Seminararbeit Cloud Data Management

Key Value Stores
Dynamo und Cassandra

Simon Bin

Januar 2010

Inhaltsverzeichnis

1 Einführung **4**

2 Amazon Dynamo **6**
2.1 Designziele 6
2.2 Technologie 8
2.3 Einschränkungen 9
2.4 Interface 10
2.5 Partitionierung und Replikation 10
2.6 Versionierung 11
2.7 Ausfälle 13

3 Apache Cassandra **14**
3.1 Designziele 14
3.2 Speicherformat 15
3.3 Interface 15
3.4 Architektur 17
3.5 Datenspeicher 17

4 Zusammenfassung **19**

Literaturverzeichnis **20**

1 Einführung

Relationale Datenbanksysteme zusammen mit der Anfragesprache SQL sind bisher State of the Art zur Speicherung und Analyse von Daten. Die vielen Verknüpfungsmöglichkeiten, z.B. durch Verbund, Gruppierung, Vereinigung und diverse Berechnungsmöglichkeiten erlauben eine vielseitige Verarbeitung und Analyse der Daten. Relationale Datenbanken werden durch eine strenge Normalisierung der Schemen begünstigt. Dies erlaubte Datensätze frei von Änderungsanomalien und die Datenbanksysteme garantieren alle die ACID Eigenschaften: Atomarität der Zugriffe, Konsistenz der Daten unter allen Umständen, Isolation gleichzeitiger Verbindungen und Dauerhaftigkeit der Operationen durch Logmechanismen etc.

Konsistenz unter hohen Schreibraten ist z.B. bei Bankgeschäften und -transaktionen unverzichtbar. Nicht überall gelten jedoch die gleichen Prioritäten. Key Value Stores versuchen effizienter für alltägliche (Web-)Anwendungen wie Warenkorb- oder Lesezeichensysteme zu sein, die keine beliebig komplexen Querys benötigen und schemalose oder nur leicht schematisierte Daten speichern. Key Value Stores fokussieren auf Skalierbarkeit, hoher Verfügbarkeit, Partitionierung/Replikation und extrem schnellen Antwortzeiten. Unter nur einem Primärschlüssel können oft beliebige binäre Datentypen als Werte abgelegt werden.

Neben Key Value Stores gibt es auch noch andere Alternativen zur relationalen Datenbank. Ein Überbegriff für nicht-relationale Datenbanken ist „NoSQL". Dazu gehören Objektdatenbanken, objekt-relationale Datenbanken oder Graph-Datenbanken wie z.B. db4o, GemStone, Statice, InfoGrid oder Virtuoso. Der Gedanke dahinter ist keinesfalls eine Konkurrenz zu SQL, sondern das Aufzeigen von Alternativen für die Fälle, wo relationale Datenbanken nicht die beste Wahl sind. [See09]

Für eine ausführlichere Einführung in das Thema sei außerdem auf [Bre10] verwiesen, der Allgemeines zum Thema Key Value Stores etwas ausführlicher behandelt und die Key Value Stores CouchDB, BigTable und Hadoop (HBase) vorstellt. Neben diesen Systemen gibt es noch eine ganze Reihe weiterer Key Value Stores: Tokyo Cabinet, Redis, Memcached, MongoDB oder Voldemort (Aufzählung nicht vollständig).

Im Rahmen dieser dieser Arbeit werden die beiden Key Value Stores Amazon Dynamo und Apache Cassandra (ehemals Facebook) vorgestellt. Zunächst können anhand Dynamos einige technologische Aspekte verdeutlicht werden welche sich danach teilweise in Cassandra wiederfinden. Abschließend wird ein kurzes Fazit über die fünf Key Value Stores CouchDB, BigTable, Hadoop, Dynamo und Cassandra unter Einbeziehung der Ergebnisse in [ebd.] erstellt.

2 Amazon Dynamo

Dynamo ist ein Key Value Store, welcher innerhalb von Amazon eingesetzt wird. Es handelt sich dabei weder um frei verfügbare Software noch um dem Endbenutzer zugängliche Software. Amazon hat jedoch die technologischen Aspekte der Anwendung in [DHJ+07] dokumentiert und frei zugänglich gemacht.

Das Einsatzgebiet von Dynamo besteht in einem Speichersystem für einige der Kernservices, wo Dynamo hohe Verfügbarkeit auch unter verschiedenen Ausfallszenarien bietet, um dem Benutzer der Amazon-Services ein uneingeschränktes Benutzererlebnis zu garantieren [DHJ+07, 205]. Jede „Nichtverfügbarkeit" (Ausfallszenario) ist für Amazon mit potenziellen finanziellen Einbußen verbunden. Dynamo ist daher insbesondere für die Auslieferung von kleineren Datenmengen pro Schlüssel optimiert und dient dem Zustandserhalt der einzelnen Anwendungen.

