Skalierbarer und zuverlässiger Zugriff auf feingranulare Daten

Christopher Eibel
Friedrich-Alexander-Universität Erlangen-Nürnberg
christopher.eibel@informatik.stud.uni-erlangen.de

ABSTRACT

Mit der Bereitstellung theoretisch unbegrenzter Speicherkapazitäten durch das Cloud Computing, sind neue Ansätze bei der Entwicklung von Dateisystemen gefragt. Einerseits spielt die Skalierbarkeit eine wesentliche Rolle, denn auch bei sehr vielen Komponenten muss ein solches System noch ein hohes Maß an Performanz bieten. Bei Belastungsspitzen müssen umgehend und spontan neue Rechner der Cloud hinzugefügt werden können, so dass sich die Daten und damit die Last gleichmäßig verteilen können. Ein Zusammenbrechen des gesamten Systems soll in diesem Fall verhindert werden. Andererseits darf unter den geforderten schnellen Antwortzeiten aber nicht die Zuverlässigkeit leiden – die Daten müssen stets hochverfügbar bleiben.

Welche Herausforderungen bei der Implementierung zu bewältigen sind, soll anhand von Amazons verteiltem Dateisystem Dynamo, welches seit 2005 in der Praxis Verwendung findet [1], aufgezeigt werden.

1. EINFÜHRUNG

Der Begriff des Cloud Computings ist zwar nicht genau definiert, es gibt jedoch einige Merkmale, über die man sich in der Literatur einig ist. Dazu gehört neben Virtualisierung, verteiltem Rechnen und dem Utility-Grid, auch der des Data-Centers [2]: Einrichtungen, die ein Netz von vielen Rechnern beherbergen [3], selbst aber auch wieder mit anderen Daten-Zentren (weltweit) vernetzt sein können. Das ermöglicht einer Cloud das Bereitstellen riesiger Speicherkapazitäten.

Grundsätzlich muss – gerade bei Größenordnungen von Tausenden von Servern, bestehend aus günstiger Commodity-Hardware – immer damit gerechnet werden, dass zu jeder Zeit Störungen auftreten können, die entweder zu einem temporären oder gar zum totalen Ausfall von Rechnern eines Netzwerks führen können. Zudem können einzelne Netzwerkverbindungen zwischen Rechnerknoten ausfallen oder ganze Datenzentren durch Katastrophen beeinträchtigt sein. Das kann bei einem naiven Ansatz dazu führen, dass die auf den Speichermedien der Rechner befindlichen Daten entweder für eine bestimmte Zeit oder im schlimmsten Fall überhaupt nicht mehr verfügbar sind. Große E-Commerce-Plattformen wie Amazon oder Web-Dienste wie Facebook, eBay und Twitter können sich solche Ausfälle nicht leisten, weil darunter direkt die Nutzerzufriedenheit und damit der Erfolg des Unternehmens leiden würde. Eigens entwickelte, spezialisierte Softwarelösungen sind für darunter liegende Dateisysteme nötig, die im Idealfall transparent für den Nutzer eine automatische Wiederherstellung der Daten gewährleisten oder noch verfügbare Rechnerknoten hinzuschalten können.

Die Zuverlässigkeit spielt also eine wesentliche Rolle für den Erfolg eines solchen Dateisystems, ebenso wie seine Schnelligkeit (geringe Latenzzeiten). Dabei müssen jedoch immer Kompromisse eingegangen werden. So existieren ähnliche Kriterien wie sie beispielsweise für relationale Datenbankverwaltungssysteme (RDBVS) existieren: Die sogenannten ACID-Eigenschaften (Atomicity, Consistency, Isolation und Durability). Hierbei wendet man sich jedoch vom Ansatz eines RDBVS ab, was zwar durch Recovery- und Logging-Mechanismen ausgesprochen zuverlässig ist, aber bezüglich der Skalierbarkeit und Performanz eher ungeeignet ist.

Der Kompromiss besteht nun darin, dass die zwingend notwendigen Eigenschaften Konsistenz (Consistency), Verfügbarkeit (Availability) und Zuverlässigkeit (Reliability) nicht gleichzeitig vollständig erfüllt werden können, sondern bei mindestens einer eine Abschwächung zu treffen ist.

Im Folgenden wird *Dynamo: Amazon's Highly Available Key-value Store* [1] genauer behandelt. Key-Value-Stores sind Speichersysteme, bei denen die Speicherung mittels Schlüssel-Werte-Paaren erfolgt, d.h. jedem abzuspeichernden Datenobjekt (Wert) wird ein eindeutiger Schlüssel zugewiesen. Die Arbeit schließt mit einem kurzen Vergleich ähnlicher am Markt befindlicher Systeme ab.

2. AMAZON DYNAMO

Dynamo ist ein dezentralisiertes, verteiltes Dateisystem, das größtenteils in Java implementiert wurde. Derzeit wird es nur für die internen Anwendungen (im Folgenden auch als Services oder Dienste bezeichnet) von Amazon genutzt, was gleichzeitig bedeutet, dass dahingehend einige Optimierungen, aber auch Vereinfachungen vorgenommen wurden. Beispiele für einige dieser Dienste sind der Shopping-Cart-Service (Warenkorb), der Produktkatalog oder das Abrufen von Details eines Artikels, indem zusätzliche Informationen wie eine Bücherbeschreibung, Autorname(n) oder Kundenrezessionen abgerufen werden.

2.1 Aufbau der Architektur

Amazon baut auf hunderten von Service-Anwendungen auf, welche alle über das Netzwerk erreichbar sind. Um diese Anwendungen bedienen zu können, sind viele der eingangs angesprochenen Rechenzentren nötig, die weltweit verteilt wiederum aus tausenden von Servern zusammengesetzt sind. *Abbildung 1* zeigt den schematischen Aufbau dieser Infrastruktur.

