

PROJEKTBERICHT

HAMAUBE

KAI MARTINEN, FABIAN KOHLER, ...

AUGUST 30, 2018



UNIVERSITÄT HAMBURG

DEPARTMENT OF COMPUTER SCIENCE

CHAIR OF DISTRIBUTED SYSTEMS AND INFORMATION
SYSTEMS

BETREUT DURCH STEFFEN FRIEDRICH

Abstract

Abstract in English

Kurzfassung

Kurzfassung auf Deutsch

Contents

Abstract	I
Kurzfassung	II
List of Tables	VI
1 Introduction	1
1.1 Initial goal and contributions	1
1.2 Thesis outline	1
2 Architektur	2
2.1 Einführung	2
2.2 Entwicklung der neuen Architektur	3
2.2.1 Analyse der Verteilung	4
2.2.2 Die Verteilung der Datenbank	5
2.2.3 Ein erste Blick auf die neue Architektur	6
2.2.4 Benötigte Komponenten	7
2.2.5 Datenfluss: Stromdatenverarbeitung	9
2.2.6 Microservice-Architektur	10
2.2.7 Die Publish-Subscribe-Architektur	11
2.2.8 Erweiterbarkeit des Systems	12
2.2.9 Denken in Ereignissen	13
2.2.10 Ausfallsicherheit	13
2.2.11 Grenzen der Architektur	14

3	Cassandra	15
3.1	Datenverwaltung	15
3.1.1	Datenschema	15
3.2	Cassandrareader	16
3.2.1	Architektur	17
3.3	Use Cases	18
3.3.1	Registrierung von Usern	19
3.3.2	Anmelden von Usern	19
3.3.3	Abfragen von Tweets	19
3.3.4	Abfragen von Usern	19
3.3.5	Abrufen der Timeline	19
3.3.6	Absenden/Speichern von Tweets	19
3.4	Webserver und Frontend	19
3.4.1	Frontend	19
3.4.2	Webserver	20
4	Umsetzung von Kommunikationsschemen über Kafka	22
4.1	Grundlagen Kafka	22
4.1.1	Registrieren eines Consumers	22
4.1.2	Nachrichtenverteilung	23
4.2	Realisierung der Kommunikationsschemen	24
4.2.1	Beliebiger Kommunikationspartner	24
4.2.2	Ausgewählter Kommunikationspartner	24
5	Conclusion	27
5.1	Future work	27
A	Glossary	29
B	Cassandra	34
B.1	Daten Model	34
B.2	System	35
B.2.1	Partitionierung	35
B.2.2	Replikation	36
B.2.3	Persistenz	37
B.3	CQL	38

C	Appendix C	39
C.1	Einführung	39
C.2	Datenmodell	39
C.3	Refinements	43
C.4	Performance Auswertung	44
D	Appendix C	45
D.1	Section 1	45
D.2	Section 2	45
E	Appendix C	46
E.1	Section 1	46
E.2	Section 2	46
F	Appendix C	47
F.1	Section 1	47
F.2	Section 2	47
G	Appendix D	48
G.1	Section 1	48

List of Figures

3.1	Cassandra Schema	16
3.2	Technologie Stack des cassandrareaders	17
3.3	Technologie Stack des cassandrareaders	18
4.1	Beispiel für die Zuordnung von <i>Consumern</i> zu <i>Consumer Groups</i> und <i>Topics</i>	23
4.2	Beispiel für die Umverteilung der Partitionen mit dem „roundrobin“ Verfahren: Zwischen Zustand 1 und Zustand 2 wurde der Consumer 2 entfernt.	25
B.1	Beispiel Daten Modell	35
B.2	Consistent-Hashing Ring	36
B.3	CQL Mapping	38
C.1	Tabellen Beispiel	40
C.2	Zugriffs Beispiel mit der BigTable API	41
C.3	Percormance Übersicht	44

List of Tables

Chapter 1

Introduction

Introduction.

You can reference the only entry in the .bib file like this: [?]

1.1 Initial goal and contributions

1.2 Thesis outline

Chapter 2

Architektur

2.1 Einführung

Ziel des Projekt ist es einen Twitter-Klon zu entwickeln. Als Mikroblogging-Dienst ist die zentrale Funktionalität:

- Senden von Tweets
- Finden von Tweets (Anzeigen der Tweets in der eigenen Timeline, Suchen nach Schlüsselwörtern/Hashtags)

Zusätzlich ist selbstverständlich eine Benutzerverwaltung nötig. Ebenso interessant sind Analytics-Funktionalitäten.

In diesem Projekt liegt der Fokus auf der Entwicklung einer Architektur, die besondere Nicht-Funktionale Anforderungen erfüllt. Diese ergeben sich aus den Anforderungen an das reale Twitter:

- Skalierbarkeit
- Erweiterbarkeit um zusätzliche Funktionalität.

Geprägt ist Twitter durch enorme Datengeschwindigkeit (Velocity in Gartners Vs). Jede Sekunde werden im Durchschnitt 6000 Tweets weltweit abgesetzt. Diese Zahlen variieren zudem stark: Der Rekord liegt bei 140.000 Tweets pro Sekunde. Zudem werden viele Features von Twitter nach und nach eingeführt, ohne das Twitter in Wartungspausen geht (gehen musste).

Diesen Herausforderungen sind herkömmliche Webarchitekturen nicht gewachsen. Dieses Projekt entwickelt einen Twitter-Clon mit den Technologien und einer Architektur, die diesen Anforderungen im Prinzip gewachsen ist.

Quellen

<https://www.brandwatch.com/de/blog/twitter-statistiken/> <http://www.gegen-den-strom.org/twitter/>

2.2 Entwicklung der neuen Architektur

Ausgangspunkt der Entwicklung der neuen Architektur ist eine einfache Client-Server-Architektur. Es gibt in ihr einen Server mit viel Rechenleistung sowie viele Clients. Die Clients verbinden sich mit dem Server, der sie parallel bedient. Die Parallelisierung erfolgt also auf Thread-/Prozessebene.

Der nächste Schritt ist es, die Datenhaltung in eine Datenbank auszulagern. Es gibt jetzt drei Komponenten: die Clients, der Webserver und der Datenbankserver. Wenn die Leistung des Serversystems nicht ausreicht gibt es unterschiedliche Möglichkeiten für mehr Leistung zu sorgen:

- Die einzelnen Server mit mehr Leistung auszustatten
- Mehr Webserver einzuführen

Möglichkeit 1 würde ein Aufrüsten der Server mit schnelleren Prozessoren, mehr Arbeitsspeicher, schnelleren Festplatten etc. bedeuten. Möglichkeit 2 ist schon etwas komplizierter. Zwar werden die Daten zentral vom Datenbankserver gehalten, trotzdem können nicht einfach mehr Webserver eingeführt werden. Die Clients müssen noch auf die Webserver verteilt werden.

Diese Aufgabe kann wieder auf zwei Arten erfolgen: Im Netzwerkprotokoll (z.B: über Anycast) oder durch eine zusätzliche Komponente im Server, nämlich dem Load-Balancer. An ihn kommen alle Anfragen an die Webserver und der Loadbalancer entscheidet mit mehr oder weniger komplexe Techniken welcher Webserver genutzt werden soll und leitet den anfragenden Client an diesen Webserver weiter. Dabei achtet diese Komponente z.B. darauf, dass bereits stark ausgelastete Webserver keine neuen Clients mehr

zugewiesen bekommen, sondern nur Webserver mit aktuell niedriger Auslastung.

Die Parallelisierung erfolgt in dieser Architektur bereits auf vielen Ebenen: Im Loadbalancer, in den Webservern und in der Datenbank. Die Datenbank führt dafür ausgefeilte Mechanismen zur Verwaltung der parallelen Anfragen und zur Datenkonsistenz aus.

Diese Architektur wird häufig mit dem LAMP Softwarestack implementiert: Linux als Betriebssystem der Server, Apache als Webserver, MySQL als Datenbank und PHP als Serverprogrammiersprache.

So weit eine erste Einführung in die Verteilung von Webanwendungen mit herkömmlichen Architekturen. Nun wollen wir die Art der Verteilung analysieren um darzustellen, weshalb eine andere Architektur für noch mehr Verteilung nötig ist.

2.2.1 Analyse der Verteilung

Es kommen zwei Techniken der Verteilung zu Einsatz

- Erhöhung der Rechenleistung der einzelnen Komponenten
- Loadbalancing

Die erste Technik wird auch Scale-Up genannt. Durch potentere Hardware kann ein Leistungszuwachs erzielt werden. Allerdings - und das ist ein zentrales Problem, dass unsere neue Architektur lösen wird - sind dieser Möglichkeit vergleichsweise enge Grenzen gesetzt. Der Prozessorgeschwindigkeit sind durch die Eigenschaften des verwendeten Silizium und der Wärmeentwicklung enge Grenzen gesetzt. Über 5 GHz gehen die Taktraten nicht mehr hinaus. Damit ist bereits bei einem Faktor von ca. 2,5 der Geschwindigkeitserhöhung hier ein Ende gesetzt. Von 2 GHz geht es bis maximal 5 GHz hoch (und selbst 5 GHz erreichen die wenigsten Prozessoren). Ein Leistungszuwachs kann jetzt nur noch durch mehr Prozessorkerne und mehr Prozessoren erreicht werden. Durch mehrere Sockel in den Server (leicht bis zu 4 Stück) und mehr Kernen in den Prozessoren (leicht bis zu 20 Stück) lässt sich hier bei sehr guter Parallelisierbarkeit der Anwendung¹ theoretisch eine Erhöhung um das 80-fache erreichen.

