Universität Leipzig
Institut für Informatik

Problemseminar NoSQL-Datenbanken

Key-Value-Stores

Amazon Dynamo

Anja Merzdorf

Januar 2012

Inhaltsverzeichnis

1 Einführung **3**

2 Hintergrund und Anforderungen **3**

3 Amazons Architektur **4**

4 Was ist Dynamo? **5**

5 Partitionierung und Replikation **6**

6 Versionierung **9**

7 Quorum-Eigenschaften **10**

8 Ausfälle **11**
8.1 Temporäre Ausfälle 11
8.2 Permanente Ausfälle 11

9 Mitgliedsfindung und Ausfallerkennung **12**

10 Optimierungen **13**
10.1 Verbesserte Partitionierungsstrategie 13
10.2 Pufferung 14
10.3 Ausgleich von Vorder- und Hintergrundaktivitäten 15

11 Zusammenfassung **15**

Literatur **17**

1 Einführung

In bestimmten Anwendungsbereichen, beispielsweise im Bereich der Webanwendungen bei Google oder Facebook, gelten veränderte Anforderungen an die Datenbanksysteme im Vergleich zu traditionellen relationalen Datenbanksystemen. Es müssen sehr große Datenmengen unter einfachen Schemata verarbeitet werden können und eine sehr gute horizontale Skalierbarkeit wird gefordert. Im Vergleich zu klassischen OLTP-Anwendungen wie z.B. im Finanzwesen, werden hier eher geringere Anforderungen bezüglich der Konsistenz gestellt. All diese Eigenschaften werden in NoSQL- Datenbanken vereint. [6]

Die einfachsten NoSQL- Datenbanken in Bezug auf ihre API sind die Key-Value-Stores [7]. Sie erlauben dem Anwendungsentwickler schemalose Daten zu speichern [5]. Diese Daten bestehen für gewöhnlich aus einem String, welcher den Schlüssel darstellt und der Information, die als Wert in der Schlüssel-Wert Beziehung fungiert [5]. Für detailliertere Informationen zu Key-Value-Stores verweise ich an dieser Stelle auf [9].

Ein Beispiel für einen Key-Value-Store ist die Datenbank Dynamo von Amazon. Diese wird im Folgenden genauer vorgestellt. Zu Beginn werden zunächst die Anforderungen an dieses System und die zugrundeliegende Struktur Amazons dargelegt. Im weiteren Verlauf werden verschiedene, in Dynamo angewendete Verfahren näher erläutert. Abschließend werden in dieser Arbeit Optimierungsmöglichkeiten für einzelne Verfahren betrachtet.

Die Ausarbeitung basiert hauptsächlich auf dem von Amazon veröffentlichten Paper zu Dynamo [1]. Daher sind alle Informationen, sofern sie nicht mit einer anderen Quellenangabe gekennzeichnet sind, auf diese Quelle zurückzuführen.

2 Hintergrund und Anforderungen

Amazon führt eine weltweite elektronische Handelsplattform, die in Spitzenzeiten zehn Millionen Kunden bereitgestellt wird. Dabei verwenden diese Zehntausende Server, welche auf viele Rechenzentren rund um den Globus verteilt sind. Um dieses System erfolgreich aufrecht zu erhalten, gibt es strenge betriebliche Anforderungen an Amazons Plattform bezüglich Performance, Ausfallsicherheit und Effizienz. Um fortlaufendes Wachstum zu unterstützen, muss die Plattform außerdem auch in einem hohen Maße skalierbar sein. Eine der wichtigsten Anforderungen ist allerdings die Ausfallsicherheit, da bereits kleinste Ausfälle zu gravierenden finanziellen Folgen führen können und das Vertrauen der Kunden beeinflussen.

Amazon verwendet eine hoch dezentralisierte, locker gekoppelte, serviceorientierte Architektur aus hunderten von Services. Eine solche Umgebung erfordert Speichertechnologien, die immer erreichbar sind, auch wenn beispielsweise Festplatten versagen, Störungen in Netzwerkverbindungen vorliegen oder Rechenzentren von Tornados zerstört werden.

Eine weitere wichtige Anforderung ist die zuverlässige Fehlerbehandlung, da in so einem Komplexen System aus Millionen von Komponenten zu jeder Zeit eine geringe aber

doch maßgebliche Anzahl an Ausfällen von Server- oder Netzwerkkomponenten auftreten wird. Diese Ausfälle sollen dabei ohne Beeinflussung von Erreichbarkeit und Performance behoben werden.

Ebenfalls wird gefordert, dass beim Hinzufügen neuer Rechnerknoten die Antwortzeiten gering gehalten werden(schrittweise Skalierbarkeit). Außerdem soll jeder Knoten die selben Aufgaben haben wie die ihm gleichgestellten und keiner soll eine spezielle Rolle einnehmen(Symmetrie). Als Erweiterung der Symmetrie, sollen dezentralisierte Peer-to-Peer Techniken gegenüber zentralisierter Kontrolle bevorzugt werden. Zudem muss das System in der Lage sein, die Heterogenität der Infrastruktur in der es läuft, auszunutzen, z.B. sollte die Arbeitsverteilung proportional zu den Leistungen des individuellen Servers sein.