Eine Client-Anfrage wird zunächst von den Seiten-Erstellungs-Komponenten (Page-Rendering-Components) entgegengenommen. Deren Aufgabe ist es, den eigentlichen Inhalt

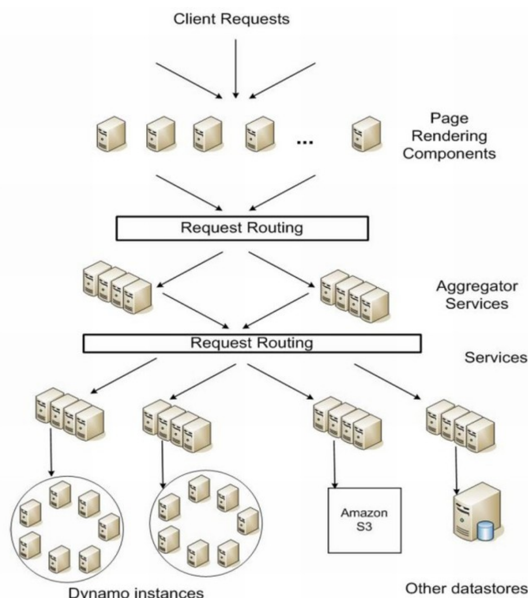

Abbildung 1: „Amazons service-orientierte Architektur“. [1]

einer angeforderten Webseite zu generieren, der dann von einem Browser angezeigt werden kann. Allerdings sind diese Seiten-Erstellungs-Komponenten nicht autonom, sondern müssen selbst wieder Anfragen an andere Dienste, den Sammel-Diensten (Aggregator-Services), stellen. Auf der untersten Schicht, die für die Datenspeicherung verantwortlich ist, kann neben den hier besprochenen Dynamo-Instanzen auch noch auf weitere Storage-Dienste wie Amazon S3 zugegriffen werden, falls das die Art der Daten erfordert. Dynamo ist nur auf feingranulare Daten unter einem Megabyte ausgerichtet. Dabei müssen von den Diensten sogenannte Service Level Agreements (SLAs) eingehalten werden: Jeder Dienst muss garantieren, innerhalb einer festgelegten Zeitspanne ein Ergebnis zu liefern. Dieses Abkommen ist nötig, damit eine gewisse Gesamtlatenzzeit eingehalten werden kann. Falls ein Dienst in Ausnahmefällen ein Zeitlimit nicht einhalten kann, spielt dessen Teilergebnis für das Gesamtergebnis keine Rolle mehr. Zu beobachen ist dies an den eingeblendeten Zusatzinformationen auf Amazon, welche auf der fertig gerenderten Webseite manchmal nicht angezeigt werden (z.B. aktuelle Produktempfehlungen). Die Vermutung liegt nahe, dass für diese Dienste die Zeitschranken aufgelockert wurden [4]. Dynamo gibt jedem Dienst die Möglichkeit, eigenverantwortlich seine Systemparameter festzulegen, so dass ein optimaler Kompromiss zwischen Funktionalität, Geschwindigkeit und Kosteneffizienz gefunden werden kann. Würde die Optimierung der Zugriffszeiten nach dem Durchschnitt oder dem Median erfolgen, so würde zwar die Mehrheit der Kunden davon profitieren, es blieben allerdings auch Nutzer übrig, deren Anfragen nur unzureichend befriedigt werden. Um eine maximale Nutzerzufriedenheit zu erreichen, sind die SLAs bei Amazon so ausgelegt, dass 99,9% *aller* Anfragen innerhalb einer bestimmten Zeitspanne (z.B. 300 ms) bedient werden können.

Eine der angesprochenen Vereinfachungen zeigt sich darin, dass die komplette Infrastruktur als vertrauenswürdig angesehen wird. So waren bei der Implementierung keine Mechanismen zur Authentifizierung oder Autorisierung (Rechtemanagement), wie sie für eine Public Cloud nötig wären, zu berücksichtigen; die Dynamo-Stores befinden sich von außen abgeschottet in einer Private Cloud und werden nur von den Amazon-eigenen Diensten genutzt. Weiterhin ist Dynamo auf Anwendungen spezialisiert, die keine komplexe, relationale Anfragesprache benötigen. Die Daten sind also ohne spezielle Schemata abgespeichert.

2.2 Design-Entscheidungen und Anforderungen

Das oberste Ziel besteht darin, durch Key-Value-Zugriffe eine hohe Verfügbarkeit bereitzustellen, so dass auch bei schwerwiegenden Ausfällen zu jeder Zeit Änderungen durchgeführt werden können und nicht zurückgewiesen oder unterschlagen werden.

Die Schnittstelle besteht nur aus den Operationen `put()` und `get()`. Das Objekt und sein Schlüssel besitzen keine eigenen Datentypen, sondern bestehen nur aus Byte-Arrays:

- **`get(byte[] key)`** Lokalisiert das zum Schlüssel *key* zugehörige Objekt. Die zurückgegebene Liste enthält entweder nur ein Element oder im Falle von Versionskonflikten mehrere Elemente (siehe Kapitel 2.3.1).

- **`put(byte[] key, Context c, byte[] object)`** Veranlasst das Schreiben der Replikate (siehe Kapitel 2.3.1) des Objekts *object* mit dem Schlüssel *key* auf die Festplatte(n). Der Kontext *c* enthält Metadaten des Objekts, z.B. eine Liste von Vektoruhren (siehe Kapitel 2.3.2) und wird immer zusammen mit dem Objekt auf das Speichermedium abgelegt.

Im Vordergrund steht außerdem, dass auch beim Hinzufügen neuer, einzelner Knoten die Antwortzeiten gering gehalten werden sollen („incremental scalability“ [1]).

Zur vereinfachten Instandhaltung des Systems trägt bei, dass alle Knoten die selben Aufgaben bzw. Verantwortungen tragen und keiner eine spezielle Rolle einnimmt (Symmetrie). Die Dezentralisierung, die als eine Art Erweiterung der Symmetrie verstanden werden kann, hat wesentlichen Einfluss auf die Ausfallsicherheit des Systems. Den Entwicklern von Dynamo zufolge haben bisherige Ansätze mit zentralisierter Koordination zu erhöhten Ausfallraten geführt. Darum war es bei Dynamo das Ziel, diese Art von Koordination zu vermeiden, was zudem einer erhöhten Skalierbarkeit zu Gute kam.

Ein weiterer wichtiger Punkt, der oft im Zusammenhang mit solchen verteilten Systemen auftaucht, ist der der Heterogenität. Abweichungen in der Hardware wie verschiedene Netzwerkkomponenten oder Prozessoren sind zu berücksichtigen [5]. Dynamo versucht nicht nur die einzelnen, heterogenen Server kompatibel zu machen; vielmehr möchte man die Unterschiede auch gezielt optimieren, wozu beispielsweise eine Ausnutzung erhöhter Speicherkapazitäten einiger Rechner zählen.

2.3 Implementierung

Dynamo macht von einigen fundierten Verfahren der Informatik Gebrauch, die schon seit längerer Zeit existieren und nicht extra bei dessen Entwicklung erforscht wurden. *Tabelle 1* gibt einen Überblick über die wichtigsten Verfahren, die im weiteren Verlauf genauer erläutert werden.