2.1 Designziele

Innerhalb Amazons herrschen strikte Anforderungen an alle Anwendungen in Bezug auf die Performance, Zuverlässigkeit und Effizienz. Das ständige Wachstum Amazons erfordert weiterhin eine hohe Skalierbarkeit aller dort eingesetzten Anwendungen. Daraus ergeben sich auch die Designziele bei der Entwicklung von Dynamo, welche auf das Einsatzgebiet optimiert werden.

Berücksichtigt werden insbesondere die diversen Ausfallszenarien: Ausfall von Platten, Zusammenbruch von Routen, Zerstörung der Data Center, ... Bei möglichst jeglicher Nichtverfügbarkeit von Teilinfrastruktur soll die Verfügbarkeit von Daten in einem Schlüssel/Wert-Modell garantieren werden. Eine vollständige relationale Datenbank wird für diesen Teil von Amazons Anwendungen nicht benötigt. Als Beispiele gelten u.a. der Einkaufswagen, die Einstellungen, Sitzungsverwaltung oder die Bestseller-Listen.

Dynamo erreicht dies indem er eine Datenbank mit einem einfachen Key/Value Interface (Key Value Store) zur Verfügung stellt, welches in der

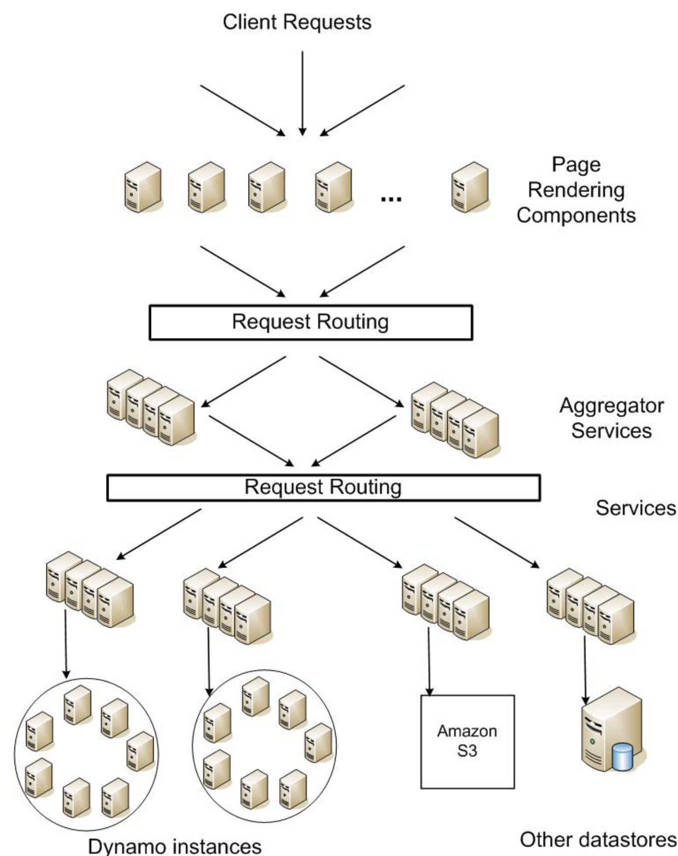

Abbildung 2.1: Position der Dynamo-Instanzen innerhalb Amazons Plattform

Arbeit vorgestellt wird. Dabei werden bereits bekannte Techniken für die Realisierung der Verfügbarkeit und Skalierbarkeit eingesetzt, auf welche im Folgenden ebenfalls näher eingegangen wird.

In Abbildung 2.1 wird die Position Dynamos innerhalb Amazons deutlich. Die eben genannten Anwendungen (Services) sind an die Dynamo-Key Value Stores angeschlossen. Andere Anwendungen verwenden aber auch andere Datenbanken, wie z.B. Amazon S3. Ein Seitenaufbau (in der Page Rendering Component) wird also über die Aggregator Services aus verschiedenen Datenquellen beantwortet. Damit wird klar, dass eine Amazon-Webseite nicht auf Dynamo als einziger Datenbank basiert.