¹Amdahl's Law zeigt, dass sich durch mehr Hardware die Geschwindigkeit nicht linear erhöht

Hier ist nun die Grenze für das Scale-Up erreicht. Herkömmliche Server lassen sich nun kaum mehr beschleunigen. Sobald also eine Komponente in unserer Architektur mehr Leistung benötigt weil mehr Client-Anfragen kommen, als sie bearbeiten kann, müssen wir etwas in der Architektur ändern. Die Ausnahme sind hier die Webserver. In unserer Architektur können wir bereits mehr Webserver hinzufügen und die Last auf mehr Rechner verteilen, anstatt auf schnellere Rechner. Das ist ein Grundgedanke unserer neuen Architektur. Im Gegensatz zum Scale-Up wird dieser Ansatz Scale-Out genannt.

Die Komponente, die dementsprechend als erstes in unserer Architektur geändert werden muss, ist die Datenbank. Damit beschäftigen wir uns im nächsten Abschnitt.

2.2.2 Die Verteilung der Datenbank

Wir müssen also unsere Datenbank beschleunigen. Wir nehmen an, dass wir kein Scale-Up mehr durchführen können oder wollen.

Eine Möglichkeit wäre noch die Nutzung von sehr spezialisierter Datenbankhardware. Diese hat sehr große Investitionskosten und legt einen auf eine spezielle Datenbanksoftware fest. Wir möchten diese deshalb nicht betrachten.

Ein erster Ansatz wäre es, das Load-Balancing der Webserver zu wiederholen. Mehr Datenbankserver und eine Komponente, die die Anfragen an einen Datenbankserver mit genug verfügbarer Kapazität weiterleitet. Allerdings geht das nicht mehr so einfach wie bei den Webservern. Die Webserver waren untereinander unabhängig. Die Datenbank ist das nicht. Man kann nicht einfach einen Datenbankserver nehmen und dort Daten ändern. Wird der Nutzer nun bei seiner nächsten Anfrage an einen anderen Datenbankserver geleitet, sind die Änderungen dort nicht vorgenommen. Das ist also keine Alternative.

Aber diesen Ansatz kann man noch verfeinern und schlussendlich gewinnbringend einsetzen. Das Problem war, dass die geänderten Daten nur auf einem Server verändert wurden und der User plötzlich auf einen anderen Datenbankserver landete. Dann muss der Nutzer also immer auf dem gleichen Datenbankserver landen. Eine einfache Möglichkeit der Verteilung wäre es also, die Nutzer fest auf die Datenbankserver zu verteilen. Beispielsweise

werden alle Nutzer mit Nachnamen, die mit A-K beginnen auf den ersten Datenbankserver zugewiesen, die anderen auf den zweiten Datenbankserver. In Verwaltungseinrichtungen wird so etwas sogar physisch betrieben. Wenn der Nutzer nun ein zweites Mal kommt, wird er wieder an den gleichen Datenbankserver weitergeleitet und seine Daten sind dort aktuell. Diese Aufteilung muss dann aber in der Applikationslogik fest eingebaut werden. Besser wäre es, wenn das automatisch passieren würde.

Der andere Ansatz wäre es nicht den Benutzer immer auf den gleichen Datenbankserver weiterzuleiten, sondern die Datenbankserver alle aktuell zu halten. Dann wäre es wieder egal, welchen Datenbankserver er beim nächsten Besuch nutzt. Allerdings sieht man sofort, dass dann sehr viel Netzwerkverkehr nötig wäre um die Daten auf allen Server aktuell zu halten. Es ist also zweifelhaft, ob das so viel schneller wäre.

Das wiederum hängt vom Nutzungsprofil ab. Wenn Daten z.B. nur selten geändert werden, aber häufig gelesen, dann ergibt das wieder Sinn. Der Overhead für das Aktualhalten der Daten wäre dann sehr gering und der Leseoperationen würden skalieren.

2.2.3 Ein erste Blick auf die neue Architektur

Im vorigen Abschnitt haben wir gesehen, dass es nötig wird die Datenbank verteilt auszulegen. Die eine Möglichkeit war die händische Aufteilung der Anfragen auf verschiedene, unabhängige Datenbankserver. Diese Möglichkeit ist aufwendig. Eine andere Möglichkeit war das Spiegeln auf mehrere Datenbankserver. Das funktioniert nur dann, wenn die Leseoperationen die Schreiboperationen weit übertreffen und die Schreiboperationen nur eine Last erzeugen, die ein einzelner Datenbankserver bearbeiten kann.

Nun betrachten wir eine dritte Möglichkeit: die Verwendung verteilter Datenbanken. Damit werden Datenbankprogramme bezeichnet, die für den Einsatz auf mehreren Servern gebaut wurden. Sie sind eine Spielart der so genannten NoSQL-Datenbanken. Zur Skalierung verwenden sie den Scale-Out-Ansatz statt des Scale-Up-Ansatzes. D.h. sie sind so designed, dass man mehr Leistungsfähigkeit durch das Einbinden weiterer Server erreichen kann. Diese Server bestehen aus herkömmlicher Serverhardware. Es ist keine spezialisierte Hardware erforderlich. Die Koordinierung erfolgt über Netzwerkprotokolle. Damit wandert die Skalierbarkeit in die Software. Die

Realisierung wird weiter unten in einem gesonderten Kapitel beschrieben.

In der herkömmlichen Architektur wird nun die SQL-Datenbank durch eine verteilte Datenbank ersetzt. Jetzt lässt sich die benötigte Leistung durch die Verteilung der Datenbank auf genügend Server erreichen. Diese Komponente ist zusätzlich noch skalierbar, d.h. wenn mehr Leistung benötigt wird, können weitere Server hinzugefügt werden.

Das ist ein Prinzip unserer neuen Architektur: Nicht skalierbare Softwarekomponenten werden durch skalierbare Softwarekomponenten ausgetauscht. Wir haben mit der Datenbank begonnen. Jetzt ist es Zeit die benötigten Komponenten für unseren Twitter-Klon zu ermitteln. Im nächsten Schritt setzen wir für die Komponenten Software ein, die skalierbar ist.

2.2.4 Benötigte Komponenten

Unser Twitter-Klon soll die folgende Funktionalität haben:

- Nutzerverwaltung: Benutzerkonto mit Login-Funktionalität
- Kernfunktionalität: Tweets senden und lesen
- Tweets entdecken: Tweets suchen und Empfehlungen bekommen
- Statistische Auswertungen der Tweets

Es gibt offensichtlich eine Frontend-Komponente. Das Backend besteht nun aus verschiedenen Komponenten. Eine Komponente für die Nutzerverwaltung und eine Komponente für die Kernfunktionalität. Sie greifen auf die Daten in der verteilten Datenbank zu. Da ein eindeutiger Schlüssel bekannt ist, unterscheidet sich der Zugriff nur in Details von der Nutzung einer normalen SQL-Datenbank. Die verteilte Datenbank ist in der Lage die Datenmengen zu handeln und die benötigten Information zurückzuliefern.

Wir verwenden als verteilte Datenbank Cassandra.

Das ist anders beim Suchen der Tweets: Wir gehen davon aus, dass sich enorme Mengen von Tweets in der Datenbank befinden. Ein naives Durchsuchen wäre sehr teuer und wird deshalb nicht von der verteilten Datenbank angeboten. Wir brauchen also eine Komponente für das Durchsuchen der Tweets. Für sie definiert man Kriterien, nach denen gesucht werden kann, dann liest sie alle Tweets ein und erstellt die entsprechenden verteilten Indexstrukturen. Ab dann ist die Komponente einsatzbereit und kann dann

ebenfalls verteilt unter Zuhilfenahme der vorher erstellten Indexstrukturen die Suchanfragen beantworten. Zudem hat sie Methoden des Rankings aus dem Dokument Retrieval implementiert um die Suchergebnisse gewerten zurückzugeben.

Die nächste Funktionalität, nämlich die statistische Auswertung der Tweets benötigt wiederum eine eigene Komponente, die wiederum verteilt arbeiten muss. Für die statistischen Auswertungen werden in Regelfall alle Tweets untersucht, damit handelt es sich wieder um enorme Datenmengen. Diese können nicht auf einen einzelnen Server geladen werden und dort ausgewertet werden. Zumindest nehmen wir das bei der Entwicklung der Architektur an. Diese Komponente muss also verteilte Auswertung unterstützen. Wie das funktioniert, wird ebenso später beschrieben.

Die Komponenten für die Suche existieren bereits und auch die Komponente für die verteilte Auswertung existiert schon: Elasticsearch für die Suche und Empfehlung und Spark für die verteilte Auswertung.

Wir fassen nochmal zusammen:

- Kernfunktionalität: Tweets rein wie raus
- Suche und Empfehlung: On-Demand-Berechnung von Antworten auf Suchanfragen unter Nutzung vorbereiteter Datenstrukturen
- Analytics: Vorberechnung der Analyseergebnisse und dann einfache Rückgabe dieser Ergebnisse auf Anfrage
- Realtime-Analytics: Analyse von Stromdaten.

Es reicht nicht aus, skalierbare, verteilte Komponenten zu benutzen. Zwischen den Komponenten werden enorme Datenmengen ausgetauscht. Damit hier keine Flaschenhälse entstehen, müssen auch die Verbindungen zwischen den Komponenten verteilt sein, wenn sie die ganzen Daten auf einmal verarbeiten. Alternativ müssen die Komponenten die Daten durchgehend während ihre Entstehung laden.

Es lohnt sich den Datenfluss in unserer Twitter-Klon-Architektur sich genauer anzuschauen.

2.2.5 Datenfluss: Stromdatenverarbeitung

Um zu verdeutlichen, wie der Datenfluss aussieht, betrachten wir kurz unterschiedliche Webanwendungen mit hohem Leistungsanforderungen

- Nachrichtenportal: Hier stehen viele Artikel zum Lesen bereit. Die Anzahl Lesezugriffe (Abrufe von Artikeln) übersteigt die Anzahl Schreibzugriffe (Einstellen von neuen Artikeln) deutlich. Es geht also im Wesentlichen darum, immer wieder neue Artikel sehr häufig auszuliefern.
- Video-On-Demand: Ähnlich wie das Nachrichtenportal, nur dass die auszulieferenden Einheiten viel größer sind (Videos vs. Texte). Dafür müssen nicht so viele Einheiten in kurzer Zeit ausgeliefert werden. Auch hier steht die Auslieferung im Vordergrund.