Problem	Technik	Vorteil(e)
Partitionierung	Konsistentes Hashing	Stufenweise Skalierbarkeit
Hochverfügbarkeit beim Schreiben	Vektoruhren mit Abgleich während den Lesevorgängen	Versions-Overhead unabhängig von der Updaterate
Temporäre Ausfälle	Sloppy Quorum und Hinted Handoff	Stellt Hochverfügbarkeit und Dauerhaftigkeit sicher, auch wenn einzelne Replikate nicht verfügbar sind.
Wiederherstellung bei permanenten Fehlern	Anti-Entropie mittels Merkle-Bäumen	Synchronisation unterschiedlicher Replikate im Hintergrund
Mitgliedsfindung und Ausfallerkennung	Gossip-basiertes Protokoll	Kein zentraler Koordinator nötig

Tabelle 1: Überblick über die von Dynamo verwendeten Techniken und ihre Vorteile. [1]

2.3.1 Konsistentes Hashing zur Partitionierung und Replikation

Um Skalierbarkeit, aber auch einen ausgewogenen Lastausgleich, zu gewährleisten, wird Partitionierung verwendet. Bevor ein Wert abgspeichert werden kann, muss ihm ein eindeutiger Schlüssel zugewiesen werden. Aus dem Schlüssel wird wiederum mittels einer Hashfunktion der Hashwert berechnet. Im konkreten Fall handelt es bei dem von Dynamo angewandten Algorithmus der Hashfunktion um MD5; die Länge des Hashwerts beträgt 128 Bit. Der Wertebereich der Hashfunktion wird auf einen Ring abgebildet, auf dem die Rechnerknoten logisch verteilt werden. Im Allgemeinen sind die Rechner physikalisch anders angeordnet. Die Position eines Rechnerknotens wird dabei durch einen zufälligen Wert festgelegt. Zu welchem Knoten ein bestimmter Wert korrespondiert, geschieht auf folgende Weise: Der zugehörige Schlüssel des Wertes wird mittels der Hashfunktion auf den Ring abgebildet. Danach wird der Ring im Uhrzeigersinn abgelaufen, um den direkt nachfolgenden Knoten in dieser Richtung zu ermitteln. Der gefundene Knoten ist dann für die Speicherung verantwortlich.

Abbildung 2 zeigt exemplarisch einen Ring mit fünf Knoten, welche zufällig und nicht äquidistant verteilt sind. Ein Wert mit dem Schlüssel K soll abgespeichert werden. Die Position auf dem Ring (Pfeil zwischen den Knoten A und B) wurde durch die Hashfunktion ermittelt. Da der gesamte Bereich zwischen A und B (genauer gesagt: (A,B]; rot markiert) zum Knoten B korrespondiert, wird der Wert folglich auf B abgespeichert.

Replikation Gleichzeitig wird in *Abbildung 2* noch eine weitere Technik ersichtlich: Die Replikation. Um erhöhte Dauerhaftigkeit und Verfügbarkeit zu erreichen, wird ein Datum nicht nur auf einem Knoten abgespeichert, sondern auf mehreren. Eine Besonderheit ist dabei, dass diese Anzahl N an Replikaten inviduell konfigurierbar ist. Die Wahl der

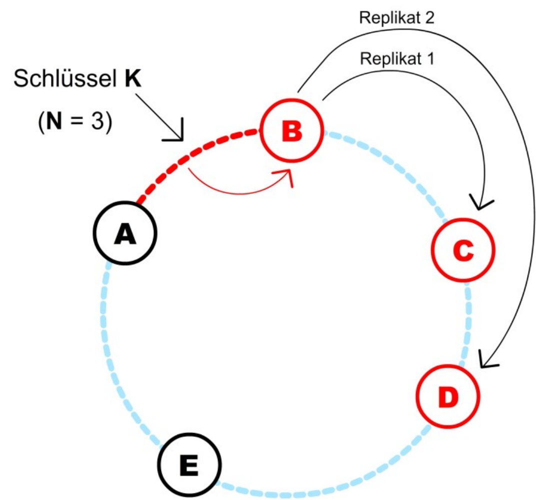

Abbildung 2: Beispiel zur Anordnung der Rechnerknoten auf dem Ring mit Replikation.

N-1 Knoten, die die Replikate speichern, kann nach dem gleichen Prinzip erfolgen wie bei der Ermittlung des ersten Knotens (Im Beispiel: B): Der Ring wird weiter im Uhrzeigersinn abgelaufen bis die N-1 Nachfolger ermittelt wurden. Zurück im Beispiel heißt das für eine Wahl von N = 3, dass der Knoten B, der hier eine spezielle Art Koordinatorrolle einnimmt, all seine Daten auf die Knoten C und D repliziert. Das widerspricht auch nicht der Symmetrie, die besagt, dass jeder Knoten keine spezielle Rolle einnehmen darf. Da die Replikation gleichermaßen für alle Knoten gilt, kann folglich jeder Knoten diese Rolle des Koordinators einnehmen – die Symmetrie-Eigenschaft bleibt erfüllt.

Wird ein Knoten entfernt, so wird der Wertebereich des direkten Nachfolgeknotens um den Wertebereich dieses entfernten Knotens erweitert; die Replikation erfolgt fortan auf die Replikatsknoten des Nachfolgers.

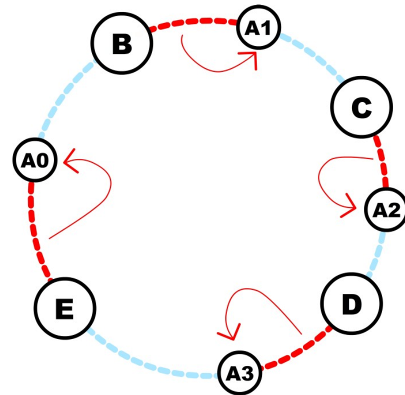

Abbildung 3: Beispiel zur Aufteilung eines Rechnerknotens (ursprünglich A in *Abbildung 2*) auf virtuelle Knoten (A0-A3).

Virtuelle Knoten Nicht alle Rechner im Netzwerk bzw. in der Cloud verfügen über die gleichen Eigenschaften. Insbesondere verfügen die Computer über unterschiedliche Speicher- und Rechenleistungen. Das führt dazu, dass mit Ungleichmäßigkeiten bezüglich der Datenaufteilung auf die

einzelnen Speichersysteme zu rechnen ist. Genau das möchte man jedoch vermeiden; die Daten sollen möglichst so verteilt werden, dass sich selbst bei Belastungsspitzen eine gleichmäßige Verteilung der Last ergibt. Um dies zu erreichen, wurden *virtuelle Knoten* eingeführt: Ein Rechnerknoten kann auf mehrere virtuelle Knoten auf dem Ring abgebildet werden. Ein Knoten, wie er in *Abbildung 2* dargestellt ist, kann also mehrere Plätze (= *Tokens*) auf dem Ring einnehmen. Das bedeutet, dass ihm ein größerer Wertebereich zugewiesen wird und er demzufolge mehr Daten abspeichern kann. Die Tokens müssen nicht alle aufeinanderfolgend auf dem Ring abgebildet werden, sondern können wie in *Abbildung 3* auf dem Ring verstreut werden.