Problem	Technologie	Vorteil
Partitionierung	Konsistentes Hashing	Schrittweise Skalierbarkeit
Hohe Verfügbarkeit bei Schreibzugriffen	Vektoruhren mit Abgleich während Leseoperationen	Trennung von Versionierung und Aktualisierungsort
Behandlung nicht-dauerhafter Ausfälle	Sloppy Quorum und Hinted Handoff	Hohe Verfügbarkeit und Garantie der Dauerhaftigkeit wenn Replikationsknoten nicht erreichbar sind
Wiederherstellung nach dauerhaften Ausfällen	Anti-entropy mit Merkle-Bäumen	Synchronisation abweichender Replikationsknoten im Hintergrund
Mitglieder und Ausfallerkennung	Gossip-basiertes Mitgliederprotokoll und Ausfallerkennung	Vermeidung zentraler Datenbanken und Knoten mit speziellen Aufgaben

Tabelle 2.1: Zusammenfassung der Technologien die in Dynamo zum Einsatz kommen und deren Vorteile [DHJ+07, 209]

2.2 Technologie

Um Verfügbarkeit, Zuverlässigkeit, Skalierbarkeit und Performance zu bieten, wird ein Partitionierungs- und Replikationsschema verwendet. Dazu wird ein konsistentes Hashing Verfahren eingesetzt. Das heißt, die Daten werden je nach Hash-Wert des Schlüssels auf unterschiedliche Knoten der Dynamo-Instanzen verteilt (Abschnitt 2.5).

Inkonsistenz der Daten können entstehen, wenn zwei verschiedenen Knoten unterschiedliche Änderungen des selben Objekts vornehmen. Um dies zu kompensieren, wird eine Objekt-Versionierung eingesetzt, welche zu jedem Schlüssel eine pro Knoten eigene Versionsnummer abspeichert. Unter Mitwirken der Anwendung kann so letztendlich eine Konsistenz erreicht werden (Abschnitt 2.6).

Des weiteren wird eine Quorum-Technologie eingesetzt, um sicherzustellen, dass Operationen von einer Mindestanzahl Knoten erfolgreich ausgeführt werden (siehe Abschnitt 2.7).

Zur Ausfallerkennung und Benachrichtigung zwischen den Knoten wird schließlich ein Gossip-basiertes Mitgliederprotokoll verwendet. Dieses verbreitet periodisch Informationen über alle vorhandenen Knoten. Dazu tauscht sich jeder Knoten sekündlich mit einen anderen, zufällig gewählten Knoten aus. Dieses Vorgehen ist jedoch bei einer großen Anzahl von Knoten nicht praktikabel.

In Tabelle 2.1 sind die einzelnen Technologien und deren Vorteile nochmals zusammengefasst.

2.3 Einschränkungen

Im Vergleich zu relationalen Datenbanken bringt das Design von Dynamo auch einige Einschränkungen mit sich. Der grundlegende Unterschied besteht darin, dass wie bei den Key Value Stores üblich nur ein Primärschlüssel zum Zugriff auf Daten verwendet werden kann. Dies ist in vielen Fällen aber ausreichend.

Weiterhin muss beachtet werden, ob durch die fehlende Erfüllung der ACID-Eigenschaften Einschränkungen zu erwarten sind. Dynamo verzichtet bewusst auf die ACID-Garantie, da insbesondere die Konsistenz mit schlechter Verfügbarkeit korreliert [DHJ+07, 206]. Isolationsgarantie bietet Dynamo ebenfalls nicht. Da Änderungen immer nur auf einem Schlüssel stattfinden, sind sie dort folglich atomar. Durch die Beschränkung auf die Abfrage nur eines Schlüssels ist die fehlende Atomarität der Transaktionen zumindest schon stark abgemildert, da Zugriffe über mehrere Schlüssel seltener vorkommen werden. Ein weiteres Problem entsteht möglicherweise durch die fehlenden Konsistenz (bzw. letztendlichen Konsistenz), welches softwareseitig gelöst werden muss.

Auch das Mitgliederprotokoll skaliert nicht endlos. Als mögliche Lösungen dieses Problems wäre eine Einführung von einer mehrschichtigen Knotenhierarchie oder die Verwendung von DHT (verteilten Hashtabellen) denkbar. Ein letzter Kritikpunkt ist die fehlende Optimierung für große Daten. Daher sollte je nach Anforderung die richtige Technologie ausgewählt werden. Innerhalb von Amazon stehen zum Beispiel Amazon S3 oder auch andere Datenbanken als weitere Datenspeicher zur Auswahl (siehe Abbildung 2.1).

2.4 Interface

Dynamo stellt ein simples Interface zur Interaktion mit Anwendungssoftware zur Verfügung:

get(*key*) Objekt mit Schlüssel *key* aus dem Speicher abrufen.

put(*key* , *object*) Objekt *objekt* unter dem Schlüssel *key* abspeichern/aktualisieren. Intern wird noch ein *context* mitgegeben, der Metadaten über das Objekt transportiert (darunter die Objektversion).

Als Datentypen für *key* und *object* werden verwendet:

	Datentyp
key	byte[]
object	byte[]

Dies bedeutet, dass es keinerlei Datentypen gibt. Stattdessen wird alles direkt binär abgelegt. Die Interpretation muss die Anwendungssoftware übernehmen. Der Schlüssel wird dann unter seinem MD5-Hash auf die Dynamo-Instanzen verteilt.