Das Anforderungsprofil eines Mikroblogging-Dienst ist etwas anders. Zum einen sind die einzelnen Einheiten sehr klein (Tweets), zum anderen werden diese von den Nutzern selbst generiert. Und zwar sehr viele in sehr kurzer Zeit. Die Tweets besonders populäre Nutzer müssen ebenfalls millionfach ausgeliefert werden. Der Fokus liegt aber längst nicht mehr auf der Auslieferung von Daten. Vielmehr geht es um einen ständigen Strom von Daten in das System ebenso wie aus dem System heraus. Es geht also um Stromdatenverarbeitung.

Dieses Anforderungsprofil harmoniert mit dem Parallelisierungskonzept unserer Architektur. Die einzelnen Komponenten sind verteilt ausgelegt und verteilen die Arbeit auf mehrere einzelne Server. Das geht nun recht einfach. Die einzelnen Tweets können als (nahezu) unabhängig voneinander betrachtet werden. Sie können insbesondere unabhängig voneinander gespeichert werden. Damit können die Tweets zur Speicherung an einen beliebigen Datenbankserver weitergeleitet werden.

Es gibt Ausnahmen von der Unabhängigkeit. Bei einer Folge von Tweets kann es sich um ein Gespräch handeln, d.h. Tweets antworten aufeinander. Es ist wünschenswert, dass solche Folgen auch vollständig angezeigt werden (können) und nicht plötzlich ein Tweet im Gespräch fehlt. Über die Realisierung solcher Anforderungen findet sich unten mehr.

Neben der Skalierbarkeit hat unsere Architektur auch weitere Ziele. Es stellt sich noch die Frage, wie die einzelnen Komponenten verbunden werden und was sie funktional machen.

2.2.6 Microservice-Architektur

Eine ständige Herausforderung bei der Entwicklung von großen System ist die Aufteilung in kleinere Einheiten, die dann besser gehandelt werden können. Schließlich werden diese Systeme meist von vielen Entwicklern geschrieben, die häufig auch noch an verschiedenen Standorten sitzen. In unserem Fall werden sogar einige unterschiedliche Technologien im Backend genutzt.

Der Ansatz der Aufteilung ist es, einzelne Komponenten mit jeweils einer Aufgabe zu entwickeln. Diese Komponenten sollen unabhängig voneinander entwickelt werden können. Offensichtlich hängen die einzelnen Komponenten in unserem Twitter-Clon stark voneinander ab. Alle brauchen zum Beispiel Zugriff auf die Tweets. Das ist kein Widerspruch zum Microservice-Ansatz, es müssen nur die Schnittstellen vorher erstellt werden (und später eingehalten werden).

Dabei ist es wichtig, dass die Schnittstellen nicht eine spezielle Technologie (insbesondere eine spezielle) Programmiersprache voraussetzen. In unserer Architektur ist das Austauschformat an den Schnittstellen meist JSON. Die Tweets sind z.B. JSON-Dateien bei ihrer Erzeugung. Das andere Austauschformat sind die Datenbanktabellen. Für die Datenbank existieren Connectoren in diversen Programmiersprachen. Die einzelnen Spalten sind zum einen universale Datentypen (Ints, Strings, ...) zum anderen Verschachtelungen von den universalen Datentypen.

Hier soll nicht verschwiegen werden, dass dieses Konzept der definierten Schnittstellen vor allem ein Ideal darstellt. Von ihm wird (und muss) manchmal davon abgewichen werden. Beispielsweise zeigt sich später, dass für die eine Komponente zusätzliche Felder im JSON notwendig sind. Es ist nicht möglich, dass von Anfang alles vollständig zu wissen. Zudem sind die Komponenten unabhängig voneinander. Und nicht alle Komponenten stehen in der Kontrolle der Entwicklungsteams. Werden Daten von außen verwendet, in unserem Fall benutzen wird Daten von Twitter, so können die das Format festlegen und auch ohne Nachfrage ändern. In diesem Projekt ist das auch mehrmals passiert. Zwar lässt sich das häufig mit Wrappern, Schema Matching oder auch durch Anpassung an die Änderungen von außen lösen. Allerdings darf der Aufwand dafür nicht unterschätzt werden.

Der zweite Punkt, der hier kritisch angemerkt werden soll, ist das diese Architektur auch nicht vor den typischen Data Cleaning-Aufgaben schützt. Eine Definition des Austauschformats bedeutet nicht, dass a. alle Felder Werte haben, b. das die Werte korrekt sind und c. dass das Format eingehalten wird. Bei den Twitterdaten heißt das z.B. das bei weitem nicht alle Tweets Geodaten haben. Trotzdem sind dafür einige Felder vorgesehen. Auch dieses Problem lässt sich meist lösen. Es zeigte sich aber, dass in unserem Fall einer der Connectoren zur verteilten Datenbank grobe Schwierigkeiten mit Null-Werten hatte.

Die Microservice-Architektur wird bei uns mit weiteren Konzepten erweitert, wodurch ihr Nutzen nochmal deutlicher wird. Ohne diese Erweiterungen ist die Microservicearchitektur vor allem eine Empfehlung an die Systemdesigner Komponenten möglichst unabhängig voneinander zu designen und dadurch eine niedrige Kopplung zu erreichen. Das bedeutet auch, programmiersprachenunabhängige Schnittstellen zu bevorzugen.

Wir erweitern das Microservice-Konzept um eine Publish-Subscribe-Architektur und später durch ein Denken in Ereignissen. Das führt zu einer guten Erweiterbarkeit.

2.2.7 Die Publish-Subscribe-Architektur

Zur Übertragung der Datenströme zwischen den Komponenten nutzen wir Kafka. Es handelt sich dabei um eine verteilte Streaming Plattform. Diese Software kümmert sich um die Umsetzung der Datenströme. Sie ist selbst wieder natürlich skalierbar, und zwar scale-out skalierbar. Im Kern geht es um die Weiterleitung von Nachrichten an alle interessierten Komponenten. Ein Datenstrom besteht also aus einer Folge von Nachrichten.

Im Kern von Kafka steht das Publish-Subscribe-Modell. Es werden Kanäle angelegt, so genannte Topics. Eine Komponente kann sich nun bei Kafka für diesen Kanal anmelden ("subscribe") und bekommt ab dann alle Nachrichten von Kafka zugestellt. Eine Komponente kann dann auch Nachrichten an den Kanal schicken ("publish") und Kafka kümmert sich dann um die Verteilung an alle Subscriber.

Dies ist eine der Stellen, bei der leicht ein Flaschenhals entstehen kann. Die verteilte Datenbank kann sich nicht direkt mit Kafka verbinden. Stattdessen muss eine Komponente dazwischen geschaltet werden. Diese Komponente

muss wiederum verteilt arbeiten. Ansonsten werden alle Tweets über einen Server geleitet. Bis zu einer gewissen Geschwindigkeit (Tweets pro Zeiteinheit) mag das gut gehen, aber unsere Architektur soll ja gerade so designed sein, dass man in diesem Fall einfach weitere Server hinzufügen kann.

Ein weiterer Fall sind die Webserver. Wenn sie für den Nutzer Tweets aus der Datenbank anfordern, dann läuft das wieder über ein solches Topic in Kafka. Das Problem entsteht dann, wenn sich einfach alle Webserver auf das Topic subscriben. Denn dann erhalten alle Webserver alle Tweets, die von Webservern angefordert worden. Sie sollen aber nur die Tweets bekommen, die sie auch selbst angefordert haben. Kafka löst das Problem mit so genannten Consumer Groups. Wie das genau funktioniert, wird weiter unten beschrieben.

Nun wollen wir uns noch anschauen, wie unsere Architektur Erweiterbarkeit unterstützt.

2.2.8 Erweiterbarkeit des Systems

Wir schauen uns das am Beispiel der Realtime-Analytics an. Ihre Aufgabe ist es einfache statistische Auswertungen zu den aktuell abgesendeten Tweets zu liefern, z.B. die Top 10 Hashtags der aktuellen Stunde. Es gibt jetzt verschiedene Möglichkeiten an die dafür benötigten Tweets zu kommen:

- die Tweets aus der Datenbank lesen
- die Tweets vom Webserver nicht nur in die Datenbank schreiben lassen, sondern auch an die Realtime-Analytics-Komponente zu schicken

Beide Verfahren haben erhebliche Nachteile. Das erste Verfahren würde in sehr vielen Datenbankabfragen münden, wenn man die Statistiken z.B. jede Sekunde aktualisieren möchte. Das zweite Verfahren würde eine Änderung des Webserverns benötigen.

Beide Verfahren sind also nicht geeignet genug. Unsere Architektur kann das Problem hier komfortabel lösen. Es ist ähnlich dem zweiten Verfahren. Allerdings sendet nicht der Webserver die Daten an die Realtime-Analytics-Komponente. Stattdessen subscribt sie sich die Realtime-Analytics-Komponente auf das Topic, auf dem der Webserver die Tweets an die Datenbank sendet. Und schon kümmert sich Kafka darum, dass die Tweets auch dieser Kompo-

nente in Echtzeit zur Verfügung gestellt werden. Es müssen weder Änderungen am Webserver vorgenommen werden, noch an der Datenbank.

Ebenso einfach könnte man die Realtime-Analytics-Komponente auch wieder entfernen. Man meldet sie einfach von Kafka ab.

Hier sieht man, dass eine kleine Änderung im Verständnis der Komponenten vorteilhaft ist: Statt ein Topic für eine Aufgabe zu benutzen, benutzt man eine Topic für Ereignis in der Realwelt.