3 Amazons Architektur

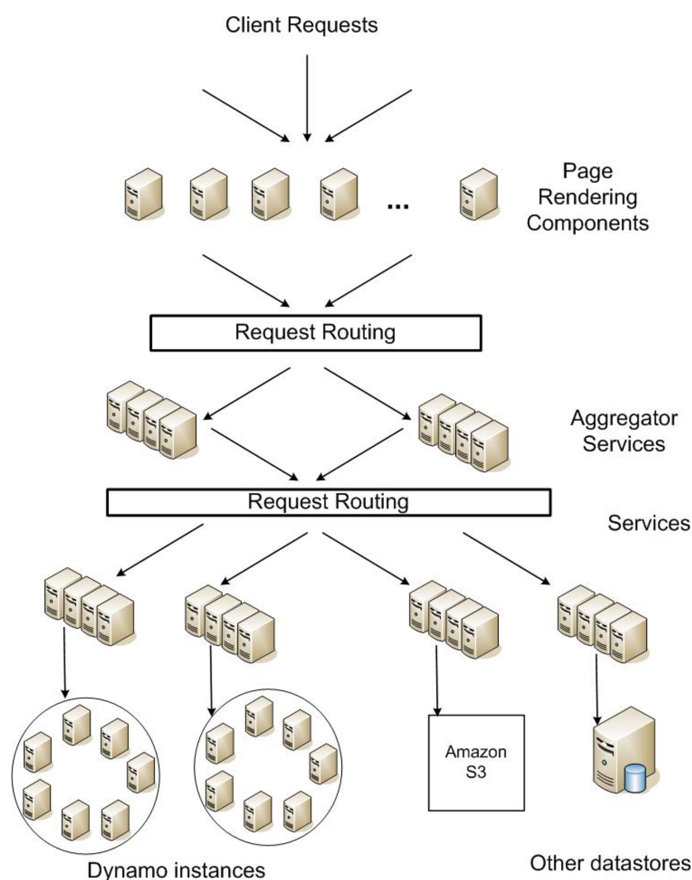

Abbildung 1: Amazons Architektur, [1]

In Abbildung 1 ist Amazons Architektur in abstrakter Sichtweise dargestellt. Es ist zu sehen, dass Client-Anfragen von Seiten-Erstellungs-Komponenten entgegengenommen werden. Diese generieren dynamischen Webinhalt und müssen dafür selbst wiederum nacheinander Anfragen an viele andere Services stellen. Ein Service kann zur Verwaltung seines Zustandes verschiedene Datenspeicher verwenden wie z.B. Instanzen von Dynamo oder die Datenbank Amazon S3, wobei diese nur innerhalb der jeweiligen Servicegrenzen zugänglich

sind. Manche Services fungieren als Aggregatoren indem sie verschiedene andere Services verwenden, um eine zusammengesetzte Antwort zu erzeugen.

Eine der hauptsächlichen Entwurfsüberlegungen für Dynamo ist daher, den Services die Kontrolle über ihre Systemeigenschaften wie beispielsweise Persistenz und Konsistenz zu geben und den Services ihre eigenen Kompromisse zwischen Funktionalität, Performance und Kosteneffizienz machen zu lassen.

4 Was ist Dynamo?

Dynamo ist ein hochverfügbares verteiltes Dateisystem, welches eine einfache Schlüssel-Wert-Paar Speichermethode verwendet. Diese eignet sich sehr gut, da die Absicht besteht nur relativ kleine Objekte zu speichern mit einer Größe < 1 MB. Dynamo erlaubt die weitere Ausführung von Lese- und Schreiboperationen sogar während Netzwerkpartitionierungen und löst Aktualisierungskonflikte unter Verwendung verschiedener Konfliktlösungsmechanismen. Dabei wird der Lösungsprozess von Aktualisierungskonflikten auf die Leseoperationen gelegt, um sicherzustellen, dass die Schreiboperationen nie zurückgewiesen werden. Die Ausführung des Konfliktlösungsprozesses kann sowohl vom Datenspeicher als auch von der Anwendung vorgenommen werden. Allerdings kann der Datenspeicher zur Lösung der Konflikte nur einfache Kontrollen nutzen, z.B. letzte Schreiboperation gewinnt, während sich die Anwendung bei Kenntnis des Datenschemas entscheiden kann, welches die beste Konfliktlösungsmethode für den Problemverlauf seines Clients ist.

Dynamo wurde für Amazon-interne Dienste entworfen, weshalb es keine Mechanismen zur Authentifizierung und Autorisierung gibt sondern vorausgesetzt wird, dass alle Knoten vertrauenswürdig sind. Zudem benötigen Anwendungen, die Dynamo verwenden, keine Unterstützung für hierarchische Namensräume oder komplexe relationale Schemata. Des weiteren ist Dynamo für Latenzzeit-empfindliche Anwendungen gebaut, welche die Ausführung von mindestens 99,9% der Lese- und Schreiboperationen innerhalb weniger Millisekunden verlangen. Zu Gunsten von hoher Verfügbarkeit und Partitionstoleranz garantiert Dynamo keine volle Konsistenz, sondern beschränkt sich auf „letztendliche“ Konsistenz, da nach dem CAP-Theorem ein verteiltes System maximal zwei der drei Eigenschaften gleichzeitig erfüllen kann [6].

In nachfolgender Tabelle sind die von Dynamo verwendeten Verfahren und ihre Vorteile dargestellt:

Problem	**Verfahren**	**Vorteil**
Partitionierung	Konsistentes Hashing	Schrittweise Skalierbarkeit
Hochverfügbarkeit für Schreibzugriffe	Vektoruhren mit Abgleich während Lesevorgängen	Zahl der Versionen ist nicht an Aktualisierungsrate gekoppelt
Behandlung temporärer Ausfälle	Sloppy Quorum und Hinted Handoff	Unterstützt Hochverfügbarkeit und garantiert Dauerhaftigkeit, auch wenn einige Replikate nicht verfügbar sind
Wiederherstellung nach dauerhaften Ausfällen	Anti-Entropie mit Merkle Bäumen	Synchronisierung abweichender Replikate im Hintergrund
Mitgliedsfindung und Ausfallerkennung	Gossip-basiertes Mitgliederprotokoll und Ausfallerkennung	Bewahrt Symmetrie und vermeidet zentrale Koordination