2.5 Partitionierung und Replikation

In diesem Abschnitt wird die Partitionierung, also die Verteilung der Schlüssel in einem Verband von Dynamo-Instanzen, näher erläutert. Zunächst besitzt jeder Knoten eine zufällige ID, genannt *token*. Diese gibt die „Position“ auf einem virtuellen Ring an, der den gesamten Bereich des MD5-Hashes in einzelne Abschnitte unterteilt (siehe Abbildung 2.2). Der gestrichelte Ring stellt den gesamten Bildbereich der Hashfunktion dar, darauf sind die Knoten A bis G verteilt. Soll nun der Schlüssel K an der gekennzeichneten Stelle gespeichert werden, so wird der Knoten B als Speicherort ausgewählt, da dessen Position die nächsthöhere nach dem Hashwert von K ist.

Der Vorteil dieses Vorgehens ist, dass der Wegfall eines Knotens nur dessen Nachbarn auf dem Ring betrifft. Ein Nachteil ist jedoch, dass die zufällige Zuordnung von Knoten an Positionen auf dem Ring zu Ungleichverteilungen von Last führen kann. Um dies auszugleichen, führt Dynamo virtuelle Knoten ein. Dabei werden jedem physischen Knoten mehrere token zugeordnet, was ihn so gleichzeitig für mehrere Positionen zuständig macht. Bei einer großen Anzahl von zufällig besetzten Positionen auf dem Ring verteilt sich die Last

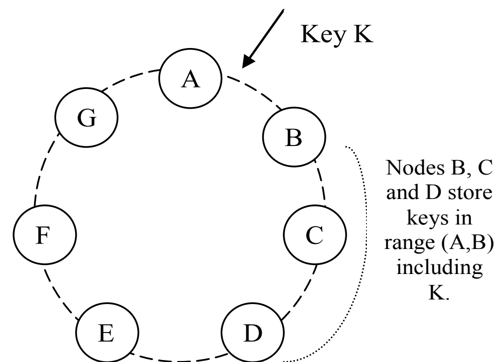

Abbildung 2.2: Partitionierung und Replikation [DHJ+07, 209]

gleichmäßiger. Außerdem kann die Anzahl der virtuellen Knoten je nach Leistungsfähigkeit des physischen Knotens angepasst werden.

Um Ausfälle zu mildern müssen die gespeicherten Daten auf mehrere Knoten repliziert werden. Dazu bedient sich Dynamo eines einfachen Replikationsschemas welches ebenfalls auf der Ringstruktur basiert. Die Daten werden zu den nächsten N Nachfolgeknoten repliziert, wobei N vorher festzulegen ist und je nach Anwendungsfall optimiert werden kann. Je höher der Wert von N gewählt wird, desto mehr Kopien der Daten existieren (und das System ist somit sicherer gegen Ausfall); Abbildung 2.2 zeigt ein Beispiel für den Fall $N = 2$. Der Schlüssel K wird dann neben dem Speichern in dem für ihn zuständigen Knoten B weiterhin noch in den zwei Nachfolgeknoten C und D repliziert.

2.6 Versionierung

Da Dynamo ein verteiltes System ist, kann es auf unterschiedlichen Knoten zu mehreren divergenten Versionen derselben Daten kommen. Dies resultiert aus dem Wunsch, schnelle Antwortzeiten zu garantieren. Dann kann nämlich beim Ausfall (Nichterreichbarkeit) des nach dem Partitionierungsschemas zuständigen Knotens eine Operation von einem anderen Knoten bearbeitet werden. Klassisch würde man sagen, die Daten sind damit inkonsistent geworden.

Die Verwendung von Vektorzeit ermöglicht jederzeit eine Erkennung solcher Zustände. Die hierdurch aufgezeichnete Versionsgeschichte liefert alle Informationen über die am Schreiben beteiligten Knoten. Das Dynamo-System sucht die aktuelle(n) Versionen und kann so in den meisten Fällen letztendlich