2.2.9 Denken in Ereignissen

Bleiben wir im Beispiel des Hinzufügens der Realtime-Analytics-Komponente. Möglicherweise hätte man das Topic, in dem der Webserver seine Tweets an die Datenbank schickt auch so benannt und gedacht: `WebserverTweetsToDatabase`. Sobald jetzt aber die Realtime-Analytics-Komponente hinzugefügt werden soll, passt der Name nicht mehr. Eine Erweiterung auf `WebserverTweetsToDatabaseAndRealtimeAnalytics` löst das Problem nicht. Sobald man die nächste Komponente hinzufügen will, muss man wieder das Topic ändern. Also ziehen wir es zusammen zu `WebserverTweetsToX`.

Unsere Architektur schlägt vor nicht in den verbundenen Komponenten zu denken, sondern in Realweltereignissen. Was wäre nämlich dann, wenn Tweets nicht nur über Webserver gesendet werden könnten, sondern auch über Bots? Dann würde man schnell bei `XTweetsToY` als Beschreibung landen. Und das zugehörige Ereignis in der Realwelt wäre: Der Nutzer sendet einen Tweet ab. Und so sollte nach dieser Regel in der Architektur das Topic auch `UserIssuesTweet` genannt werden.

2.2.10 Ausfallsicherheit

Zusätzlich zur Skalierbarkeit der Verteilung können die Komponenten eine gewisse Ausfalltoleranz sicherstellen. Dadurch dass die Daten auf unterschiedliche Server verteilt werden, können die Daten auch jeweils an mehr als einen Server geschickt werden. Bei einer Datenbank können die Daten dann auf mehreren Servern gespeichert werden. Dadurch entstehen lokale Kopien der Daten. Diese Kopien können dann beim Ausfall die Rolle des ausgefallenen Server übernehmen. Auch die verteilte Auswertesoftware hat ein Mechanismus zur Ausfallsicherheit.

Dabei handelt es sich um einen Tradeoff. Zum einen beim Speicherplatz. Je häufiger die Daten gespeichert werden, desto mehr Platz wird für die gleiche Menge Daten benötigt. Zum anderen in zeitlicher Hinsicht. Die Daten werden über das Netzwerk an die Server gesendet. Wenn man nun sicher gehen möchte, dass die Kopien auch vorhanden sind, muss darauf warten dass die Server Erfolg signalisieren. Dies kann bei mehr als einem Server deutlich länger dauern als bei einem Server.

Diese Überlegung führt zu Grenzen der Architektur. Durch die Verbindung über das Netzwerk können nicht alle gewünschten Eigenschaften sichergestellt werden.

2.2.11 Grenzen der Architektur

Normalerweise wird die Software aus unserer Architektur in IP-basierten Netzwerken verwendet werden. Dies sind so genannte asynchrone Netzwerke. Das heißt, dass die Nachrichtenlaufzeit zwischen zwei Servern ist nicht durch einen festen Wert begrenzt. Das bedeutet, dass man nicht feststellen kann, ob einen Server ("Knoten") ausgefallen ist oder die Nachricht nur verzögert ist und noch zugestellt werden würde. In der Praxis wird diese Eigenschaft simuliert, indem nach einem gewissen Timeout angenommen wird, dass der andere Knoten ausgefallen ist. Er wird dann aus dem System genommen.

Zum anderen kommt es zu Problemen, wenn die Rechner untereinander nicht mehr erreichbar sind. Dadurch, dass die Ausführung verteilt wird, sind auch mehrere Server zum Funktionieren erfordert. Dadurch steigt die Ausfallwahrscheinlichkeit mit der Anzahl der Rechner. Deshalb sind auch Mechanismen zur Ausfallsicherheit notwendig. Trotzdem besteht die Möglichkeit, dass das Netzwerk zwischen den Servern ausfällt. Das CAP-Theorem besagt, dass in diesem Fall nur entweder das System an jedem Knoten erreichbar bleibt oder konsistent ist. Für was man sich entscheidet, hängt von den gewünschten Zieleigenschaften der Architektur ab.

Chapter 3

Cassandra

Cassandra ist die Quelle aller Daten von Twitter, die wir brauchen. Dazu werden die Daten direkt von der Twitter API über Kafka in Cassandra geladen und auf ein vorher für unsere Bedürfnisse zugeschnittenes Datenschema gemappt.

3.1 Datenverwaltung

Wir haben uns entschieden das Twitter Datenschema zu übernehmen. Da allerdings die Twitter Dokumentation nicht genau genug ist und nicht alle Attribute aller Datentypen übersichtlich darstellt, haben wir eine Applikation geschrieben, die sich Tweets vom Twitter Stream holt und daraus das Datenschema im Json Format zusammen baut. Nach dem wir die Applikation lange genug laufen lassen haben, hat sich an dem Datenschema nichts mehr geändert und wir konnten die Datentypen extrahieren.

3.1.1 Datenschema

Nach einer eingehenden Untersuchung aller möglichen Use Cases sind wir zum Schluss gekommen, dass uns fünf Tabellen alle Funktionen bieten die wir brauchen. Wir haben dabei zwei Tabellen für die User `user_by_id` und `user_by_screen_name` entworfen wie man in Abbildung 3.1. Da man bei Cassandra nur über den Primary Key (PK) auf Zeilen zugreifen und Bereichsabfragen über CQL machen kann, gilt es hier den PK so zu wählen, dass

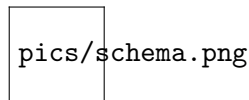


Figure 3.1: Cassandra Schema

alle unsere Funktionen abgedeckt sind. Deshalb haben wir bei neben der User-Id für `user_by_id` und dem Screen-Name der Users für `user_by_screen_name` auch den Timestamp mit aufgenommen. Da Cassandra leider keine vollständige Konsistenz bietet, müssen wir uns selber darum kümmern. Durch den Timestamp können wir verschiedene Versionen eines Objektes auseinander halten und die neuste bestimmen. Somit können wir zumindest einen gewissen Grad an Konsistenz bieten. Die weiteren Attribute der beiden Tabellen lassen sich einfach erklären. Die Follower und Friends eines Users sind wichtig, um die Timeline zu erstellen. Den `password_hash` in `user_by_screen_name` brauchen wir für den Login.

Die anderen drei Tabellen sind dafür da, die Tweets zu speichern und alle Tweet betreffenden Anfragen zu beantworten. Auch hier haben wir wieder den Timestamp bei allen Tabellen mit in den PK aufgenommen um Teilkonsistenz zu gewährleisten. `tweets_by_tweeted` speichert alle Tweets nach der User-Id des Users ab, der den Tweet abgesetzt hat. `tweets_by_follower` hingegen speichert einmal alle Tweets nach User-Id eines jeden Followers ab. Das Konzept hier ist es, durch die mehrfache Speicherung eines Tweets die Zeit bei der Abfrage nach allen Tweets, die ein User auf seiner Timeline sehen kann, zu verkürzen. Da man einmal abgesetzte Tweets auch nicht mehr ändern kann haben wir auch kein Problem damit jedes Objekt für Änderungen wieder herausuchen zu müssen. `tweets_by_hashtag` speichert dann die Tweets danach ab, welche Hashtags in ihnen verwendet werden. Somit können auch Abfragen über Tweets eines Hashtags effizient beantwortet werden.

3.2 Cassandrareader

Die zentrale Applikation in der alle Funktionen und Schnittstellen umgesetzt werden ist der Cassandrareader und es ist eine modular aufgebaute in Java geschriebene Anwendung. Als Framework zur Unterstützung von ver-

schiedenen Funktionen haben wir uns für SpringBoot entschieden. Spring-Boot hat den Vorteil, dass es eine native API für Kafka besitzt, die es uns so sehr leicht ermöglicht Publisher und Subscriber für Kafka-Topics zu schreiben. So ist die Verbindung mit Kafka sehr einfach konfigurierbar und kann innerhalb von kurzer Zeit verwendet werden. Für die Verbindung von Java zu Cassandra haben wir der DataStax Treiber genutzt [2]. Er bietet eine generische Schnittstelle über die man mit Cassandra über CQL kommunizieren kann. Da er gut dokumentiert ist und es sehr viele Beispiele für verschiedene Anwendungen im Internet gibt, verlief die Einarbeitung in die Nutzung des DataStax Treibers sehr schnell. Alle hier und im folgen-

TODO:

*Vielleicht
genau
beschreiben
wie beans
von spring
benutzt
werden*

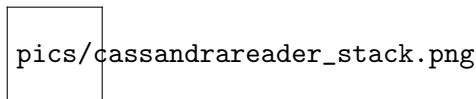


Figure 3.2: Technologie Stack des cassandrareaders

den genutzten Bibliotheken werden über Maven eingebunden und der Applikation so zur Verfügung gestellt. Somit ist sichergestellt, dass immer die richtige Version geladen wird und keine Kompatibilitätsprobleme entstehen.

3.2.1 Architektur

Der Aufbau des Cassandrareaders ist sehr einfach gehalten wie man in Abbildung 3.3 sehen. Die Verbindung zu Cassandra wird vom Singleton CassandraConnector gemanaged. Diese Klasse stellt die Verbindung zu Cassandra her und bietet verschiedene Methoden an, Abfragen an Cassandra über CQL abzusetzen. Die eigentliche Funktion und Implementierung der Use Cases geschieht aber in den Kafka Subscribern. Dazu gibt es eine abstrakte Klasse AbstractKafkaSubscriber, die sozusagen die Infrastruktur bereitstellt. Diese besteht aus dem CassandraConnector, eine Gson-Instanz und mehreren Methoden, die die Optimierungen der Methoden aus dem Cassandra Connector darstellen, wie z.B. Batch-Queries und asynchrone Queries. Die Gson-Instanz kommt von der Google Gson Bibliothek, die für die JSON Konvertierung von Java Klasse zuständig ist. Sie wird in den abgeleiteten Klassen so benutzt, dass Kafka-Anfragen direkt in POJO Objekte gemappt werden, aus denen man dann alle relevanten Informationen bekommt. In den abgeleiteten Klassen wird dann auch die eigentliche

Funktion eines Use Cases implementiert. Alle möglichen Anfragen und


pics/cassandrareader_architecture.png

Figure 3.3: Technologie Stack des cassandrareaders

Antworten über Kafka sind als Java Klassen modelliert und können über Getter-Methoden abgefragt werden. Jeder der abgeleiteten Subscriber besitzt einen Publisher, über den die Antwort, durch Gson konvertiert, wieder versendet werden kann. Die Konfigurationsparameter sind in der application.properties abgelegt und werden in den einzelnen Klassen angesprochen.