Der ursprüngliche Knoten A aus *Abbildung 2* wurde auf die virtuellen Knoten A0 bis A3 in *Abbildung 3* aufgeteilt. Dieses Vorgehen hat den Vorteil, dass im Falle des Wegfalls eines solch zerstückelten Knotens dessen Last gleichmäßig auf die anderen Knoten aufgeteilt wird. Würde der Knoten A wirklich nur einen Token des Rings besitzen und dieser sich zwischen E und B (wie in *Abbildung 2*) befinden, so würde sich seine Last nur auf B und damit C und D (Replikation) verteilen – der Knoten E bliebe unberücksichtigt. Gleiches gilt auch für die umgekehrte Richtung: Beim Hinzufügen eines neuen Knotens kann man durch geschickte Aufteilung und Platzierung der virtuellen Knoten dafür sorgen, dass er jeweils möglichst gleich viel Last von den anderen Knoten übernimmt.

Vorzugsliste (Preference-List) Jeder Knoten unterhält eine sogenannte Vorzugsliste, in welcher für jeden Schlüssel alle Knoten, auf die repliziert werden kann, abgespeichert sind. Das müssen nicht notwendigerweise genau N Knoten sein; zur Erinnerung: N ist die Anzahl der Tokens, auf die ein Datum abgespeichert werden soll. In der Praxis wählt man Werte größer als N, da immer wieder einzelne Knoten ausfallen können. Dadurch kann bei solchen Ausfällen der eigentlich festgelegte Wert N dennoch erreicht werden.

Durch die Vorzugsliste muss die Replikation auch nicht auf die direkten Nachfolger eines Knotens erfolgen. Virtuelle Knoten können nämlich direkt aufeinanderfolgen, d.h. es würde bei Replikation auf die direkten Nachfolger mehrmals auf ein- und denselben Rechner repliziert werden. Hinsichtlich der Hochverfügbarkeit ist dies nicht sinnvoll, weil mit dem Ausfall dieses einen Rechners alle Replikate verloren gehen. Aus diesem Grund können einzelne (virtuelle) Knoten auf dem Ring beim Hinzufügen zur Vorzugsliste übersprungen werden. In der Regel geht man noch einen Schritt weiter: Da immer die Möglichkeit gegeben ist, dass ganze Rechenzentren ausfallen können (Stromausfälle, Kühlungsprobleme, Netzwerkprobleme, Katastrophen), wird versucht, die Replikate über möglichst viele Standorte zu verteilen. Im Falle eines Komplettausfalls eines Data-Centers besteht so eine erhöhte Wahrscheinlichkeit, dass nicht alle Replikate betroffen sind.

Hinted Handoffs und Merkle-Bäume Falls bei der Replikation einer der Knoten, der eigentlich ein Replikat erhalten sollte – im Beispiel mit N = 3: einer der Knoten B, C oder D – ausfällt, kann ein anderer Knoten (hier: E oder A) hinzugezogen werden, welcher das Replikat stattdessen erhält. Dieser Knoten wird das Replikat in einer separaten Datenbank mit einem Hinweis („Hint") im Kontext abspeichern, der auf den ausgefallenen Knoten verweist („Handoff"). Sobald der ursprünglich ausgefallene Knoten wieder erreichbar ist, kann das Replikat aus der lokalen Datenbank gelöscht werden, nachdem es zu ihm übertragen wurde . Um dies zu bewerkstelligen, müssen diese lokalen Datenbanken periodisch durchsucht werden.

Damit hat sich aber leider eine weitere Fehlerquelle aufgetan: Der Knoten, welcher ein per Hinted Handoff repliziertes Objekt beherbergt, kann auch selbst ausfallen, bevor er das Replikat an den ursprünglichen Knoten weitergegeben hat. Deshalb führt jeder ausgefallene Knoten nach Wiederanlauf einen Abgleich mit den Knoten seiner Vorzugsliste durch, um die Replikate zu synchronisieren. Dieser Vorgang wird als Anti-Entropie bezeichnet [6]. Um den dabei notwendigen Overhead beim Datenaustausch zu minimieren, werden Merkle-Hash-Bäume verwendet. Diese Bäume speichern in der Wurzel und in den inneren Knoten Hash-Werte, die aus den direkten Kindknoten berechnet werden. An den Blättern befinden sich die eigentlichen, eindeutig zu einem Schlüsselwert zugehörigen Hash-Werte. Jeder Knoten legt für all seine Wertebereiche[1] solche Merkle-Bäume an. Beim Abgleich kann schon anhand der Wurzel entschieden werden, ob sich die darunter liegenden Replikate voneinander unterscheiden. Ist der Hash-Wert in der Wurzel gleich, muss nichts unternommen werden – die Replikate unterscheiden sich nicht. Unterscheiden sich die Wurzel-Hash-Werte, so müssen die Bäume rekursiv traversiert werden bis die abweichenden Replikate gefunden sind.

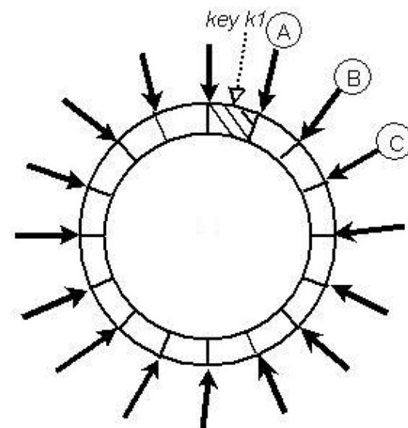

Abbildung 4: Verbesserte Partitionierungsstrategie mit fixen Partitionierungsbereichen. [1]

Verbesserte Partitionierungsstrategie Die oben angedeutete Partitionierungsstrategie hat generell einen entscheidenden Nachteil: Partitionierung und Datenplatzierung sind gekoppelt. Da jeder Knoten quasi seinen eigenen Wertebereich hat, führt dies dazu, dass beim Hinzufügen und Entfernen von Knoten umfangreiche und teure Neuverteilungen der Daten durchzuführen sind. Besonders bei Lastspitzen muss das Hinzufügen von weiteren Knoten aber sehr schnell gehen. Dies wird ermöglicht, indem der Ring im Vorfeld in Q äquivalente Bereiche aufgeteilt wird und jeder Knoten Q/S (S = Anzahl der physikalisch vorhandenen Knoten) virtuelle Knoten erhält (siehe *Abbildung 4*). Außerdem gilt: $Q \gg N$.