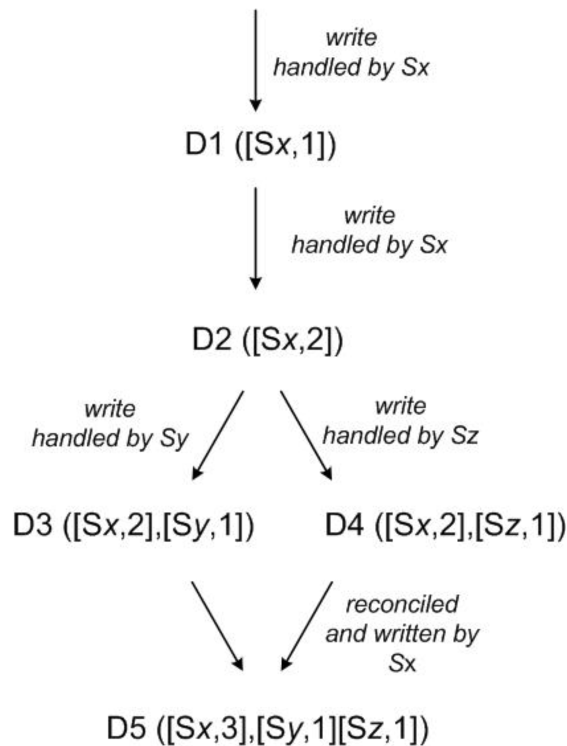

Abbildung 2.3: Versionierung im zeitlichen Verlauf [DHJ+07, 211]

eine Konsistenz herstellen. Wenn aber Änderungen von einer Version ausgehend auf zwei verschiedenen Knoten gespeichert wurden, kann es zu Konflikten kommen, die gegebenenfalls durch die Anwendungssoftware behoben werden müssen (vergleiche dazu auch [Bre10, CouchDB]).

In Abbildung 2.3 ist der zeitliche Verlauf der Versionierung illustriert, wobei die aktuelle Vektorzeit jeweils an den einzelnen Versionen des Datensatzes (D1...D5) stehen. Jeder Knoten hat eine eigene Dimension in der Vektorzeit. Die Knoten heißen in diesem Beispiel Sx, Sy und Sz. Solange das Ändern der Daten D immer von Knoten Sx übernommen wird, wie es bei D1 und D2 der Fall ist, gibt es keine Inkonsistenz. Wenn danach jedoch Sx temporär nicht mehr verfügbar ist und zwei unterschiedliche Schreiboperationen auf Basis des Datenstandes D2 von zwei verschiedenen Knoten übernommen werden (hier Sy und Sz), existieren mit D3 und D4 zwei inkonsistente Versionen. Wenn die Anwendungssoftware diese Versionen wieder zusammenführt, werden die Dimensionen aller Vektorzeiten kombiniert. So hat D5 in seiner Version sowohl den Versionsstand bei Sx als auch Sy und Sz angegeben. Die aktuelle Version ist also die mit den höchsten Versionsnummern in allen Vektordimensionen (sofern vorhanden).

2.7 Ausfälle

Um besser gegen temporäre Ausfälle geschützt zu sein, wird eine Quorum-Technologie eingeführt. Dies ist eine Form der bestätigten Ausführung von Schreib- und Leseoperationen. Es wird eine Mindestanzahl von Knoten W und R vorgegeben, welche eine Schreib- bzw. Leseoperation bestätigen müssen, bevor die Operation als erfolgreich gewertet wird. Dadurch kann sichergestellt werden, dass eine Operation auch an genügend Knoten repliziert wurde, und somit durch den Ausfall eines Knotens nicht verloren geht. W und R sind auch konfigurierbar und lassen sich somit an die Anforderungen der Anwendungssoftware anpassen.

Eine weitere Technologie im Zusammenhang mit temporären Ausfällen ist das sogenannte Hinted Handoff. Dabei übernehmen weiter hinter liegende Knoten die Schreiboperation wenn Knoten nicht erreichbar sind an deren Stelle. Wenn der ursprünglich zuständige Knoten wieder erreichbar ist, werden die in der Zwischenzeit von anderen Knoten übernommenen Schlüssel auf den eigentlich zuständigen Knoten zurücksynchronisiert.

Bei der Synchronisation zwischen Replikationen kommen Merkle-Bäume zum Einsatz. Dies sind Hash-Bäume in denen jede Wurzel eines Teilbaums einen Hashwert ihrer Kindknoten berechnet. Die Blätter enthalten den Hash-Wert der Daten (*object*). Damit ist eine schnelle Überprüfung von Teilbäumen möglich, denn es müssen nur die Teilbäume mit abweichenden Hashwerten weiter verfolgt werden.

Um Informationen über alle vorhandenen Knoten eines Dynamo-Systems (die Mitgliederliste) zu verbreiten, wird ein Gossip-basiertes Mitgliederprotokoll eingesetzt. Permanentes Hinzufügen und Entfernen von Knoten benötigt dabei einen expliziten Befehl. Die Mitgliederliste wird dann von jedem Knoten aus sekündlich an zufällige Knoten übermittelt, so stellt sich irgendwann ein konsistenter Zustand ein. Spezielle Keim-Knoten („Seeds“) sind in der Konfigurationsdatei eingetragen und werden bevorzugt befragt, um eine Fragmentierung des Netzes zu verhindern. Ausfälle von Nachbarknoten werden lokal von den jeweiligen Knoten dadurch erkannt, dass sie in einer vorgegebenen Zeitspanne keine Antwort erhalten. Ein verteiltes Protokoll mit Informationsaustausch ist dabei nicht nötig.