3.3 Use Cases

Bei der Identifizierung der Use Cases die für Cassandra relevant sind haben wir uns für die Funktionen entschieden, die für einen Twitter-Client nach MVP-Prinzip (Minimal Viable Product) notwendig sind. Dabei haben wir vor allem die dazu gehören folgende Funktionen:

- Registrierung von User
- Anmelden von Usern
- Abrufen der Timeline:
 - Abfragen von Tweets
 - Abfragen von Usern
- Absenden/Speichern von Tweets

Diese Schnittstellen machen es möglich die Grundfunktionen, also das Erstellen eines Users, das Anmelden, das Senden von Tweets und das Lesen von Tweets über die Timeline von außen über vordefinierte Kafka Nachrichten anzusprechen. Als weitere Schnittstellen für erweiterte Funktionen haben wir eine API für Volltextsuche über Elasticsearch gebaut, die die normale Volltextsuche aber auch Autocompletion unterstützt.

TODO:
*vielleicht
deletion
hinzufügen*

3.3.1 Registrierung von Usern

3.3.2 Anmelden von Usern

3.3.3 Abfragen von Tweets

3.3.4 Abfragen von Usern

3.3.5 Abrufen der Timeline

3.3.6 Absenden/Speichern von Tweets

3.4 Webserver und Frontend

Der Zugriff auf unsere Anwendung wurde mit dem Framework Angular realisiert. Als Schnittstelle zwischen Frontend und dem Message-Broker Kafka wurde ein Webserver geschaltet. Damit es von Außerhalb keine Möglichkeit gibt, direkt auf Kafka zuzugreifen. Ein weiterer Vorteil bei dieser Konstellation ist, dass nur die Verbindung zwischen Frontend und dem Webserver abgesichert werden. Jede weitere Kommunikation innerhalb der einzelnen Prozesse findet in einem eigenen Netzwerk statt.

3.4.1 Frontend

Das Frontend wurde mit dem Framework angularjs erstellt. Es gab keinen speziellen Grund, warum sich das Team für angularjs entschieden hat. Zu Beginn der Hamaube wurde auch kein Fokus auf eine Weboberfläche gesetzt. Der Fokus lag auf dem Backend und nicht dem Frontend. Da es keine Pflicht Aufgabe war und es "schnell" gehen sollte, wurde angularjs gewählt, da schon gewisse Vorkenntnisse vorhanden waren. Zudem bietet angularjs einem die Möglichkeit Module zu entwickeln, welche dann passend eingebunden werden. Der Vorteil daran ist, dass nur einmal das Layout eines Tweets definiert werden muss. Wenn der Webserver ein JSON-Dokument mit einer Liste von Tweets an das Frontend bereit stellt, kann einfach über diese Liste iteriert werden um eine Timeline zu erstellen.

Die klassischen Probleme bei der Erstellung der Webanwendung war das zugreifen auf die JSON-Dokumente. Zum Beispiel wurde das Objekt in dem

JSON-Dokument in einer anderen Ebene erwartet oder beim Bezeichner wurde die Groß/Kleinschreibung verwechselt.

3.4.2 Webserver

Der Webserver wurde mit dem Framework NodeJs erstellt. Dieser bietet verschiedene REST-Schnittstellen, welche mit Hilfe des Paket *express* bereitgestellt wurden. Außerdem wurde eine Bibliothek für die Anbindung an Kafka implementiert. Da der Webserver nur als Vermittler fungiert, wurde eine Funktion implementiert welche zwei Parameter benötigt um das ganze so automatisiert wie möglich zu gestalten. Der erste Parameter dient dem Frage-Topic an den Kafka Server und der zweite für die Antwort in Kafka. Zusätzlich zu diesen zwei Parameter wurde eine Schnittstelle definiert, welche das Frontend mittels einer HTTP REST-Methode ansprechen kann. Der folgende Teil beschreibt die oben beschriebene Funktion:

Es wurde ein LoginController erstellt welcher mit den zwei Parameter *USER_ISSUES_LOGIN* und *LOGIN_RESULT_IS_PROVIDED* die oben beschriebene Funktion aufruft:

```
var kommunikationHandler = new KommunikationHandler
    .constructor( 'USER_ISSUES_LOGIN'
                  , 'LOGIN_RESULT_IS_PROVIDED' );
```

Diese zwei Topics wurden in dem Kaptiel XYZ definiert. Im folgenden Code-Beispiel wird der LoginController mit einer REST-Schnittstelle */users/login* verknüpft.

```
var LoginController =
    require( './endpoints/loginController' );

app.use( '/users/login' , LoginController );
```

Durch dieses Vorgehen, konnte ohne großen Aufwand weitere REST-Endpunkte mit hinterlegtem Topics für das Frontend definiert werden.

Probleme gab es am Anfang zwischen der Kommunikation vom Frontend und der Springboot Anwendung. Durch die Überlastung der Springboot Anwendung, dauerte eine Antwort sehr lange. Damit der HTTP-Request nicht vorzeitig beendet wurde, musste ein ausreichender Timeout gesetzt werden, ansonsten lief eine Anfrage ins leere. Um die Benutzerfreundlichkeit und Sicherheit zu erhöhen generiert der Webserver nach einer erfolgreichen Authentifizierung eine Session-Id. Somit sind alle weiteren HTTP-Request, welche das Frontend versendet abgesichert.

Nachdem die Grundfunktionalität funktionierte wurde der Webserver erweitert.

Das Ziel war es, mehrere Webserver parallel laufen zu haben, damit ein "Load Balancing" simuliert werden konnte. Da wir aber nur eine Virtuelle Maschine zur Verfügung hatten und außerdem kein Netzwerktraffic simulieren konnten wurde das Frontend um eine weitere Funktion erweitert. Im unteren Bereich der Webanwendung kann ein Port definiert werden. Voraussetzung, damit diese Funktion funktioniert, ist dass der Webserver drei mal auf einem verschiedenen Port gestartet wurde. Außerdem ist es nicht möglich, den Webserver während des Betriebs zu wechseln, da die drei Webserver nicht untereinander Kommunizieren und somit keine Session-Ids austauschen.

Wir sind davon ausgegangen, dass mehrere User gleichzeitig Anfragen im Frontend an den gleichen Webserver absenden. Da bei Kafka gesendete Anfragen nicht in der gleichen Reihenfolge abgearbeitet werden, wie sie abgesendet wurden und wir davon ausgehen, dass mehrere Anwender gleichzeitig einen Request an den selben Webserver senden, muss sichergestellt werden, dass die Antwort an den Richtigen Client gesendet wird. Um dieses Problem zu lösen, wurde jede Anfrage von der Webanwendung an den Webserver mit einer inkrementierenden Request-Id versehen. Diese ID wurde dem JSON-Dokument, welches an Kafka zur Bearbeitung weiter gereicht wurde, hinzugefügt. Das Antwort-Topic enthielt diese zu beginn hinzugefügte Request-ID. Somit konnte der Webserver mehrere Anfragen gleichzeitig bearbeiten und es wurde sichergestellt, dass keine Anfrage an einen falschen Client versendet wird.

TODO:
*weitere
hürden bei
mehreren
Web-
server?*

Chapter 4

Umsetzung von Kommunikationsschemen über Kafka

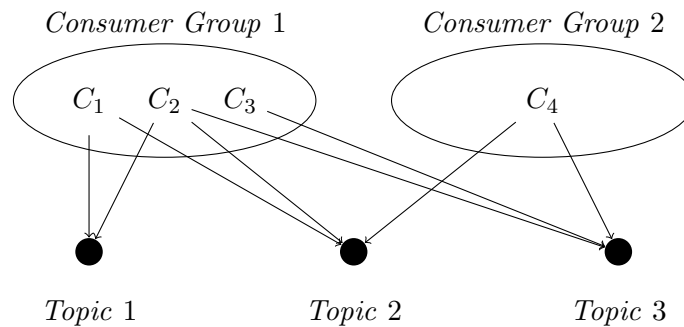
Um Hamaube skalierbar aufsetzen zu können, musste auf die Skalierbarkeit der Kommunikation zwischen den Komponenten geachtet werden. In diesem Kapitel wird beschrieben, welche Techniken eingesetzt wurden, um die Skalierbarkeit der Kommunikation zu gewährleisten. Hierfür werden zunächst die benötigten Grundlagen bezüglich Kafka beschrieben und anschließend wird auf die Realisierung von zwei Kommunikationsschemen mithilfe von Kafka eingegangen.

4.1 Grundlagen Kafka

4.1.1 Registrieren eines Consumers

Wird eine Nachricht über eine Kafka-Instanz verbreitet, so ist sie einem *Topic* zugeordnet. Möchte eine Komponente von Kafka Nachrichten erhalten, muss sie sich als *Consumer* registrieren. Hierbei muss sie mindestens ein *Topic* angeben, an dem sie interessiert ist, sowie eine *Consumer Group*. Wird nun eine Nachricht über Kafka verbreitet, wird ein *Consumer* jeder *Consumer Group*, der an dem *Topic* der Nachricht interessiert ist, ausgewählt und die Nachricht diesen *Consumer* weitergeleitet. Abbildung 4.1 zeigt

Figure 4.1: Beispiel für die Zuordnung von *Consumern* zu *Consumer Groups* und *Topics*.



ein Beispiel für die Zuordnung von *Consumern* zu *Consumer Groups* und die Registrierung der *Consumer* bei *Topics*. Wird in diesem Beispiel ein Nachricht für *Topic 2* hinterlegt, so werden *Consumer 1* oder *Consumer 2*, sowie *Consumer 4* informiert. Wird allerdings eine Nachricht für *Topic 1* hinterlegt, wird nur *Consumer 1* oder *Consumer 2* informiert.