Tabelle 1: Verfahren und deren Vorteile, [1]

System Interface

Dynamo speichert Objekte in Verbindung mit einem Schlüssel und stellt hierfür die Operationen *get()* und *put()* bereit. Die *get(key)* Methode sucht die Objekt Replikationen verbunden mit dem Schlüssel im Speichersystem und gibt ein einzelnes Objekt oder eine Liste von Objekten mit widersprüchlichen Versionen zusammen mit einem Kontextes zurück. Die Operation *put(key, Context, object)* ermittelt, wo die Replikation des Objektes basierend auf dem zugehörigen Schlüssel platziert werden soll und schreibt die Replikation auf den Datenträger. Der Kontext kodiert System-Metadaten über das Objekt und enthält Informationen wie beispielsweise die Version des Objektes. Die Kontextinformation wird zusammen mit dem Objekt gespeichert, wodurch das System die Gültigkeit des Kontextobjektes überprüfen kann, welches in der aktuellen Anfrage geliefert wird. Dynamo betrachtet Schlüssel und Objekt, welche durch den Aufrufer geliefert werden, als Byte-Arrays. Ein MD5 Hash wird auf den Schlüssel gelegt, um eine 128-Bit Kennung zu generieren, welche verwendet wird, um die Speicherknoten ausfindig zu machen, die zuständig für die Lieferung der Schlüssel sind.

5 Partitionierung und Replikation

Um schrittweise Skalierung zu ermöglichen, ist ein Mechanismus zur dynamischen Partitionierung der Daten über die Knotenmenge erforderlich. Dynamos Partitionierungsschema beruht auf konsistentem Hashing, um die Last auf mehrere Speicher Hosts zu verteilen. Dabei wird der Wertebereich der Hash-Funktion als Ring dargestellt. Jedem Knoten im

System wird ein zufälliger Wert innerhalb dieses Bereiches zugewiesen, welcher seine Position auf dem Ring darstellt. Dabei ist jeder Knoten für den Bereich im Ring zwischen ihm und seinem Vorgängerknoten zuständig. Ein Dateneintrag wird durch einen Schlüssel, der einem Knoten zugeordnet ist, identifiziert. Durch Hashing wird die Position des Schlüssels auf dem Ring ermittelt und der erste Knoten mit einer höheren Position als dieser, der durch anschließendes Wandern auf dem Ring im Uhrzeigersinn gefunden wird, ist der zugehörige Knoten.

In Abbildung 2 befindet sich die Position des Schlüssels K im rot gekennzeichneten Bereich zwischen den Knoten A und B. Folglich ist der Schlüssel K dem Knoten B zugeordnet, da dieser der nächste Knoten auf dem Ring im Uhrzeigersinn ist.

Grundlegender Vorteil dieses Verfahrens ist, dass ein Hinzukommen oder Wegfall eines Knotens nur den unmittelbaren Nachbar betrifft und alle anderen Knoten unberührt bleiben. Ein Nachteil ist allerdings, dass die zufällige Bestimmung der Position jedes Knotens auf dem Ring zu ungleicher Lastverteilung führt. Des weiteren erkennt der Hauptalgorithmus die Unterschiedlichkeit der Performance der Knoten nicht.

Um Hochverfügbarkeit und Dauerhaftigkeit zu erreichen, repliziert Dynamo seine Daten auf mehrere Knoten. Jeder Dateneintrag wird dabei auf N Knoten repliziert, wobei der Parameter N für jeden spezifischen Fall angepasst wird.

Wie bereits erwähnt ist jeder Schlüssel K einem Koordinator-Knoten zugeordnet. Dieser ist verantwortlich für die Replikation der Dateneinträge, die in seinen Bereich fallen. Zusätzlich zur lokalen Speicherung jedes Schlüssels innerhalb seines Bereiches, repliziert der Koordinator diese auch auf die N-1 im Uhrzeigersinn nachfolgenden Knoten im Ring. Auf diese Weise entsteht ein System, in dem jeder Knoten für den Bereich zwischen ihm und seinem N-ten Vorgänger des Ringes zuständig ist.

Abbildung 2: Replikation, [4]

In der Beispielabbildung mit N = 3 repliziert Knoten B den Schlüssel K auf die Knoten C und D zusätzlich zur lokalen Speicherung. Knoten D wird alle Schlüssel speichern, die in die Bereiche (A,B], (B,C] und (C,D] fallen.

Wie oben bereits erwähnt, besteht ein Nachteil in der Zuordnung eines Knoten zu nur einer Position darin, dass dies zu ungleicher Lastverteilung führen kann. Wenn beispiels-

weise ein Knoten im Ring wegfällt, so wird dessen Wertebereich seinem Nachfolgeknoten zugeordnet. Zusätzlich zu seinem Bereich ist dieser fortan auch für den Bereich des weggefallenen Knotens zuständig und muss nun sozusagen eine größere Menge Last tragen. Die Replikation erfolgt für den hinzugekommenen Bereich ebenfalls auf die Replikationsknoten des Nachfolgers. [3]

Aus diesem Grund verwendet Dynamo das Konzept der „virtuellen Knoten“. Ein virtueller Knoten sieht wie ein einzelner Knoten im System aus, jedoch kann jeder Knoten auf mehrere virtuelle Knoten abgebildet werden, wodurch eine Zuordnung zu mehreren Punkten des Ringes möglich wird, statt nur zu einem.