3 Apache Cassandra

Cassandra [LM09a] wurde ursprünglich für den Einsatz in Facebook entwickelt, inzwischen aber als Open Source freigegeben und wird als Apache Incubator Projekt weiterentwickelt. Bei Facebook handelt es sich um ein riesiges soziales Netzwerk mit mehr als 250 Mio. Nutzern. Ähnlich wie Amazon verfügt der Betreiber über Datenzentren mit tausenden Komponenten, die auch von Ausfällen betroffen sein können. Cassandra ist ein verteilter, hoch skalierbarer Key Value Store für große Datenmengen, der nach seiner freien Veröffentlichung auch von anderen Webseiten, wie z.B. Digg, verwendet wird [cas09]. Er wurde auch von einem der an Dynamo beteiligten Autoren mitentwickelt.

3.1 Designziele

Bei der Entwicklung von Cassandra war es wichtig, große Datenmengen effizient verwalten zu können. Facebook sah sich mit dem Problem konfrontiert, dass das Durchsuchen des bisher auf MySQL laufenden Posteingangs (eine Art Webmail-Anwendung) insbesondere durch die zu der Zeit rasant steigende Nutzerzahl zu viel Zeit beanspruchte. [LM09b] Nutzer des sozialen Netzwerks wollen ihre Posteingänge durchsuchen, auf die täglich Milliarden Schreibzugriffe erfolgen. [LM09a, 1]

Aus diesen Anforderungen ergeben sich die Designziele von Cassandra. Es soll eine hohe Verfügbarkeit garantieren werden um die Nutzer der Onlineplattform zufrieden zu stellen. Starke Konsistenz kann im Problemfall vernachlässigt werden.

Cassandra ist von der technologischen Idee an Dynamo angelehnt, verwendet diese aber nicht direkt. Von Seiten der Verfügbarkeit werden ähnliche Ansprüche wie von Dynamo gestellt. Die einfache Key/Value Datenhaltung ist für das Einsatzziel jedoch nicht ausreichend. Vielmehr erfordert ein unter einem Schlüssel abgelegter Posteingang ein komplexeres Schema. BigTable bietet dazu schon eine gutes theoretisches Konzept (vergleiche [Bre10, BigTable]) an dem sich Cassandra folglich orientiert.

3.2 Speicherformat

Cassandras Speicherformat wurde nach der von Google vorgestellten BigTable modelliert. Ähnlich wie bei HBase (vergleiche dazu [Bre10, HBase]) werden hierbei Tabellen verwendet, bei denen im Gegensatz zum normalisierten Schema der relationalen Datenbanken für jeden neuen Wert eine neue Spalte eingefügt wird. Im Vorab werden sogenannte *column families* (Spaltenfamilien) definiert, bei denen der Typ (Simple oder Super) und die Sortierreihenfolge angegeben werden muss. Innerhalb der Spaltenfamilie können dann beliebig Spalten eingefügt werden, welche jeweils aus Name, Wert und Zeitstempel bestehen.

Ist der Typ der Spaltenfamilie auf Super eingestellt, lässt sich eine zusätzliche Hierarchieebene erreichen. Jede Spalte enthält statt Wert und Zeitstempel eine weitere (simple) Spaltenfamilie, also verschachtelte Spalten. Dies funktioniert aber nur einmal und nicht beliebig rekursiv.

Die verwendeten Datentypen sind:

	Datentyp
Name	String (ca. 16 bis 36 Byte)
Wert	beliebige Binärdaten

Für die Sortierreihenfolge stehen die Möglichkeiten der Sortierung nach Name oder nach Zeitstempel zur Auswahl. Beispielsweise ist die Sortierung nach Zeit für das Ausgangsproblem, dem Abspeichern der Facebook-Posteingänge, besonders nützlich.

3.3 Interface

Auch Cassandra stellt ein nicht wesentlich komplexeres Interface zur Verfügung, welches von verschiedensten Programmiersprachen aus angesteuert werden kann. Der komplexeste Teil ist der Aufbau des Pfads zur gewünschten Spalte (siehe dazu auch das Programmierbeispiel weiter unten).

insert(*table* , *key* , *column* , *value* , *ts* , *c*) Fügt den Wert *value* in der Spalte *column* ein.