4.1.2 Nachrichtenverteilung

Haben sich mehrere *Consumer* einer *Consumer Group* bei Kafka registriert, um Nachrichten für ein *Topic* zu erhalten, werden diesen *Consumern* zunächst *Partitionen* zugeordnet. Erhält Kafka nun eine Nachricht für ein *Topic*, kann entweder vom *Td* - also dem Sender der Nachricht - festgelegt werden, in welche *Partition* die Nachricht gelegt werden soll, oder Kafka teilt die Nachricht so einer *Partition* zu, dass die Nachrichten möglichst gleichverteilt den *Topics* zugeordnet werden. Nachdem die Nachricht einer *Partition* zugeordnet ist, wird sie dem *Consumer* einer jeden *Consumer Group* übermittelt, dem die *Partition* zugeteilt ist.

Für die Zuordnung der *Partitionen* zu den *Consumern* einer jeden *Consumer Group* wird eine Zuordnungsstrategie verwendet. Hier liefert Kafka die beiden Strategien „range“ und „roundrobin“ mit, von denen eine ausgewählt werden kann. Alternativ lässt sich eine eigene Zuordnungsstrategie definieren.

4.2 Realisierung der Kommunikationsschemen

In unserem Projekt „Hamaube“ haben wir die Semantik der Nachrichten so gewählt, dass sie als *Events* zu verstehen sind. Diese *Events* können potentiell von einer beliebigen Instanz ausgelesen werden. So ist beispielsweise unsere Analyseinstanz an mehrere *Topics* angeschlossen, um Analysen über die Nutzung von Hamaube bzw. die Twitterdaten durchzuführen.

Trotz dieser *Event*-Charakteristik haben viele Nachrichten ein primäres Ziel. So hat beispielsweise ein *Event* das signalisiert, dass sich ein Anwender einen Webserver nach seine Timeline gefragt hat, das primäre Ziel, dass es von einem beliebigen Datenbank-Server aufgenommen wird, um die Einträge der Timeline aus der Datenbank zu laden. Anschließend sollte diese Datenbank-Server ein *Event* verschicken, das signalisiert, welche Einträge dem Anwender angezeigt werden sollen. Dieses *Event* hat wiederum das primäre Ziel, dass der entsprechende Webserver die Anfrage des Nutzers beantworten kann. Im Folgenden wird die Komponente, die das *Event* primär erhalten soll, Kommunikationspartner genannt.

4.2.1 Beliebiger Kommunikationspartner

Für viele Nachrichten, die Hamaube über Kafka verschickt, reicht es, wenn eine beliebige Instanz einer bestimmten Gruppe von Servern das Event erhält. Wenn beispielsweise ein Webserver einen Datenbank-Server darüber informieren möchte, dass die Einträge einer Timeline geladen werden sollten, ist für den Webserver egal, welcher Datenbank-Server das Event zur Bearbeitung der Anfrage erhält.

Soll ein beliebiger Kommunikationspartner für ein Event ausgewählt werden, kann das Event von Kafka einer beliebigen Partition zugeteilt werden und eine der beiden Standard Zuordnungsstrategien gewählt werden, um die Partitionen den *Consumern* der *Consumer Group* zuzuordnen.

4.2.2 Ausgewählter Kommunikationspartner

In einigen Fällen, ist das *Event* allerdings primär nur für eine Instanz entscheidend. So sollte das *Event*, dass die Einträge einer Timeline enthält beispielsweise möglichst direkt den Webserver erreichen, an dem die Timeline ange-

Figure 4.2: Beispiel für die Umverteilung der Partitionen mit dem „roundrobin“ Verfahren: Zwischen Zustand 1 und Zustand 2 wurde der Consumer 2 entfernt.

Zustand 1:		
Consumer 1	Consumer 2	Consumer 3
Partition 1	Partition 2	Partition 3
Partition 4	Partition 5	

Zustand 2:	
Consumer 1	Consumer 3
Partition 1	Partition 2
Partition 3	Partition 4
Partition 5	

fragt wurde. Um dies zu realisieren, könnte der Webserver der Anfrage seine ID mitschicken und dann das Antwort-*Topic* nach seiner ID filtern. Diese Lösung würde allerdings dazu führen, dass die Kommunikation nicht mehr skalierbar wäre, da dann jeder Webserver jede Antwort auf jede Anfrage erhalten müsste und zum Filter der Antworten bearbeiten müsste. Dies ließe sich nicht skalierbar realisieren. Die *Consumer* müssen also der selben *Consumer Group* zugeordnet sein.

Um nun zu gewährleisten, dass der Webserver, der die Anfrage gestellt hat, die Antwort erhält, muss die Antwort zu einen in eine feste *Partition* gelegt werden und zum anderen musste eine Strategie für die Zuordnung von *Partitionen* auf die *Consumer* implementiert werden, mit der garantiert werden kann, dass die Antwort auch bei erneuter Verteilung der *Partitionen* beim richtigen *Consumer* landet. Dies können beide von Kafka mitgelieferten Verteilungsstrategien nicht gewährleisten, wie in Abbildung 4.2 beispielhaft für das „roundrobin“-Verfahren dargestellt.

Wird eine Webserver-Instanz von Hamaube gestartet, muss ihr eine Partitionsnummer als Argument übergeben werden. Registriert sich die Webserver-Instanz bei Kafka als Consumer, kann sie eine ID übergeben, die später an den Verteilungsalgorithmus für die *Partitionen* weitergegeben wird. Durch die ID kann der selbst implementierte Verteilungsalgorithmus erkennen welche

Partition diesem Consumer zugeordnet werden muss. Ist eine Partition vorhanden, für die kein Consumer existiert, dessen ID die Nummer der Partition enthält, wird die Partition einem beliebigen Consumer zugeordnet.

Schickt ein Webserver nun eine Anfrage an einen Datenbank-Server, enthält diese Anfrage ein Feld, das die Partition angibt, in die die Antwort gelegt werden soll. Durch den Verteilungsalgorithmus für die Partitionen ist garantiert, dass dem Webserver, solange er erreichbar ist, diese Partition zugeordnet ist, sodass nur er die Antwort erhält. Ist der Webserver nicht mehr erreichbar, wird seine Partition einem anderen Webserver zugeordnet, der die Antwort erhält und verwirft, da er sie Anfrage nicht gestellt hat. Die Antwort ist in diesem Falle also verloren.

Chapter 5

Conclusion

Write here you conclusions

5.1 Future work

Appendix **A**

Glossary

Just comment `\input{AppendixA-Glossary.tex}` in `Masterthesis.tex` if you don't need it!

Symbols

\$ US. dollars.

A

A Meaning of A.

B

C

D

E

F

G

H

I

J

M

N

P

Q

R

S

T

U

V

W

X

Appendix **B**

Cassandra

Cassandra ist eine NoSql Datenbank, die nach dem Wide-Column Model konzipiert ist. Cassandra vereint verschieden Eigenschaften von Googles BigTable [1] und Amazons Dynamo [3]. Sie wurde 2010 von Facebook entwickelt, um das Inbox Search Problem zu lösen [4]. Es ist eine verteilte Datenbank, die sich unter dem CAP-Theoram als AP charakterisieren lässt, wobei man das Level an Konsistenz manuell einstellen kann.

B.1 Daten Model

Das Daten Modell von Cassandra entspricht dem von BigTable, also dem Wide-Column Modell. Dies beruht im Wesentlichen, wie alle Arten an NoSql Datenbank Modellen auf Key-Value-Stores. Dabei wird für einen Primary Key jeweils eine Reihe mit Column-Families definiert. Die Column-Families bestehen wiederum aus einem Key, der sie beschreibt, und einem Value der dann den Wert an gibt. Der Wert kann allerdings wieder eine Menge an Column-Families sein, wodurch es möglich ist Column-Families beliebig zu verschachteln. Es wird bei der Initialisierung angegeben welchen Typ der Wert hat, Verschachtlung wird über benutzerdefinierte Typen erzeugt. Column-Families, die wiederum Column-Families beinhalten, bezeichnet man als Super-Column-Families. Bei Cassandra werden bei der Initialisierung einer Tabelle, die auch nichts anderes ist als eine Super-Column-Family, alle möglichen Column-Families angegeben. Sie definiert darüber hinaus den Primary Key über den auf die Werte der Column-Families zugegriffen wer-

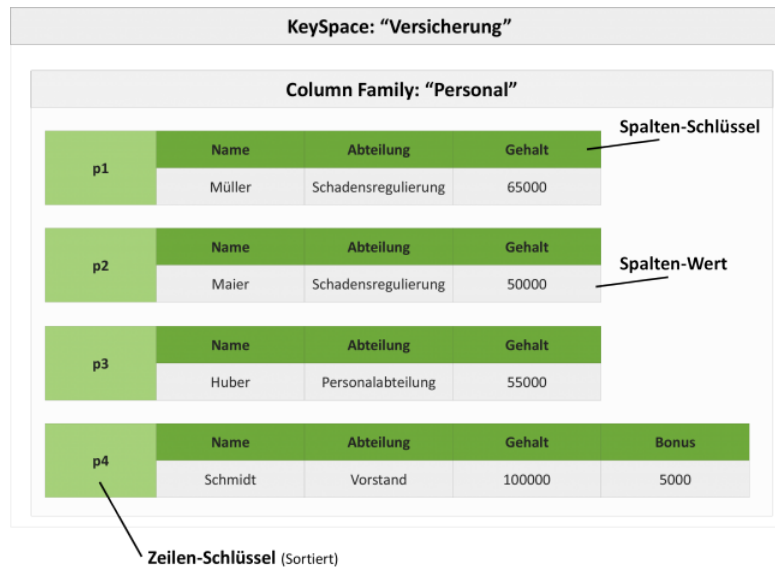


Figure B.1: Beispiel Daten Modell

den können. Im Unterschied zu herkömmlichen SQL Tabellen ist es aber möglich Werte für diese zu unterschlagen, wie in Abbildung B.1 für p1 - p3 dargestellt.

