl es im Prinzip nur auf einem Key-Value-Store aufbaut. Die Datenbank bietet viele Stellschrauben, mithilfe derer man sie an die eigenen Bedürfnisse anpassen kann. Das vom Anwender geforderte Verständnis der dahinter liegenden technischen Konzepte ist jedoch sehr hoch.

IV. Das Potential von NoSQL

A. Stärken

Die große Stärke von NoSQL-Datenbanken ist sicherlich die gute horizontale Skalierbarkeit. Hierdurch kann eine deutlich höhere Last bewältigt werden als mit relationalen Datenbanken. Damit einher geht eine höhere Verfügbarkeit und Ausfallsicherheit, denn NoSQL-Datenbanken verzichten oft auf strenge Konsistenz.

Aufgrund der Partitionstoleranz eignen sich NoSQL-Datenbanken besser für den Einsatz in der Cloud, wie Amazon EC2. Da man eine Cloud-Infrastuktur nicht selbst unter Kontrolle hat, muss man auf eine Vielzahl von möglichen Fehlern und Ausfällen vorbereitet sein. Aufgrund ihrer Robustheit eignen sich NoSQL-Datenbanken besser für den Einsatz in der Cloud. [9, S. 26]

Außerdem sind NoSQL-Datenbanken oft auf spezielle Anwendungsfälle optimiert, beispielsweise ein Key-Value-Store oder eine Graphdatenbank. Für diese Fälle sind sie natürlich besser geeignet als die universell einsetzbaren relationalen Datenbanken.

B. Schwächen

NoSQL-Datenbanken erreichen ihre Stärken im Wesentlichen dadurch, dass auf Transaktionen und ACID verzichtet wird. Die Annahme ist, dass viele Web-Applikationen ohne diese Funktionen auskommen.[5, S. 11] Je nach Anwendungsfall ist dies aber eventuell nicht der Fall. Ein gutes Beispiel sind Bankapplikationen die eine hohe Verlässlichkeit von der Datenbank erwarten. Soll bei einer Überweisung Geld von einem Konto abgezogen und auf ein anderes addiert werden, müssen beide Operationen entweder zusammen ausgeführt werden oder keine von beiden. Da NoSQL-Datenbanken keine Transaktionen unterstützen, eignen sie sich hierfür eher nicht. Gegebenenfalls müsste auf Applikationsebene eine Transaktionslogik aufwändig nachgebaut werden. Erschwerend kommt hinzu, dass man bei einem neuen Projekt vielleicht nicht am Anfang sicher abschätzen kann, ob man ohne Transaktionen auskommen wird.

Auch die fehlende Möglichkeit Joins durchzuführen kann problematisch sein. Zwar ist hier die Idee, das Datenmodell so zu wählen, dass auf Joins verzichtet werden kann, dies ist aber unter Umständen nicht immer möglich.

C. Potential

NoSQL kann derzeit sicherlich als Hype beschrieben werden. Allerdings sind Anforderungen an Datenbanken so vielfältig, dass man diese kaum durch eine „One Size Fits All“-Lösung abdecken kann, wie es durch relationale Datenbanken in der Vergangenheit versucht wurde. Bei besonderen Anforderungen, insbesondere an die Skalierbarkeit, stellen NoSQL-Datenbanken sinnvolle Alternativen dar. Allerdings sollte man sicherstellen, dass man wirklich derartige Skalierbarkeitsanforderungen hat. Die meisten Unternehmen haben nicht derart extreme Anforderungen wie Google oder Facebook

und die Skalierbarkeitsmöglichkeiten einer relationalen Datenbank sind oft ausreichend.[2] Auch sollte man sich darüber bewusst sein, dass man durch NoSQL andere Eigenschaften aufgibt, besonders Verlässlichkeit.[13]