> ***ts*** Zeitstempel
>
> ***c*** Konsistenz-Anweisung

get(*table* , *key* , *column* , *c*) Liest den Wert der entsprechenden Spalte.

remove(***table*** **,** ***key*** **,** ***column*** **,** ***ts*** **)** Entfernt einen Wert aus einer Spalte.

Selbstverständlich wird auch eine Bereichsabfrage unterstützt, d.h. es können auch alle Spalten mit Namen *von*...*bis* einem bestimmten Namen bzw. die ersten oder letzten n Spalten abgerufen werden. Zum Beispiel könnten alle Spalten abgefragt werden, deren Name mit „c“ oder „d“ beginnt, oder die jüngsten 20 Spalten abgerufen werden (eine Spalte würde hier einer Nachricht im Posteingang entsprechen).

Die Konsistenz-Anweisung kennt die Unterscheidung zwischen keinerlei Anforderungen an die Konsistenz, einem bestätigten Schreibvorgang oder einem Quorum (Mindestanzahl) von erfolgreichen Lese- bzw. Schreiboperationen.

Programmierbeispiel (hier in der Programmiersprache Perl).

```
my $key = 'student2'; my $timestamp = time;
$client->insert(
    'Studenten',
    $key,
    ColumnPath->new(
        {column_family => 'Vorlesung', column => 'v1'}
    ),
    'Cloud Seminar',
    $timestamp,
    ConsistencyLevel::ZERO
);
my $what = $client->get('Studenten', $key, ColumnPath->new(
        {column_family => 'Vorlesung', column => 'v1'}),
    ConsistencyLevel::QUORUM);
say $what->column->value; # Cloud Seminar
```

In diesem Beispiel wird zunächst der Wert „Cloud Seminar“ in die Spalte v1 der Spaltenfamilie Vorlesung geschrieben. Als Schlüssel wurde „student2“ angegeben. Die verwendete Tabelle (BigTable) heißt „Studenten“. Es wird keine Konsistenzgarantie gefordert. Danach wird der Wert wieder ausgelesen, diesmal muss eine Mindestanzahl von Knoten das Ergebnis liefern können.

3.4 Architektur

Cassandra setzt für die Partitionierung wie schon Dynamo auf konsistentes Hashing. Als kleine Besonderheit bleibt die Ordnung erhalten, die Sortierung der Schlüssel korrespondiert also mit der Sortierung der Hashwerte. Zur besseren Lastverteilung ändern sich die zugeteilten Hashwerte von nicht ausgelasteten Knoten dynamisch. Die Replikation kann auf Racks (Serverschränke) oder Datenzentren begrenzt werden, die Knoten im Cassandra-Ring wissen also, wo sie sich (physisch) befinden. Dies kann sinnvoll sein um Latenz gering zu halten.

Als Mitgliederprotokoll wird auch hier auf ein Gossip-basiertes Mitgliederprotokoll gesetzt. Es wird ebenfalls für Systemnachrichten eingesetzt. Genau wie Dynamo erfordert das permanente Hinzufügen und Entfernen von Knoten jedoch ein manuelles Bekanntmachen. Das Eintreten eines neuen Knotens in einen Cassandra-Ring startet mit dem Kopieren der Daten eines anderen, ausgelasteten Knotens. Danach werden diese beiden Knoten sich den Schlüsselbereich und somit die Arbeit teilen. Zur Ausfallerkennung wird, basierend auf dem Empfang (bzw. Nicht-Empfang) von Gossip-Nachrichten eines Knotens, dessen Ausfallwahrscheinlichkeit von jedem Knoten selbst ermittelt. Diese Idee war die erste Implementierung dieser Art.

Lese- und Schreiboperationen können an einen beliebigen Knoten im Cassandra-Verband gerichtet werden. Dieser Knoten leitet die Anfrage dann an den oder die Knoten weiter, welche für die Daten des betroffenen Schlüssels zuständig sind. Je nach Konsistenz-Anweisung wird das erste Ergebnis zurückgeliefert oder gewartet bis ein Quorum von Knoten die Anfrage beantwortet hat. Schreiboperationen verlaufen analog. Das System stellt sicher, dass ein Quorum an Knoten die Schreiboperation bestätigt hat. Hier entscheidet die Konsistenz-Anweisung, wie lange die Verbindung zur Anwendungssoftware blockiert bzw. ob der Schreibvorgang im Hintergrund bewältigt werden soll.

3.5 Datenspeicher

Cassandra speichert alle Daten sowohl auf der Festplatte als auch im Hauptspeicher. Im Hauptspeicher können natürlich nur die jeweils aktuellen Daten sowie die Indizes gehalten werden. Die Daten werden binär auf die Festplatten geschrieben. Zunächst landen sie in einem Commit-Log, welches sequenziell geschrieben wird. Die im Hauptspeicher befindliche Struktur wird ebenfalls sequenziell in verschiedene Dateien gespeichert, wenn sie zu voll wird. Dazu

werden Indizes für den Schlüssel und Startpositionen der einzelnen Spalten erstellt und abgespeichert.