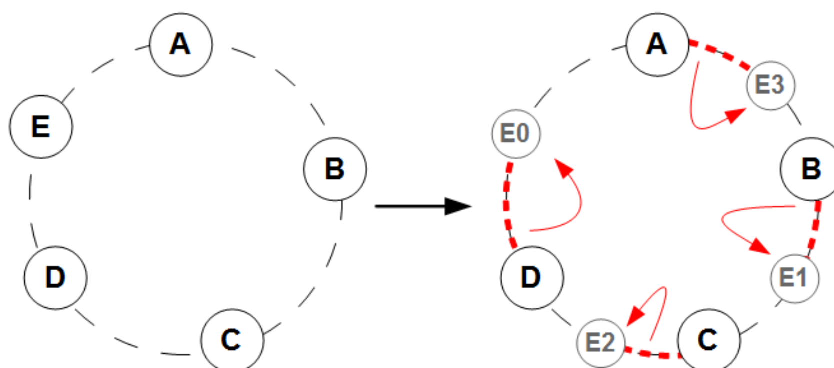

Abbildung 3: Aufteilung eines Knotens in virtuelle Knoten, [4]

Wie in Abbildung 3 zu sehen, wurde der ursprüngliche Knoten E auf die virtuellen Knoten E0 – E3 aufgeteilt. Der Vorteil hierbei ist, dass wenn ein so aufgeteilter Knoten wegfällt, sich seine Last gleichmäßig auf die übrigen Knoten verteilt. Hätte er im Vergleich dazu nur seine einzige Position (zwischen D und A) gehabt, dann wäre die Last auf A und durch die Replikationen auch auf B und C gefallen, jedoch wäre Knoten D unberücksichtigt geblieben [3]. Darüber hinaus hat die Verwendung von virtuellen Knoten auch den Vorteil, dass wenn ein Knoten wieder erreichbar wird oder ein neuer Knoten ins System eingefügt wird, er eine etwa gleich große Menge Last von jedem der anderen erreichbaren Knoten annimmt. Die Anzahl virtueller Knoten, für die ein Knoten zuständig ist, kann basierend auf dessen Kapazität bestimmt werden.

Die Liste der Knoten, die zuständig für die Speicherung eines einzelnen Schlüssels sind, wird Vorzugsliste genannt. Dabei kann jeder Knoten im System ermitteln, welche Knoten für einen speziellen Schlüssel in der Liste sein sollte. Um bei einen Knotenausfall trotzdem N Replikationen zu erreichen, enthält die Vorzugsliste mehr als N Knoten. Durch die Verwendung virtueller Knoten kann es passieren, dass die ersten N Nachfolgepositionen für einen einzelnen Schlüssel zu weniger als N unterschiedlichen physikalischen Knoten gehören. Virtuelle Knoten könnten nämlich auch direkt hintereinander liegen wodurch bei Replikation auf den Nachfolger die Replikate auf ein und den selben Knoten repliziert würden [3]. Käme es zum Ausfall dieses Knotens, würden alle Replikationen verloren gehen [3]. Darum werden bei der Erstellung der Vorzugsliste eines Schlüssels einzelne virtuellen Knoten im Ring übersprungen, um sicherzustellen, dass die Liste nur unterschiedliche

physikalische Knoten enthält. Darüber hinaus ist die Vorzugsliste so konstruiert, dass die Speicherknoten über mehrere Datencenter verteilt werden können wodurch auch jedes Objekt auf mehrere Datencenter repliziert wird [3]. Im Falle eines Ausfalles eines kompletten Datencenters besteht somit ein Schutz vor Datenverlust [3].

6 Versionierung

Dynamo bietet „letztendliche“ Konsistenz, welche es Aktualisierungen erlaubt sich asynchron auf allen Replikationen auszubreiten. Das Ergebnis jeder Modifikation wird als eine neue und unveränderliche Version der Daten betrachtet. In den meisten Fällen fassen neue Versionen vorhergehende Versionen zusammen und das System selbst kann die maßgebende Version bestimmen. Es kann allerdings auch passieren, dass Versionen eines Objektes sich in unterschiedliche Richtungen entwickeln durch Ausfälle in Kombination mit nebenläufigen Aktualisierungen. In diesen Fällen kann das System nicht mehrere Versionen des selben Objektes in Einklang bringen und der Client muss die Zusammenführung vornehmen, um die verschiedenen Zweige der Datenabstammung zurück auf einen zu bringen.

Um die Ursache zwischen verschiedenen Versionen des selben Objektes zu erfassen, verwendet Dynamo Vektoruhren. Eine Vektoruhr ist eine Liste mit (Knoten, Zähler) Paaren. Jede Version eines Objektes ist mit einer solchen Uhr verbunden. Durch prüfen ihrer Vektoruhren kann ermittelt werden, ob zwei Versionen eines Objektes auf parallelen Zweigen sind oder eine bestimmte Reihenfolge besitzen. Falls zum Beispiel die Zähler der ersten Uhr des Objektes kleiner oder gleich aller Knoten in der zweiten Uhr sind, dann ist die erste ein Vorfahre der zweiten und kann vernachlässigt werden. Anderenfalls werden die beiden Änderungen als widersprüchlich betrachtet und erfordern eine Auflösung.