[1] Plural, da jeder Knoten im Allgemeinen in virtuelle Knoten zerlegt wird, welche verschiedenen Wertebereichen zugewiesen sind.

Wenn ein Knoten hinzugefügt wird, dann wird die Anzahl der virtuellen Knoten, die jeder Knoten erhält, verringert. Das Entfernen funktioniert analog, nur dass die Anzahl erhöht wird.

Abbildung 4 veranschaulicht dies: Die Gesamtzahl der schwarzen Pfeile bleibt immer gleich, solange sich Q nicht ändert. Fügt man einen neuen Knoten hinzu, übernimmt dieser eine bestimmte Anzahl an schwarzen Pfeilen; er nimmt sie sozusagen anderen Knoten weg. Die Daten, die sich hinter diesen schwarzen Pfeilen verbergen, müssen nun auf den neuen Knoten umkopiert werden. Die Daten, die sich hinter den anderen schwarzen Pfeilen befinden, bleiben hingegen unberührt.

2.3.2 Eventual Consistency und Versionierung

Der Kompromiss zwischen Verfügbarkeit, Zuverlässigkeit und Konsistenz besteht bei Dynamo darin, letztere Eigenschaft aufzulockern: *Eventual Consistency*; „letztendliche" Konsistenz. Diese letztendliche Konsistenz stellt einen Mittelweg zwischen strenger und schwacher Konsistenz dar [7, 4]:

- **Strenge Konsistenz:** Diese Art der Konsistenz garantiert bei Erfolg eines Schreibvorgangs, dass jeder nachfolgende Zugriff auf ein Replikat denselben Wert zurückliefert. Zudem ist die Reihenfolge der Änderungen für jedes Replikat gleich.
- **Schwache Konsistenz:** Bei dieser Art von Konsistenz garantiert das System *nicht*, dass jedes Replikat in der gleichen Reihenfolge Änderungen erhält. Es existiert ein Zeitfenster, in dem bereits veranlasste Änderungen nicht zu beobachten sind.
- **Letztendliche Konsistenz:** Bei der letztendlichen Konsistenz garantiert das System, dass *irgendwann* in endlicher Zeit alle Zugriffe auf die Replikate den zuletzt geänderten Wert zurückliefern. Die Änderungen erfolgen im Gegensatz zur strengen Konsistenz nicht atomar, sondern asynchron zum Änderungsbefehl.

Die strenge Konsistenz eignet sich zwar sehr gut für Datenbanksysteme, für ein Anwendungsgebiet, das hohe Verfügbarkeit erfordert, aber nur ungenügend. Um strenge Konsistenz erreichen zu können, müsste der Zugriff auf zu ändernde Daten nämlich so lange nicht möglich sein, bis alle Replikate geändert wurden (z.B. durch Locking-Strategien). Erst dann kann eine Änderung bestätigt werden. Unzureichende Latenzzeiten sind die Folge, welche mit der letztendlichen Konsistenz gesenkt werden.

Letztendliche Konsistenz muss jedoch explizit von den Anwendungen unterstützt werden, denn es besteht die Möglichkeit, dass Replikate Änderungen in unterschiedlichen Reihenfolgen entgegennehmen. Daher können durchaus inkonsistente Zustände im Speichersystem vorliegen.

Ein Beispiel einer Anwendung, das im Dynamo-Papier aufgeführt ist und mehrere Versionen tolerieren kann, ist der Shopping-Cart-Service. Die Grundfunktionalitäten sind *„Produkt zum Warenkorb hinzufügen"* und *„Produkt vom Warenkorb entfernen"*. Als oberstes Ziel darf die Hinzufügen-Operation niemals fehlschlagen oder verloren gehen. Kann die Anwendung aufgrund von Fehlern nicht auf den aktuellen Warenkorb zugreifen, sondern nur auf eine veraltete Version, so wird diese ältere Version trotzdem für die neue Änderung herangezogen. Später können dann diese Version und die zum Zeitpunkt vor der Änderung aktuellste Version miteinander vereinigt werden, so dass am Ende keine Verluste aufgetreten sind.

Versionierung mittels Vektoruhren Die inkonsistenten Zustände können auch als unterschiedliche Versionen betrachtet werden. Um diese wiederum unterscheiden zu können, verwendet Dynamo Vektoruhren. Jede Änderung erhält mittels dieser Vektoruhren eine Versionsnummer, so dass Listen von Knoten-Versionszähler-Paaren entstehen. Da irgendwann die Daten abgeglichen werden müssen, stellen sich die grundsätzlichen Fragen, *wann* dieser Abgleich erfolgen muss und *wer* diesen Abgleich durchführen soll.

Wenn keine Fehler aufgetreten sind, müssen nur unmittelbar aufeinanderfolgende Versionen vereinigt werden. Solche Änderungen können problemlos vom System selbst vorgenommen werden. Treten hingegen Fehler, gepaart mit nebenläufigen Schreiboperationen auf, können konfliktbehaftete Versionen entstehen. Das System kann nun nicht mehr selbst entscheiden, wie die Daten wieder zusammengefügt werden müssen. Die Daten werden damit aber keineswegs unbrauchbar, sondern an die Anwendung zurückgegeben, welche normalerweise in der Lage ist, darüber zu entscheiden, welche Daten in die neue, vereinigte Version gehören.

Der Zeitpunkt des Abgleichs ist hingegen eindeutig definiert: Er erfolgt immer beim Lesen. Würde er beim Schreiben erfolgen, bestünde abermals die Möglichkeit die Bedingung der Hochverfügbarkeit zu verletzen, welche zu jeder Zeit schnelles Schreiben erfordert („always writeable" [1]).

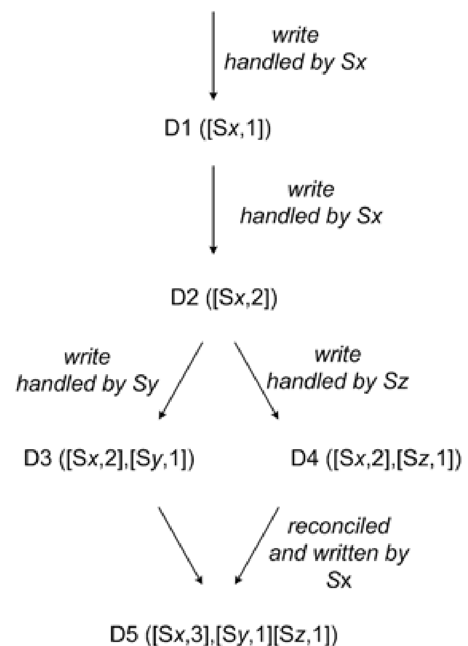

Abbildung 5: Versionsverlauf eines Objekts über die Zeit. [1]

Zur Erläuterung der Funktionsweise der Vektoruhren eignet sich eine Abbildung, die aus dem Dynamo-Papier entnommen wurde (*Abbildung 5*).