Nicht zuletzt sollte man auch den wirtschaftlichern Aufwand berücksichtigen, den die Einführung eines NoSQL-Systems mit sich bringt. Die eingesetzten Konzepte unterscheiden sich zum Teil deutlich von denen der relationalen Datenbanken und sind nicht trivial. Diese müssen von den Entwicklern erst gelernt werden, was Zeit und damit Geld kostet. Andererseits ist meistens schon ein großes Know-How für den Einsatz relationaler Datenbanken vorhanden, welches dann ungenützt bleibt. Ferner gibt es nur wenig kommerziellen Support für NoSQL-Datenbanken, was bei einem produktiven Einsatz ein Problem sein kann.[6, S. 14]

Außerdem lässt sich beobachten, dass Unternehmen, die hohe Anforderungen haben und als NoSQL-Vertreter gelten, den klassischen Konzepten treu bleiben. Facebook setzt weiterhin auf ihre bestehenden MySQL-Systeme. Des Weiteren ist Facebook bei der Neuimplementiereung ihres Nachrichtensystems von Cassandra, welches sie ursprünglich selbst entwickelt hatten, auf Hbase umgestiegen. Dieses setzt im Gegensatz zu Cassandra nicht auf Eventual Consistency. Damit bleibt Facebook im Prinzip bei den klassischen Datenbank-Eckpfeilern Konsistenz und Verfügbarkeit und setzt nicht mehr auf das in der NoSQL-Community beworbene Konzept von Eventual Consistency und Partitionstoleranz.

V. Fazit

NoSQL-Datenbanken sind kein theoretisches Konzept, sondern haben sich bereits in der Praxis bewährt. Für gewisse Anwendungsfelder haben sie erhebliche Vorteile und sind daher eine gute Wahl. NoSQL-Datenbanken besetzen bisher aber eine Nische und haben auch deutliche Schwächen. Dass sie relationale Datenbanken als dominanten Typ ablösen werden, ist zu diesem Zeitpunkt nicht zu erkennen.

Literatur

[1] Matthias Schubert, *Datenbanken: Theorie, Entwurf und Programmierung relationaler Datenbanken (2. Auflage)*, Vieweg+Teubner, 2007.

[2] Dennis Forbes, "Getting real about nosql and the sql-isn't-scalable lie", http://www.yafla.com/dforbes/Getting_Real_about_NoSQL_and_the_SQL_Isnt_Scalable_Lie, March 2010.

[3] Stefan Edlich, Achim Friedland, Jens Hampe, and Benjamin Brauer, *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*, Hanser Fachbuchverlag, October 2010.

[4] Michael Kifer, Arthur Bernstein, and Philip M. Lewis, *Database Systems: An Application Oriented Approach, Complete Version (2nd Edition)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[5] Michael Stonebraker, "Sql databases v. nosql databases", *Commun. ACM*, vol. 53, pp. 10–11, April 2010.

[6] Neal Leavitt, "Will nosql databases live up to their promise?", *Computer*, vol. 43, pp. 12–14, 2010.

[7] Seth Gilbert and Nancy Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services", *SIGACT News*, vol. 33, pp. 51–59, June 2002.

[8] Werner Vogels, "Eventually consistent - revisited", http://www.allthingsdistributed.com/2008/12/eventually_consistent.html, December 2008.

[9] Thomas Witt, "Lessons learned from converting a web application to multitenancy and deploying it to amazon web services using scalarium", AWS Tech Summit for Developers and Architects, Berlin, 16.05.2011, 2011.

[10] Reuven M. Lerner, "At the forge: Couchdb views", *Linux J.*, vol. 2010, August 2010.

[11] Eben Hewitt, *Cassandra: The Definitive Guide*, O'Reilly, 2011.

[12] Avinash Lakshman and Prashant Malik, "Cassandra: a decentralized structured storage system", *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, April 2010.

[13] Reuven M. Lerner, "At the forge: Nosql? i'd prefer somesql", *Linux J.*, vol. 2010, April 2010.