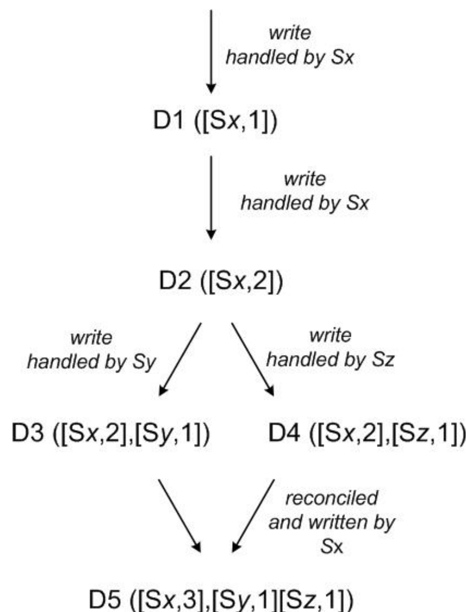

Abbildung 4: Beispiel einer Versionierung, [1]

In Abbildung 4 ist der Verlauf einer Versionierung eines Objektes dargestellt. Zuerst schreibt ein Client ein neues Objekt. Der Knoten Sx bearbeitet den Schreibvorgang für diesen Schlüssel, erhöht seine Satznummer und verwendet diese um die Vektoruhr der Daten zu erstellen. Das System hat nun D1 mit zugehöriger Uhr [(Sx,1)]. Im folgenden aktualisiert der Client das Objekt. Wenn der selbe Knoten auch diese Anfrage behandelt, bekommt das System D2 mit der zugehörigen Uhr [(Sx,2)]. Da D2 von D1 abstammt, wird D1 überschrieben. Der selbe Client aktualisiert das Objekt noch einmal und ein anderer Server (Sy) bearbeitet nun die Anfrage. Daraufhin besitzt das System jetzt die Daten D3 mit zugehöriger Uhr [(Sx,2),(Sy,1)]. Angenommen ein anderer Client liest D2 und versucht es zu aktualisieren und ein anderer Knoten (Sz) erledigt die Schreiboperationen, dann bekommt das System D4 von D2 abstammend mit Versionsuhr [(Sx,2),(Sz,1)]. Ein Knoten der D1 und D2 kennt, kann durch Erhalt von D4 und seiner Uhr ermitteln, dass D1 und D2 durch neue Daten überschrieben wurden und entsorgt werden können. Ein Knoten der D3 kennt und D4 empfängt, wird keinen kausalen Zusammenhang zwischen ihnen finden, da Veränderungen in D3 und D4 vorhanden sind, die sich nicht ineinander widerspiegeln. Daher müssen beide Versionen erhalten bleiben und einem Client zur semantischen Zusammenführung gegeben werden. Angenommen ein Client liest nun D3 und D4, dann entspricht der Lesekontext der Zusammenfassung der Uhren von D3 und D4 und zwar [(Sx,2),(Sy,1)(Sz,1)]. Wenn der Client die Zusammenführung durchführt und der Knoten Sx die Schreiboperation koordiniert, dann wird Sx seine Satznummer in der Uhr aktualisieren. Die neuen Daten D5 haben schließlich die Uhr: [(Sx,3),(Sy,1)(Sz,1)].

Die Größe der Vektoruhren kann wachsen, wenn viele Server die Schreiboperationen eines Objektes koordinieren. Um dieses Wachstum zu begrenzen, speichert Dynamo entlang jedes (Knoten, Zähler) Paares einen Zeitstempel, der den letzten Zeitpunkt anzeigt, zu dem der Knoten den Dateneintrag aktualisiert hat. Wenn die Anzahl an Paaren in der Uhr einen Grenzwert erreicht(z.B. 10), dann wird das älteste Paar entfernt.

7 Quorum-Eigenschaften

Um Konsistenz unter den Replikationen aufrecht zu erhalten, wurden neben dem Parameter N noch die Parameter R und W, ähnlich wie in Quorum Systemen, eingeführt. Dabei repräsentiert R die minimale Anzahl an Knoten, die bei einer erfolgreichen Leseoperation teilnehmen müssen und W stellt die minimale Anzahl an Knoten dar, die bei einer erfolgreichen Schreiboperation teilnehmen müssen. Die Werte von W und R beeinflussen somit die Erreichbarkeit, Dauerhaftigkeit und Konsistenz von Objekten. Werden R und W so gesetzt, dass $R + W > N$ ergibt sich ein Quorum ähnliches System. Da in diesem Fall aber die Latenzzeit durch den langsamsten der R oder W Replikationen bestimmt würde, sind R und W gewöhnlich so konfiguriert, dass sie $< N$ sind, um bessere Wartezeiten zu erreichen. An dieser Stelle sei noch gesagt, dass Dynamo keine strenge Quorum Mitgliedschaft durchführt und stattdessen „Sloppy Quorum“ verwendet, d.h. alle Lese- und Schreiboperationen werden auf den ersten N gesunden Knoten der Vorzugsliste durchgeführt, welche

nicht immer die ersten N Knoten sein müssen, die während der Bewegung durch den Ring angetroffen werden.

Ein Hauptvorteil von Dynamo ist hierbei, dass jede Client-Anwendung die Werte N, R und W verändern kann, um ihre angestrebten Level von Performance, Erreichbarkeit und Dauerhaftigkeit zu erreichen. Die Konfiguration (3, 1, 3) würde beispielsweise eine Art Lesepuffer realisieren, da nur ein Knoten für einen Lesezugriff antworten muss aber wegen N = W alle Kopien immer erfolgreich geschrieben werden müssen [2]. Wenn W = 1, dann wird das System nie eine Schreibanfrage ablehnen, solange mindestens ein Knoten im System ist, der die Schreibanfrage erfolgreich ausführen kann. Niedrige Werte von W und R können das Risiko von Inkonsistenz erhöhen, z. B. werden Schreibanfragen als erfolgreich gehalten und den Clients zurückgegeben, selbst wenn sie nicht von der Mehrheit der Replikationsknoten abgearbeitet werden. Konfigurationen, bei denen N = R = W realisieren ganz normale, nicht hochverfügbare Dateisysteme [2]. Die übliche Belegung (N, R, W), die von verschiedenen Dynamo-Instanzen verwendet wird, ist (3, 2, 2).