Initial seien alle Vektoruhren zurückgesetzt (Versionszähler = 0). Ein Datenobjekt soll geschrieben werden. Dazu wird zunächst anhand dessen Schlüssel und mittels der Hashfunk-

tion der korrespondierende Knoten ermittelt, welcher hier Sx ist. Die zugehörige Vektoruhr wird mit (Sx, 1) instanziiert und zusammen mit dem Objekt D1 abgespeichert. Der Client möchte nun erneut in das Objekt schreiben und ermittelt denselben Knoten Sx wie zuvor. Unter der Voraussetzung, dass während des Schreibens mit der Funktion `put()` keine Fehler aufgetreten sind, liefert `get()` nur ein Ergebnis, was implizit in seinem Kontext die Vektoruhr (Sx, 1) enthält. Im vorliegenden Fall kann das System selbst die Entscheidung treffen, dass das neue Objekt D2 von D1 abstammt. Der Versionszähler wird wieder um eins inkrementiert – die Vektoruhr (Sx, 2) wird mitgegeben.

Jetzt kommt es zu einem nebenläufigen Schreiben, weil die unterschiedlichen Knoten Sy und Sz in das Objekt D2 schreiben möchten. Zunächst lesen beide D2 und inkrementieren jeweils ihren Versionszähler. Bei diesen Lesevorgängen ist noch kein Abgleich möglich. Einmal wird D3 mit der Vektoruhr von Sy geschrieben und einmal D4 mit der Vektoruhr von Sz.

Nun wird angenommen, dass wieder Sx der Koordinator ist und beim Lesen D3 und D4 erhält. Dies ist der Fall, in dem das System die Entscheidung über den Abgleich dem Client überlässt. Sx inkrementiert seinen Versionszähler in der Vektoruhr und veranlasst das Schreiben des neuen Objekts D5 als Vereinigung (Entscheidung der Anwendung) der anderen beiden Datenobjekte und deren zugehörige Vektoruhren.

Konfigurierbarkeit Alle Lese- und Schreibvorgänge müssen ähnlich wie in einem Quorum von einer bestimmten Anzahl von Knoten bestätigt werden. Konkret wird die Anzahl der Knoten, die einen Lesevorgang bestätigen müssen, mit R notiert und die Anzahl der Knoten, die einen Schreibvorgang bestätigen müssen, mit W. Dementsprechend muss der Koordinator beim Schreiben, nachdem er selbst das Objekt lokal geschrieben hat, noch auf die Bestätigung von W-1 Knoten warten, bevor er den Schreibvorgang als Ganzes bestätigen kann. Der Lesevorgang funktioniert analog.

Da bei Ausfällen von Knoten andere Knoten als Ersatz herangezogen werden können, spricht man auch von einem sogenannten **Sloppy**[2] **Quorum** (keine strikte Mitgliedschaft im Quorum erforderlich).

Unterschiedliche Dienste haben unterschiedliche Ansprüche hinsichtlich Performance, Verfügbarkeit, Dauerhaftigkeit und Konsistenz. Durch Variieren der Werte des Quorum-Tripels (N, R, W) lassen sich diese Eigenschaften beeinflussen. Im speziellen Fall wählt man im Sinne der letztendlichen Konsistenz für W und R kleinere Werte als N, die aber dennoch die Ungleichung $W + R > N$ erfüllen (Quorum-Eigenschaft). Dadurch muss beim Lesen und Schreiben nicht auf die Bestätigung aller Replikatknoten gewartet werden, wodurch die Antwortzeit im Mittel gesenkt wird.

Das Dynamo-Papier schlägt konkrete Abgleichstrategien vor. Wird zum Beispiel ein hochperformantes Dateisystem für viele Lese- und wenige Schreibvorgänge benötigt, kann R auf den Wert 1 gesetzt werden. Das bedeutet, dass nur ein einziger Knoten den Lesevorgang bestätigen muss. Genauso gut kann aber auch der Wert von W auf 1 gesetzt werden, was maximale Verfügbarkeit zur Folge hat, da nur ein einziger Knoten den Schreibvorgang bestätigen muss. Das erhöht jedoch die Gefahr von Inkonsistenzen, weil theoretisch außer diesem einen Knoten im ungünstigsten Fall alle anderen Replikationsknoten ausgefallen sein könnten. Bei der Einstellung dieser Werte ist demzufolge Sorgfalt geboten.

In Dynamo wird für die meisten Dienste die Kombination (3, 2, 2) verwendet. Andere Dienste, wie der Dienst zur Session-Verwaltung, können sogar gänzlich auf einen genaueren Abgleich bei divergenten Versionen verzichten: Nur die Version mit dem neuesten physikalischen Zeitstempel ist entscheidend („last write wins“ [1]).

2.3.3 Mitgliedsfindung und Ausfallerkennung

Damit sich Knoten gegenseitig finden, verwendet Dynamo ein Gossip[3]-basiertes Protokoll. Jeder Knoten unterhält eine Mitgliederliste von Knoten im Netzwerk. Periodisch im Sekundentakt nimmt jeder Knoten mit einem anderen Knoten im Netzwerk Kontakt auf und gleicht mit ihm seine Mitgliedsliste ab.

Das Hinzufügen und Entfernen eines Knotens zum beziehungsweise vom Ring muss explizit mittels einer Administrationskonsole geschehen. Diese verbindet sich zu einem Knoten, der sich bereits auf dem Ring befindet und fügt in dessen Mitgliedsliste den neuen Knoten ein. Hier kommt wieder die letztendliche Konsistenz zum Tragen: Zunächst sind diese Änderungen nicht allen anderen Knoten bekannt, sondern werden erst mit der Zeit ins komplette Netzwerk propagiert. Durch diese temporärere Partitionierung des Rings kann es zu einer Wettlaufsituation kommen: Man stelle sich vor, dass der Ring aus *Abbildung 2* zunächst nur die Knoten C bis E enthält und nun noch die Knoten A und B fast zeitgleich eingefügt werden sollen. A wüsste zunächst nichts von B und B zunächst nichts von A. Würde nun ein `put()` erfolgen, was von A koordiniert wird, so wäre es eigentlich wünschenswert, dass auf B repliziert wird, allerdings weiß A ja noch nichts von B, so dass ein Ausweichknoten ausgewählt wird.