Ein Keyspace stellt die oberste Schicht des Datenmodells da. Für den Keyspace werden bei der Initialisierung eine Replikationsstrategie und eine Anzahl Replikas angegeben, die zu erstellen sind. Diese gelten dann für alle Tabellen, die unter dem Keyspace erstellt werden.

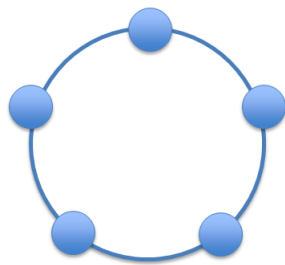
B.2 System

Implementiert ist Cassandra in Java. Darauf aufbauend sind auch die Basis Java Typen verfügbar. Die Daten werden von Cassandra redundant auf verschiedene Instanzen verteilt. Systemnachrichten werden dabei über UDP verschickt, Anwendungsnachrichten, also Nachrichten die mit den Daten zu tun haben, per TCP um den Verlust von Nachrichten zu vermeiden. Bei den Systemnachrichten ist dies zu verkraften.

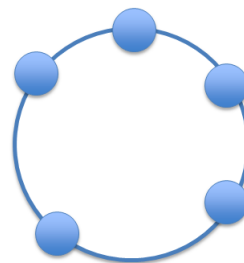
B.2.1 Partitionierung

Die Partitionierung orientiert sich an der von Dynamo [3]. Cassandra benutzt genauso wie Dynamo Consistent-Hashing, um Daten auf die ver-

schiedenen Instanzen zu verteilen. Dabei erhalten die verschiedenen Instanzen einen Wert, der sie uniform über einen vordefinierten Wertebereich verteilt, wie in Abbildung B.2a abgebildet. Consistent-Hashing macht aus dem Wertebereich einen Ring, über den dann die Daten auf die Instanzen wie folgt verteilt werden. Aus einem Datum wird über die Hashfunktion ein Hash-wert berechnet. Das Datum wird dann auf der Instanz abgespeichert, deren Wert auf dem Ring aufsteigend als nächstes kommt. Dieses Verfahren kann zu einer ungleichen Verteilung der Daten auf die Instanzen führen, sodass dadurch die Performance des Systems ineffizient wird. Cassandra löst diese Problem anders als Dynamo dadurch, dass die Werte der Instanzen an die Verteilung der Daten angepasst werden, wie in Abbildung B.2b dargestellt. So sind zwar einige Instanzen für einen größeren Wertebereich zuständig, andererseits kommen in diesem größeren Wertebereich weniger Daten vor, sodass die Daten uniform auf die Instanzen verteilt werden. Wird der Datensatz zu groß, skaliert Cassandra, indem eine Instanz im Consistent-Hashing Ring zwischen den Knoten mit den Meisten Daten eingefügt wird. Danach werden die Bereiche wieder so angepasst, dass alle Instanzen ungefähr gleich belastet sind.



(a) Consistent-Hashing uniform distributed instances



(b) Consistent-Hashing adapted distribution of instances

Figure B.2: Consistent-Hashing Ring

B.2.2 Replikation

Die Art der Replikation in Cassandra ist vom Benutzer konfigurierbar. Die Anzahl der Replikas und die Replikationsstrategie wird durch den KeySpace festgelegt. Dabei kann man zwischen SimpleStrategy und NetworkTopologyStrategy auswählen. Die SimpleStrategy repliziert ohne auf die Netz-

erkstruktur einzugehen. Somit beugt sie weniger stark potentiell dem Datenverlust vor und sollte daher nur für Test-Zwecke benutzt werden. Sei N die Anzahl Replikas, werden die Daten immer auf die $N-1$ Nachfolgeknoten repliziert. Bei der NetworkTopologyStrategy wird die Hierarchie von Datacentern und drin enthaltenen Racks bei der Verteilung betrachtet. Somit wird diese Strategie auch für das Deployment empfohlen. Innerhalb dieser Strategie kann man sich wiederum zwischen Rack Aware und Datacenter Aware entscheiden. Dabei werden die Replikas entweder auf verschiedene Racks oder Datacenter verteilt, um die höchst mögliche Datensicherheit zu gewährleisten. Diese Strategien beziehen sich auf den Koordinator, also die Hauptinstanz einer Partition von Daten, da dieser für die Replikation zuständig ist. Bei der Bestimmung eines Koordinators wird Zookeeper verwendet. Dadurch sind alle Netzwerkänderungen und -konfigurationen persistent gespeichert, da Zookeeper die Konfigurationen jedes Knotens automatisch persistent speichert. Zur Kommunikation werden bei Zookeeper Gossip Algorithmen verwendet.

B.2.3 Persistenz

Persistenz erreicht Cassandra über ein ähnliches System wie BigTable [1]. Zunächst gibt es die MemTable, die im Hauptspeicher gehalten wird und als Cache fungiert. Sie besitzt eine Konfiguration einer Schranke, ab der die MemTable auf die Platte persistiert wird. Auf der Platte gibt es die SSTable, Bloom Filter, index file, compression file und statistics file. Die Daten werden in eine SSTable geschrieben, also eine eigene Datei geschrieben, wenn sie noch nicht vorhanden sind. Wenn dies nicht der Fall ist, wird der betreffende Teil einer SSTable in die Memtable geladen, alle Operationen ausgeführt und die Daten wieder zurück auf die Platte geschrieben. Der Bloom Filter verhindert unnötige Lookups in nicht relevante SSTables. Der Index beschleunigt den Lookup innerhalb einer SSTable. Damit man nicht viele kleine SSTable-Dateien hat, werden zwei SSTables durch einen merge-Prozess immer dann zusammengefasst, wenn einer mindestens halb so groß ist wie der andere. Vorhandene SSTable files können zusätzlich noch komprimiert werden.

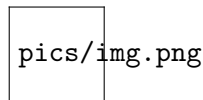


Figure B.3: CQL Mapping

B.3 CQL

Mit CQL (Cassandra Query Language) gibt es eine auf Cassandra zugeschnittene SQL-ähnliche Abfragesprache, die es den Anwendern konventioneller Datenbanken deutlich leichter macht mit Cassandra zu arbeiten. Dabei ist es wichtig zu Wissen dass CQL bei weitem nicht so ausdrucksstark ist wie SQL. Das liegt daran, dass CQL im wesentlichen eine abstrakte API für das Cassandra Datenmodell darstellt. In CQL sind normale Datenbank Typen wie `int`, `text`, etc. möglich, allerdings kann man auch von Collections wie `List`, `Set` und `Map` Gebrauch machen, da es dafür direkte Java Typen gibt. Des Weiteren ist es möglich eigene Typen zu definieren, wie schon in Abschnitt B.1 beschrieben.

Tabellen und Spalten werden wie ebenso in Abschnitt B.1 beschrieben durch Column-Families dargestellt. und wie in SQL erzeugt. Dabei wird ein Primary Key benötigt der dann als Row Key fungiert. Es kann nur über diesen Row Key auf die Zeilen zugegriffen werden. Deshalb kann man in der WHERE-Klausel in Cql auch nur Elemente des Primary Key angeben. Auf diesen Elementen wird durch die Indizierung schon beim speichern in Cassandra eine Sortierung berechnet was die Anfragen deutlich performanter macht. Für alle anderen Spalten der Zeile wäre dies also inperformant und wird von CQL nicht gestattet. Die Spalten und Zeilen werden wie folgt auf das Cassandra Datenmodell abgebildet.

Der erste Teil des Primary Keys bildet wie man sehen kann den Row Key, die restlichen Teile werden mit ihren Werten in die Beschreibung der Column-Families integriert, so wie in Abbildung B.3 beschrieben. Somit werden alle Zeilen mit dem gleichen ersten Teil des Primary Keys in der gleichen Zeile abgespeichert. Über dieses Mapping ist es möglich eine tabelleartige Abstraktion zu erzeugen, die sich durch CQL ausdrückt.

Appendix C

Some text.

C.1 Einführung

Täglich kommen mehrere Petabytes an Daten von über 60 Google Anwendungen zusammen. Dafür verantwortlich sind mehr als 1000 Computer die untereinander vernetzt sind. Um diese Daten verwalten zu können wurde Bigtable ins Leben gerufen. Das Ziel der Datenbank war es in vielen Bereichen anwenden zu können. Dazu sollte es Skalierbar sein sowie eine hohe Performance und Verfügbarkeit besitzen.

C.2 Datenmodell

Bigtable ist eine verteiltes, persistentes multidimensionale sortierte Map. Diese Map ist indexiert über eine row key, column key und einem timestamp. Jeder Wert in dieser Map ist ein Array mit Bytes. Das folgende Datenmodell soll eine Speicherung von Webseiten veranschaulichen.