8 Ausfälle

8.1 Temporäre Ausfälle

Falls temporäre Knoten- oder Netzwerkausfälle auftreten, greift das sogenannte Hinted Handoff, um weiterhin erfolgreiche Lese- und Schreiboperationen zu garantieren. Wenn auf das Beispiel in Abbildung 2 bezogen mit N = 3 Knoten C ein Replikat erhalten soll, er aber zeitweise nicht verfügbar ist, dann würde die Kopie an Knoten A gesendet werden (*Handoff*). Zusätzlich wird vermerkt, dass diese Replikation eigentlich zu Knoten C gehört(*Hinted*). Knoten A speichert die Kopie in einer separaten lokalen Datenbank und fragt in periodischen Abständen nach, ob Knoten C wieder verfügbar ist. Wenn dies der Fall ist, werden alle *Hinted*-Replikate wieder an Knoten C zurückgegeben. [3]

8.2 Permanente Ausfälle

Es gibt Fälle, bei denen die *Hinted*-Replikationen unerreichbar werden, bevor sie zum originalen Replikationsknoten zurückgegeben werden können. Um damit und auch mit anderen Eigenschaften der Dauerhaftigkeit umzugehen, implementiert Dynamo ein Protokoll zur Synchronisation zwischen den Replikationen.

Um Inkonsistenzen zwischen Replikationen schneller aufspüren zu können und die Menge an Transferdaten zu minimalisieren, verwendet Dynamo Merkle Bäume. Ein Merkle Baum ist ein Hash-Baum, bei dem die Blätter Hash-Werte von individuellen Schlüsselwerten sind. Die Elternknoten weiter oben im Baum sind Hashes bezüglich ihrer Kinder. Der Hauptvorteil von Merkle Bäumen ist, dass jeder Zweig des Baumes unabhängig geprüft werden kann ohne den ganzen Baum oder die ganze Datenmenge herunterladen zu müssen. Merkle Bäume helfen bei der Reduzierung der Datenmenge, die während der Überprüfung, ob

inkonsistente Replikationen darunter sind, übergeben werden muss. Außerdem verringern sie die Anzahl der Lesevorgänge, die während des Synchronisationsprozesses durchgeführt werden müssen.

Für die Synchronisation zwischen den Replikationen finden Merkle Bäume folgendermaßen Verwendung: Jeder Knoten erhält einen separaten Merkle Baum für jeden Schlüsselbereich, den er verwaltet. Das erlaubt den Knoten zu vergleichen, ob die Schlüssel innerhalb eines Schlüsselbereiches aktuell sind. Hierfür erneuern zwei Knoten die Wurzel des Merkle Baumes entsprechend der Schlüsselbereiche die sie gemeinsam verwalten. Anschließend ermitteln die Knoten, ob sie irgendwelche Unterschiede haben unter Verwendung eines bestimmten Baum-Durchquerungs-Schemas und führen die zugehörige Synchronisations-Aktion durch. Der Nachteil hierbei ist, dass sich durch Hinzufügen oder Entfernen eines Knotens viele Schlüsselbereiche ändern, wodurch eine Neuberechnung der Bäume notwendig wird.

9 Mitgliedsfindung und Ausfallerkennung

Für temporäre Ausfalle von Knoten greift Hinted Handoff, um nicht die komplette Kreisstruktur neu aufbauen zu müssen. Andererseits besteht auch die Möglichkeit Knoten dauerhaft aus dem System zu entfernen oder hinzuzufügen. Dafür verbindet sich der Administrator per Komandozeilentool oder Browser mit einem Dynamo Knoten und stellt eine Mitgliedschaftsänderung aus, um einen Knoten dem Ring hinzuzufügen oder zu entfernen. Der Knoten, der die Anfrage bedient, schreibt die Mitgliedsänderung und deren Ausstellungszeit auf einen persistenten Speicher. Die Mitgliedschaftsänderungen bilden eine Historie, da Knoten mehrfach entfernt und wieder hinzugefügt werden können. Ein Gossip-basiertes Protokoll verbreitet die Mitgliedschaftsänderung und erhält eine „letztendlich“ konsistente Sicht auf die Mitgliedschaft. Jeder Knoten kontaktiert jede Sekunde einen zufällig ausgewählten gleichgestellten Knoten und beide Knoten tauschen ihre persistenten Mitglieds-Historien aus.

Wenn ein Knoten das erste mal startet, wählt er sich seine Menge an Tokens (virtuelle Knoten im konsistenten Hash Raum) und bildet die Knoten auf ihrer zugehörigen Tokenmenge ab. Die Abbildung bleibt dauerhaft auf einem Datenträger und enthält zu Beginn nur die lokalen Knoten und Tokenmengen. Die in verschiedenen Dynamo Knoten gespeicherten Abbildungen werden durch die selbe Kommunikationsart mitgeteilt wie die Mitglieds-Historie. Aufteilungs- und Platzierungsinformation werden folglich ebenfalls über das Gossip-basierte Protokoll verbreitet und jeder Speicherknoten hat Kenntnis von den Token-Bereichen, die von gleichrangigen Knoten bearbeitet werden. Das ermöglicht jedem Knoten eine Lese-/Schreiboperation für einen Schlüssel direkt zur richtigen Knotenmenge zu senden.