Wegen diesem Sachverhalt, wird die Eigenschaft der Symmetrie etwas gelockert: Einige Knoten nehmen die Rolle eines sogenannten „Seeders“ ein, die von allen Knoten von vornherein bekannt sind. Die Seeder sind bei einem Discovery-Service (Erkennungsdienst) registriert, so dass sie von außerhalb gefunden werden können. Der Administrator verbindet sich für das Hinzufügen beziehungsweise Entfernen nur zu diesen Seed-Knoten; alle anderen Knoten gleichen ihre Mitgliedsliste mit diesen Seedern ab. Das Problem wird zwar durch dieses Vorgehen nicht völlig eliminiert, wohl aber entscheidend entschärft.

Ausfallerkennung Die Ausfallerkennung von Knoten und das Gossip-basierte Protokoll sind eng miteinander verknüpft. Von einem eigenen, zentralisierten Ansatz kann abgesehen werden. Falls die Knoten nichts zu kommunizieren haben, ist das Wissen über den Ausfall anderer Knoten zweitrangig. Aus diesem Grund genügt es, Ausfälle dadurch zu erkennen, dass bei der Kommunikation mit einem Knoten keine Antwort erwidert wird. Erst dann wird in regelmäßigen, zeitlichen Abständen nach einem Wiederanlauf abgefragt.

2.4 Optimierungen

Neben der verbesserten Partitionierungsstrategie sind noch weitere Optimierungstechniken einsetzbar. Über die Aktivierung der ersten beiden vorgestellten Techniken entscheiden direkt die Programmierer einer Dynamo-Anwendung.

[2]engl.: „schlampig“

[3]engl.: „tratschen“

Load Balancing Bisher wurde davon ausgegangen, dass irgendein Knoten eine Schreib- oder Leseanforderung koordiniert. Dieser Knoten wird zunächst von einem Load Balancer ausgewählt. Die Anforderung gelangt somit erst durch den Umweg über diesen Load Balancer zum Koordinator („extra network hop“ [1]).

Die Optimierung besteht darin, direkt in den Anwendungen Bibliotheken mit Informationen über die Erreichbarkeit der Knoten zu pflegen. Um hohe Skalierbarkeit zu wahren, wählte man das Pull-Verfahren: Die Anwendungen fragen selbst periodisch und zufällig einzelne Knoten nach deren Mitgliedsinformationen ab. Das verhindert, dass die Knoten Referenzen zu allen Anwendungen (Observer) speichern müssen, um diesen bei Änderungen die neuen Informationen mitzuteilen (publish).

Die Anwendung kann so die einzelnen Anfragen auf direktem Wege selbstständig an einen passenden Knoten weiterleiten, muss aber auch erst entsprechend angepasst werden. Der Erfolg dieses Vorgehens hängt maßgeblich von der Aktualität der Informationen über die Knoten ab.

Pufferung Für Dienste, die maximale Performance benötigen, kann eine Pufferung hinzugeschaltet werden. Jeder Knoten hält hierbei eine bestimmte Anzahl von Objekten direkt im Hauptspeicher; das tatsächliche Schreiben auf einen persistenten Speicher erfolgt nicht sofort. Bei einem erneuten Lese-Zugriff wird zunächst der Puffer nach dem angefragten Objekt durchsucht. Die Zugriffe auf den Hauptspeicher sind zwar sehr schnell, allerdings riskiert man im Falle eines Serverabsturzes den Totalverlust der noch abzuarbeitenden Schreiboperationen im Puffer. Konsistenz und Dauerhaftigkeit werden zugunsten einer höheren Verfügbarkeit eingetauscht. Pufferung sollte also nicht pauschal für alle Dienste aktiviert werden und gut überlegt sein.

Admission Controller Im Allgemeinen bremsen verschiedene Hintergrundaktivitäten wie das Synchronisieren von Replikaten die im Vordergrund stehenden `get`- und `put`-Operationen aus. Der Admission Controller hat die Aufgabe, die Vordergrundaktivitäten zu überwachen und zu analysieren, um den Hintergrundaktivitäten nach Bedarf Zeitschlitze zu gewähren. Die Auslastung entscheidet über die Häufigkeit, mit der die Hintergrund-Aufgaben eingelastet werden. Dies trägt wieder zur Hochverfügbarkeit bei, da die im Vordergrund stehenden Aktivitäten höhere Priorität haben und schneller ausgeführt werden können.

2.5 Verwandte Dateisysteme

Mittlerweile existieren eine ganze Reihe von verteilten Dateisystemen. Beispiele anderer Key-Value-Stores sind *Cassandra (Facebook)*, *BigTable (Google)*, *Amazon Simple Storage Service (S3)*, *HBase (Apache)* und *CouchBase (ebenfalls Apache)* [8]. Diesen Dateisystemen ist gemeinsam, dass sie – genau wie Dynamo – Hochverfügbarkeit und Ausfallsicherheit durch Partitionierungs- und Replikationsstrategien erreichen. Dabei steht die Verwendung von kostengünstiger Hardware im Vordergrund. Die einzelnen Implementierungskonzepte sind sich zwar sehr ähnlich, jedoch waren die genannten Firmen aufgrund unterschiedlicher Anforderungen dazu gezwungen, eigene Systeme zu entwerfen. Während Amazon auf die Verwaltung feingranularer Daten mit sehr geringen Zugriffszeiten abzielt, möchte Google mit seinem *Google File System (GFS)* maximalen Durchsatz auf sehr große Daten (> 1 GB) erreichen [9]. Entstehende, höhere Latenzzeiten werden dabei in Kauf genommen. Anders als bei Dynamo erfolgt die Koordination mittels zentraler Master-Server. Diese speichern und verwalten die Metadaten der abgespeicherten Dateien, welche sich auf mehreren Tausend Chunkservern pro Master-Server befinden. Das Verwenden solcher Master-Server verletzt eigentlich die Bedingung der Symmetrie, was hinsichtlich der Skalierbarkeit ein Nachteil wäre. Für jeden Master-Server existieren jedoch einige Spiegelungen (sogenannte „Schatten-Server“), die stellvertretend für ihn einspringen.

Während bei Dynamo das Format der Daten für das System keine Rolle spielt, sind alle Daten bei *Cassandra* mit einem Schema abgespeichert. Die `get`-Operation benötigt als Parameter beispielsweise nicht nur einen Schlüsselnamen, sondern noch einen Tabellen- und einen Spaltennamen. Zudem möchten die Entwickler in einer zukünftigen Version die Kompression der Daten ermöglichen [10].