Im Hintergrund läuft dann ein Prozess, der die vielen Speicherabbilder zusammenführt und einen „Superindex“ erstellt, der beschreibt, in welchen Dateien ein Schlüssel überhaupt vorkommt. Dieses Prinzip entspricht dem von Google als *compaction* bezeichneten Prozess. Wenn eine Anfrage ankommt, schaut Cassandra erst im Speicher, ob die angefragten Daten dort noch gespeichert sind. Wenn dies nicht zum Erfolg führt werden danach die entsprechenden Daten unter Zuhilfenahme der Indizes von der Festplatte geladen.

4 Zusammenfassung

Hier und in [Bre10] wurden fünf verschiedene Key Value Stores betrachtet. Es gibt darüber hinaus noch zahlreiche andere, wie in Kapitel 1 erwähnt wurde. Für jeden Einsatz und Anwendungsfall muss abhängig von den jeweiligen Anforderungen evaluiert werden, auf welches System zurückgegriffen werden soll. Von den fünf vorgestellten Systemen sind nur drei (Hadoop, CouchDB, Cassandra) frei zugänglich, eins hingegen (Amazon Dynamo) ganz auf den internen Einsatz bei einem bestimmten Unternehmen beschränkt. Das fünfte (BigTable) steht den Nutzern nur indirekt durch die von dem Anbieter angebotene Service-Plattform zur Verfügung.

CouchDB eignet sich als Dokumentenspeicher, wobei die einzelnen Instanzen auch oft voneinander getrennt und ohne ständige (Netzwerk-)Verbindung existieren können und gegebenenfalls manuell oder unter Mithilfe des Anwenders synchronisiert wird. Die entsprechenden Aufgaben fallen vollständig der Anwendungssoftware zu.

BigTable ist nicht nur der Datenspeicher in Googles AppEngine, sondern steht inzwischen auch für ein allgemeines Konzept, welches auch in Hadoop und Cassandra umgesetzt wurde. Sie eignet sich somit für Anwendungen, in denen eine semistrukturierte Speicherung der Daten wie sie im BigTable-Konzept implementiert wird, sinnvoll ist.

Hadoop verfolgt einen eher zentralen Ansatz (Steuerserver der die Aufgaben verteilt), bietet dafür aber ein ausgeprägtes Map/Reduce Framework welches für verteilte Datenverarbeitung gut geeignet ist. Cassandra hingegen hat seine Stärken im Bereich der verteilten Datenhaltung in einem P2P-artigen Netzwerk. [Ste09]

Dynamo spielt für den Endanwender keine Rolle. Es ist auch nur als Speicher von eher kleineren Datenmengen pro Schlüssel in einem schnellen Datenzentrum sinnvoll. Die Dokumentation der verwendeten Technologien hat aber zur Entwicklung von dem öffentlich verfügbaren Cassandra geführt.

Literaturverzeichnis

[Bre10] Jonas Brekle: *Key Value Stores – BigTable, Hadoop, CouchDB*. Seminararbeit Cloud Data Management, Universität Leipzig, 2010.

[cas09] *The Apache Cassandra Project*, 2009. http://incubator.apache.org/cassandra/, besucht: 27. Dezember 2009, Online.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall und Werner Vogels: *Dynamo: amazon's highly available key-value store*. In: *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, Seiten 205–220, New York, NY, USA, 2007. ACM, ISBN 978-1-59593-591-5.

[LM09a] Avinash Lakshman und Prashant Malik: *Cassandra – A Decentralized Structured Storage System*. In: *LADIS 2009: The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, New York, NY, USA, 2009. ACM.

[LM09b] Avinash Lakshman und Prashant Malik: *Cassandra: Structured Storage System over a P2P Network*, 2009. http://static.last.fm/johan/nosql-20090611/cassandra_nosql.pdf, besucht: 30. Dezember 2009, Online.

[See09] Marc Seeger: *Key-Value stores: a practical overview*. Seminararbeit Ultra-Large-Sites, Hochschule der Medien Stuttgart, 2009. http://blog.marc-seeger.de/assets/papers/Ultra_Large_Sites_SS09-Seeger_Key_Value_Stores.pdf, besucht: 02. Januar 2010, Online.

[Ste09] Bradford Stephens: *HBase vs. Cassandra: NoSQL Battle!* Road to Failure, 2009. http://www.roadtofailure.com/2009/10/29/hbase-vs-cassandra-nosql-battle/, besucht: 18. Dezember 2009, Online.