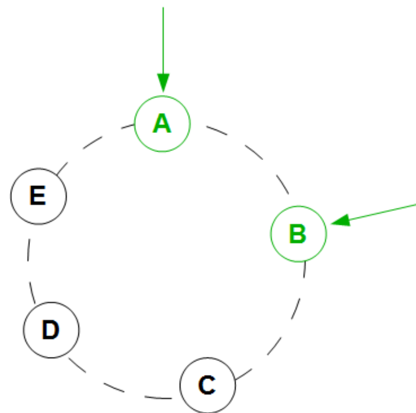

Abbildung 5: Nahezu zeitgleiches Hinzufügen zweier Knoten

Dieses Verfahren kann allerdings zeitweise zu einem logisch partitionierten Ring führen. Wenn beispielsweise wie in Abbildung 5 die Knoten A und B nahezu zeitgleich hinzugefügt werden, würden beide zunächst nichts voneinander wissen. Angenommen es erfolgt nun eine put()-Operation, die von A koordiniert werden soll. Dann wäre es wünschenswert, dass auf B repliziert wird. Da A aber B noch nicht kennt, würde die Replikation auf einem anderen Knoten stattfinden.[3]

Um das zu verhindern, spielen manche Dynamo Knoten die Rolle von „Seeds“. Seeds sind voll funktionale Knoten in Dynamo Ringen, die bei einem Erkennungsdienst registriert und allen anderen Knoten bekannt sind. Der Administrator verbindet sich nur mit dem Seed um einen Knoten zu entfernen oder hinzuzufügen und alle anderen Knoten tauschen ihre Mitgliedsliste mit dem Seed aus. Dadurch wird eine logische Aufteilung sehr unwahrscheinlich. Seeds können entweder durch statische Konfiguration oder durch einen Konfgurationsservice erreicht werden.

Die Ausfallerkennung geschieht während der Kommunikation. Wenn ein Knoten auf die Nachricht eines anderen Knotens nicht antwortet, so geht der anfragende Knoten davon aus, dass der kontaktierte Knoten ausgefallen ist. Daraufhin weicht der anfragende Knoten auf einen anderen Knoten aus, um seine Anfrage beantworten zu lassen. Zusätzlich prüft er in periodischen Abständen nach, ob der zuerst kontaktierte Knoten wieder erreichbar ist.

10 Optimierungen

10.1 Verbesserte Partitionierungsstrategie

Ein Nachteil der oben beschriebenen Partitionierungmethode liegt darin, dass die Partitionierung mit der Positionierung gekoppelt ist. Die folgende Methode löst diese Kopplung auf, in dem der Hash-Bereich im Vorfeld in Q gleich große Partitionen geteilt wird. Jeder Knoten erhält Q/S virtuelle Knoten, wobei S die Anzahl der physikalischen Knoten im

System ist. Wenn ein Knoten das System verlässt, werden seine virtuellen Knoten zufällig an die übrigen Knoten verteilt, so dass die Eigenschaft von Q/S virtuellen Knoten für jeden physikalischen Knoten erhalten bleibt. Wenn ein Knoten dem System beitritt, stiehlt er unter Einhaltung der Eigenschaft virtuelle Knoten von den anderen Knoten.

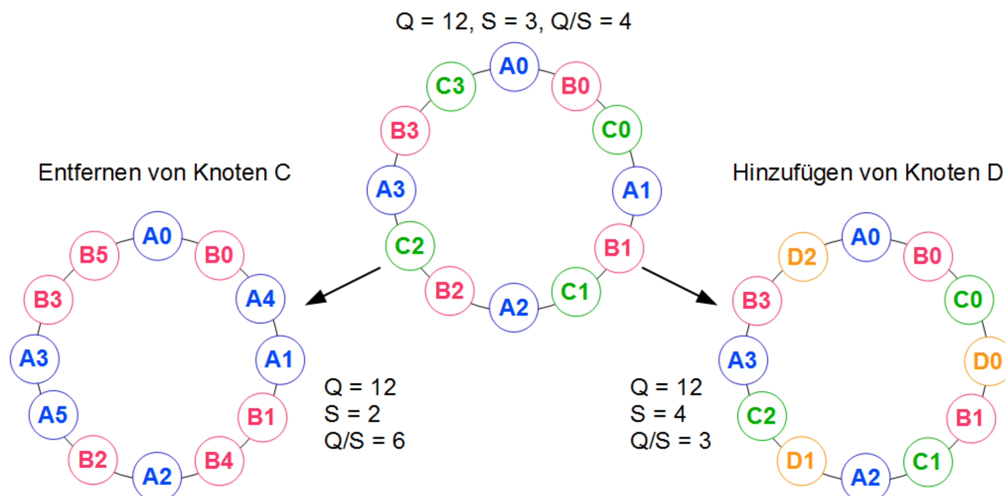

Abbildung 6: Optimierte Partitionierungsstrategie

In der Abbildung 6 ist ein Beispiel dessen dargestellt. Q wurde auf zwölf festgesetzt und es existieren zu Beginn drei physikalische Knoten A, B und C, wodurch jeder Knoten vier virtuelle Knoten zugeordnet bekommt. Wird nun Knoten C entfernt (linker Pfad), sind nur noch zwei physikalische Knoten vorhanden und die virtuellen Knoten von C werden unter ihnen so aufgeteilt, dass jeder der beiden verbleibenden Knoten am Ende sechs virtuelle Knoten besitzt um die Eigenschaft einzuhalten. Wird andererseits zu der ursprünglichen Verteilung ein Knoten D hinzugefügt (rechter Pfad), gibt es jetzt vier physikalische Knoten und jeder von ihnen darf unter Einhaltung der Eigenschaft drei virtuelle Knoten besitzen. Aus diesem Grund bekommt Knoten D drei virtuelle Knoten, die von den anderen Knoten weggenommen werden, so dass diese am Ende ebenfalls jeweils noch drei virtuelle Knoten besitzen.

Der Vorzüge dieser Strategie liegen in einer guten Effizienz und der Reduzierung der Größe der Mitgliedschaftsinformationen, die von jedem Knoten geführt werden. Ein weiterer Vorteil ist die Vereinfachung der Prozesse von Bootstrapping und Wiederherstellung. Ebenfalls wird eine einfachere Archivierung ermöglicht, da die Partitionierungsdateien separat archiviert werden können. Der einzige Nachteil besteht in der Änderung der Knotenmitgliedschaft. Hierfür wird eine Koordination benötigt, damit die Eigenschaften der Belegung eingehalten werden.