Mit *Project Voldemort* [11], *Dynomite* [12] und *Riak* [13] existieren Dynamo-Klone, die die wesentlichen Techniken wie Versionierung, Partitionierung und Replikation unterstützen. Diese Projekte sind kostenlos und sogar Open Source.

3. FAZIT

Einer der besten Beweise dafür, dass ein beliebiges System gut funktioniert, ist der jahrelange, produktive Einsatz ohne nennenswerte Vorkommnisse. Genau dies scheint bei Dynamo der Fall zu sein. Die Milliardenumsätze von Amazon [14] geben Rückschlüsse über die enormen Besucherzahlen, die täglich von den Servern der Firma zu bewerkstelligen sind. Nach Angaben der Entwickler ist das System seit 2005 im Einsatz und bis zur Herausgabe des Papiers im Jahre 2007 soll es zu keinem einzigen Datenverlust gekommen sein [1] – ist dies die Wahrheit, spricht das für eine sehr hohe Zuverlässigkeit.

Zwar bleiben dem Leser des Papiers einige Details wegen der Wahrung von Betriebsgeheimnissen verwehrt, so reicht der Einblick aber dennoch aus, um ein Verständnis dafür zu bekommen, auf was es bei der Umsetzung eines hochverfügbaren Dateisystems ankommt. Da Dynamo nur von Amazon selbst genutzt wird und keine Endkundenlösung bereitsteht, spielt es für den Einsatz in einer Public Cloud keine Rolle.

Allerdings werden für die Entwicklung von Dynamo zahlreiche Schlüsselkonzepte des Cloud Computings [15] angewendet. Die Partitionierung sorgt zum einen für eine ausgewogene Verteiltheit der Daten und zum anderen für hohe Skalierbarkeit. Dabei werden die Daten möglichst so verteilt, dass bei Ausfällen noch genügend Replikate verfügbar bleiben; die Ausfallsicherheit steigt. Solche Verfahren werden deshalb auch in den angesprochenen Open-Source-Projekten, die Dynamo als Vorbild haben, implementiert.

Amazon stellt mit S3 [16] einen Online-Speicherdienst zur Verfügung, der Dynamo in vielen Punkten sehr ähnlich ist. Dennoch kann dieser Dienst der Infrastructure-as-a-Service-Schicht als richtige Public Cloud angesehen werden: Der Kunde kann zu jeder Zeit und von überall, wo der Zugang zum Internet möglich ist, gegen Bezahlung Speicherplatz anmieten.

Komplexe Backup-Lösungen, wie zum Beispiel teure RAID-Bundles, gehören immer mehr der Vergangenheit an. Große Unternehmen – wie Google und Amazon – setzen schon heute auf die Verwendung von kostengünstigen Hardware-Komponenten, die im Zusammenspiel mit verteil-

ten Dateisystemen trotzdem für genügend Datenredundanz sorgen. Es ist zu erwarten, dass das Konzept solcher verteilter Dateisysteme auch in Zukunft immer wieder aufgegriffen wird und deren Bedeutung weiter wächst.

4. LITERATUR

[1] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principle*, pages 205–220, 2007.

[2] Tobias Distler and Timo Hönig. Ausgewählte Kapitel der Systemsoftware: Cloud Computing, 2010. http://www4.informatik.uni-erlangen.de/Lehre/SS10/HS_AKSS/slides/00_Einfuehrung.pdf (zuletzt besucht: 21.05.2010).

[3] Data Center Energy Management. Definition of Data Center. http://hightech.lbl.gov/dctraining/definitions.html (zuletzt besucht: 22.05.2010).

[4] Henry Robinson. Consistency and availability in Amazon's Dynamo, 2008. http://the-paper-trail.org/blog/?p=51 (zuletzt besucht: 25.05.2010).

[5] Rüdiger Kapitza and Wolfgang Schröder-Preikschat. Verteilte Systeme (VS) - Vorlesung (SS 2009), 2009. http://www4.informatik.uni-erlangen.de/Lehre/SS09/V_VS/Vorlesung/.

[6] Cassandra Wiki. AntiEntropy. http://wiki.apache.org/cassandra/AntiEntropy (zuletzt besucht: 24.05.2010).

[7] Werner Vogels. Eventually Consistent - Revisited, 2008. http://www.allthingsdistributed.com/2008/12/eventually_consistent.html (zuletzt besucht: 23.05.2010).

[8] Richard Jones. Anti-RDBMS: A list of distributed key-value stores, 2007. http://www.metabrew.com/article/anti-rdbms-a-list-of-distributed-key-value-stores/ (zuletzt besucht: 24.05.2010).

[9] Wikipedia. Google File System. http://de.wikipedia.org/wiki/Google_File_System (zuletzt besucht: 29.05.2010).

[10] Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. In *3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, 2009.

[11] Project Valdermont, 2010. http://project-voldemort.com/.

[12] Dynomite, 2010. http://github.com/cliffmoon/dynomite.

[13] Riak, 2010. http://riak.basho.com/.

[14] heise online. Amazon steigert Umsatz und Gewinn, 2008. http://www.heise.de/newsticker/meldung/Amazon-steigert-Umsatz-und-Gewinn-189717.html (zuletzt besucht: 24.05.2010).

[15] Wikipedia. Cloud Computing. http://en.wikipedia.org/wiki/Cloud_computing (zuletzt besucht: 29.05.2010).

[16] Amazon. Amazon Simple Storage Service (Amazon S3), 2010. http://aws.amazon.com/de/s3/.

[17] Alex Iskold. Amazon Dynamo: The Next Generation Of Virtual Distributed Storage, 2007. http://www.readwriteweb.com/archives/amazon_dynamo.php (zuletzt besucht: 22.05.2010).

[18] Wikipedia. Amazon Dynamo. http://de.wikipedia.org/wiki/Amazon_Dynamo (zuletzt besucht: 24.05.2010).

[19] Björn Patrick Swift. Amazon's Dynamo. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007.

[20] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[21] Simon Bin. Key Value Stores Dynamo und Cassandra, Seminararbeit Cloud Data Management. Master's thesis, Universität Leipzig, 2010. http://dbs.uni-leipzig.de/file/seminar0910_Bin_Ausarbeitung.pdf.

[22] David K Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162, 1979.

[23] Ken North. The NoSQL Alternative, 2010. http://www.informationweek.com/news/development/architecture-design/showArticle.jhtml?articleID=224900559 (zuletzt besucht: 24.05.2010).