10.2 Pufferung

Einige Kundenoberflächendienste erfordern höhere Performance Level. Für diese stellt Dynamo die Möglichkeit bereit, eine Pufferung hin zuzuschalten. Dadurch wird Dauerhaftigkeit zu Gunsten einer höheren Performance eingetauscht.

In der Optimierung führt jeder Speicherknoten einen Objektpuffer in seinem Hauptspeicher. Jede Schreiboperation wird zunächst in dem Puffer gespeichert bevor sie später in periodischen Schritten durch einen Schreib-Thread auf den Speicher geschrieben wird. Bei einem erneuten Lesezugriff wird dann erst im Puffer nach dem Anfrage-Schlüssel gesucht. Wenn dieser dort gefunden wird, wird das Objekt direkt vom Puffer gelesen anstatt vom Speicher.

Die Abarbeitung der Leseanfragen wird somit beschleunigt. Durch den Austausch der Dauerhaftigkeit kann es jedoch bei Serverausfällen dazu kommen, dass die noch nicht abgearbeiteten Schreiboperationen im Puffer verloren gehen.

10.3 Ausgleich von Vorder- und Hintergrundaktivitäten

In der Regel führt jeder Knoten verschiedene Arten von Hintergrundprozessen z.B. Synchronisation von Replikaten durch. Diese bremsen dabei seine normalen put/get Operationen im Vordergrund aus. Deshalb ist es notwendig sicherzustellen, dass Hintergrundprozesse nur laufen, wenn normale Operationen nicht signifikant beeinflusst werden.

Zu diesem Zweck werden die Hintergrundprozesse mit einem Zugangskontrollmechanismus versehen. Dieser Admission Controller überwacht konstant das Verhalten der Quellzugriffe während er im Vordergrund eine put/get Operation ausführt. Überwachte Aspekte sind dabei Wartezeiten für Speicher Operationen, fehlgeschlagene DB-Zugriffe auf Grund von Sperrkonflikten oder Timeouts während der Transaktion, und Wartezeiten der Anfragewarteschlange. Je nach Bedarf gewährt der Admission Controller den Hintergrundprozessen Zeitschlitze. Dabei richtet sich die Anzahl der verfügbaren Zeitschlitze nach dem Grad der Auslastung. Somit haben die Prozesse im Vordergrund immer Vorrang und können schneller ausgeführt werden.

11 Zusammenfassung

Dynamo ist ein gutes Beispiel dafür, dass Speichersysteme mit „letztendlicher“ Konsistenz eine Basiskomponente für hochverfügbare Anwendungen sein können. Dabei verwendet es ein einfaches Key-Value Interface zur Speicherung und Abfrage der Daten, da viele Dienste nur eine Primärschlüssel-Verbindung zum Datenspeicher benötigen. Aufgrund dessen werden keine relationalen Schemata in Anwendung gebracht, was zugleich schnellere Datenbankabfragen begünstigt, um den Nutzern innerhalb von Sekundenbruchteilen das Ergebnis einer Produkt-Suchanfrage zu präsentieren. Durch das von Dynamo angewendete Verfahren der Partitionierung und Replikation wird eine hohe Skalierbarkeit gewährleistet und somit das stetige Wachstum der Plattform Amazons unterstützt. Auch die Ausfallsicherheit gehört zu den Kriterien, die von Dynamo erfüllt werden. Die Designwahl eines verteilten Dateisystems ermöglicht die Speicherung der Daten an verschiedenen Orten und durch mehrfache Replikation aller Daten besteht ein sehr guter Schutz vor Datenverlusten.

Dynamo erfüllt alle gestellten Anforderungen und ist somit ein wichtiger Bestandteil

in Amazons Architektur. Nicht zuletzt sprechen auch die sehr hohen und immer noch steigenden Besucherzahlen, welche zeitweilig bereits die von Ebay überstiegen, für ein sehr gut funktionierendes System mit geringstmöglicher Ausfallquote [8].

Literatur

[1] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, 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

[2] *Amazon Dynamo*, http://de.wikipedia.org/wiki/Amazon_Dynamo, zuletzt besucht: 12.11.2011

[3] Christopher Eibel: *Skalierbarer und zuverlässiger Zugriff auf feingranulare Daten*, Ausarbeitung, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2010, http://www4.informatik.uni-erlangen.de/Lehre/SS10/HS_AKSS/papers/03_Ausarbeitung_Christopher_Eibel.pdf, zuletzt besucht: 13.11.2011

[4] Christopher Eibel: *Skalierbarer und zuverlässiger Zugriff auf feingranulare Daten*, Folien, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2010, http://www4.informatik.uni-erlangen.de/Lehre/SS10/HS_AKSS/slides/03_Vortrag_Christopher_Eibel.pdf, zuletzt besucht: 13.11.2011

[5] 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, zuletzt besucht: 21.11.2011

[6] Prof. Dr. Uta Störl: *NoSQL-Datenbanken*, Folien, Hochschule Darmstadt, 2011, https://www.fbi.h-da.de/fileadmin/personal/u.stoerl/DBAkt-SS11/Vorlesung/DBAkt-SS11-NoSQL-Kap9.pdf, zuletzt besucht: 13.11.2011

[7] Ayende Rahien: *That No SQL Thing - Key/Value stores*, http://ayende.com/blog/4449/that-no-sql-thing-key-value-stores, zuletzt besucht: 13.11.2011

[8] Axel Gronen: *Die Reichweite von Onlinemarktplätzen im Vergleich*, http://www.wortfilter.de/news11Q1/news3958.html, zuletzt besucht: 13.11.2011

[9] Natanael Arndt: *Key-Value-Stores am Beispiel von Scalaris*, Seminararbeit NoSQL-Datenbanken, Universität Leipzig, 2012