Das Datenmodell besteht aus zwei Familien dem Inhalt und den Ankerpunkten. Die erste Familie beinhaltet den Inhalt der Webseite, mit drei unterschiedlichen Zeitstempeln (t_3, t_5, t_6). Drei unterschiedliche Zeitstempeln bedeutet, dass die Website `www.cnn.com` in drei unterschiedlichen Versionen abgespeichert wurde. Die Anker Familie beinhaltet jeweils nur eine Version. Den Anker mit „CNN.com“ mit dem Zeitstempel t_8 und dem Anker „CNN“

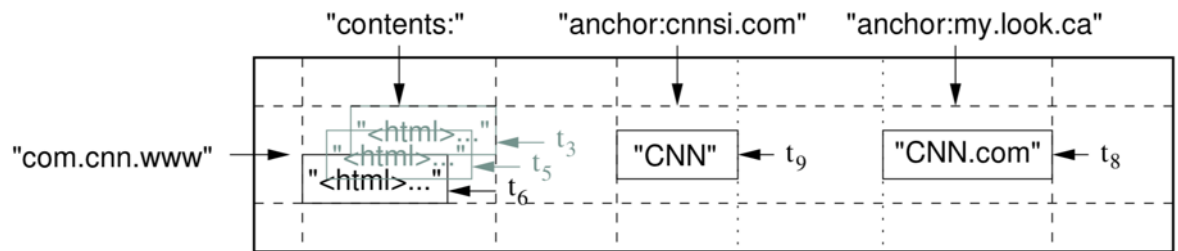


Figure C.1: Tabellen Beispiel

mit dem Zeitstempel t9.

Rows Bigtable speichert Daten in lexikographischer Reihenfolge und sortiert diese nach Zeilen. Eine row range beinhaltet alle gleichnamigen URL's, so dass alle mit der gleichen Domain zusammen abgespeichert werden. Das vereinfacht die Analyse und das Hosting der gleichnamigen Domains und macht dies zudem effizienter.

Column families Verschieden column keys werden in eine gemeinsame Gruppe gespeichert. Das nennt man column families. Alle Daten, welche in der gleichen Gruppe gespeichert werden sind vom gleichen Typ. Bevor Daten in einer Gruppe gespeichert werden können, muss diese column family als erstes erstellt werden.

Timestamp Jede Zelle in Bigtable kann mehrere Versionen der gleichen Daten beinhalten. Dieser Versionen werden indexiert durch den Zeitstempel (timestamp). Der Zeitstempel ist bis auf die Micro Sekunde genau. Durch die „two per-column-family“ kann der Benutzer festlegen, wie viele Versionen der gleichen Daten gespeichert werden sollen. Alle weiteren Versionen werden automatisch gelöscht.

API Die API von Bigtable ermöglicht das Erstellen und Löschen von Tabellen und Spaltennamen, sowie das Ändern von Tabellen, Cluster und Metadaten einer Spaltenfamilie. Das folgende Codebeispiel wurde in C++ geschrieben und verändert den Inhalt der Tabelle Webtable.

Der Code verändert die Spalte „com.cnn.www“ und fügt einen neuen Anker hinzu. Im nächsten Schritt wird ein vorhandener Anker „anchor:www.abc.com“

```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation rl(T, "com.cnn.www");
rl.Set("anchor:www.c-span.org", "CNN");
rl.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &rl);
```

Figure C.2: Zugriffs Beispiel mit der BigTable API

gelöscht und festgeschrieben.

Building Blocks Bigtable ist auf mehreren anderen Teilen der Google-Infrastruktur aufgebaut. Zum Beispiel benutzt Bigtable das verteilte Google File System (GFS). Welches für die Speicherung von Logs und Daten verantwortlich ist. Ein Bigtable Cluster operiert typischerweise auf einem verteilten Pool von Computern. Auf diesem Pool laufen eine breite sparte von verschiedenen Anwendungen. Bigtable basiert auf einem Cluster Management System für Zeit-Planung-Jobs, Ressourcenmanagements auf geteilten Computer, agieren mit Computerfehlern und dem Anzeigen des Computer Status. Das SSTable Format stellt eine persistente, geordnete unveränderliche Abbildung von Schlüsseln zu Werten zur Verfügung, wobei sowohl Schlüssel als auch Werte willkürliche Bytefolgen sind. Operationen werden bereitgestellt, um den Wert zu suchen, der einem bestimmten Schlüssel zugeordnet ist.

Implementation Bigtable besteht aus drei Haupt Komponenten, einer Bibliothek, einem Master Server und vielen weiteren tablet servern. Die Bibliothek ist in jeden Client verlinkt, somit kann der Client auf alle Funktionen zugreifen. Der Master Server wird zufällig ausgewählt. Dieser teilt den tablet servern die tablets zu. Außerdem ist für die Verteilung der Lasten zuständig und ist für die garbage collection. Sobald eine Tabelle zu groß wird, wird diese von einem tablet server gesplittet. So wird sichergestellt, dass eine Tabelle nie größer als 100-200 MB ist.

Tablet location Um Daten zu speichern, verwendet Google bei Bigtable eine „three-level hierarchy“. Das erste Level ist eine Datei, welches auch das Chubby file genannt wird, dort wird der Speicherort des root tablets hinterlegt. Das root tablet beinhaltet alle Speicherorte aller tablets in einer METADATA Tabelle. Das spezielle an dieser Tabelle ist, dass egal wie groß sie wird, diese niemals geteilt wird. Somit wird sichergestellt, dass die „three-level hierarchy“ eingehalten wird. Die METADATA Tabellen speichern die Orte aller anderen tablets in einer Tabelle ab.

Tablet Zuweisung Jedes tablet ist zu einem Zeitpunkt immer nur einem tablet server zugeordnet. Der Master server verfolgt die lebenden tablet servern und die aktuell zugeordneten tablets zu den tablet servern inklusive aller unzugeordneten tablet servern. Beim Start einer Bigtable führt der Master Server folgende Schritte aus:

1. Wählt einen einzigartigen Master Lock in Chubby
2. Scannt die Server Verzeichnisse um die lebenden tablet server zu finden
3. Kommuniziert mit den vorhandenen tablet server um bereits zugeordnete tablets zu identifizieren
4. Master scannt METADATA Tabelle um die vorhandene Zugehörigkeiten zu lernen

Tablet serving Ein persistenter Zustand eines tablets wird in GFS gespeichert. Alle Updates werden auf „well-formed“ geprüft und anschließend in einem commit-log gespeichert. Die neusten Updates werden in eine memtable gespeichert, ältere updates werden in die SSTable geschrieben. Wenn Daten aus dem tablet server abgefragt werden muss ein merge zwischen den neuen Daten in der memtable sowie den älteren Daten in der SSTable erstellt werden.

Compaction Je mehr Daten gespeichert werden, umso größer wird die memtable. Damit diese tabelle nicht zu groß wird, gibt es ein „minor compaction“. Diese Funktion friert eine memtable ein sobald diese eine bestimmte Größe erreicht hat und erstellt eine neue memtable. Die gefrorene

mentable wird zu einer SSTable konvertiert. Je mehr Daten gespeichert werden, desto unordentlicher wird die Ansammlung von SSTable. Um die SSTable zu sortieren wird periodisch ein „merging compaction“ ausgeführt. Dies strukturiert die SSTable neu und es werden Ressourcen, durch die Löschung von Daten, freigegeben. Außerdem werden gelöschte Daten endgültig gelöscht, das ist wichtig für Services, welche sensible Daten beinhalten.

C.3 Refinments

Um die hohe Performance, Verfügbarkeit und Zuverlässigkeit beizubehalten, werden einige Verbesserungen (refinments) benötigt.

Lokale Gruppen Gruppierung erspart Zugriffszeit. Zum Beispiel bei dem Datenmodell Webseite. Die „page metadata“ und „content“ der Webseite werden in einer anderen Gruppe gespeichert. So muss eine Anwendung, welche nur die Metadaten möchte, nicht durch den kompletten Inhalt einer Seite iterieren. Zudem gibt es Tuningparameter, welche bestimmen ob Daten in den Arbeitsspeicher geladen werden um die Zugriffszeit zu minimieren.

Kompression Ein Benutzer kann selbst bestimmen ob SSTable komprimiert wird und falls ja, in welchem Ausmaß. Jeder SSTable Block kann einzeln ausgewählt werden. Für die Komprimierung kommen die Verfahren Bentley und McIlroy's zum Einsatz. Diese können mit 100-200Mb/s kodiert und mit 400-1000 MB/s enkodiert werden.

Caching für Lesezugriffe Für das Caching von Lesezugriffen gibt es zwei Verfahren. Der Scan Cache (high-level), speichert key-value Paare und liefert eine SSTable zurück. Das Block Cache (low-level) Verfahren speichert SSTable Blocks, die von der GFS gelesen werden.

Bloom Filter Benutzer legt selbst fest ob ein Filter zum Einsatz kommt. Der Vorteil eines Filters liegt darin, dass wenn Daten gesucht werden, nicht jede SSTable nach den bestimmten Daten durchsucht werden muss. Ein Bloom Filter erlaubt es, nach einer bestimmten Art von row/column paaren zu fragen, ob diese in einer SSTable gespeichert sind.

Beschleunigte tablet Wiederherstellung Wenn der Master ein tablet von einem Server zu einem anderen Server verschiebt, führt der Ursprungs Server erst ein „minor compaction“ aus um die Ladezeit für den neuen tablet server zu verkürzen.

Unveränderlichkeit ausnutzen Es können nur Daten verändert, welche in der memtable stehen. Daten in der SSTable können nicht verändert werden. Das macht man sich zu nutzen indem man keine Synchronisation braucht, wenn auf die Daten zugegriffen wird. Memtable sind die einzigen Daten auf die man schreiben kann und gleichzeitig lesen. Damit es zu keinen konflikten kommt, setzt Bigtable hier auf „Copy-on-write“.

C.4 Performance Auswertung

Experiment	# of Tablet Servers			
	1	50	250	500
random reads	1212	593	479	241
random reads (mem)	10811	8511	8000	6250
random writes	8850	3745	3425	2000
sequential reads	4425	2463	2625	2469
sequential writes	8547	3623	2451	1905
scans	15385	10526	9524	7843

Figure C.3: Percormance Übersicht

Die Performance wird hauptsächlich durch die verwendete CPU (2ghz) begrenzt. Zudem kann man erkennen, dass bei einem tablet server der Durchsatz bei ca. 75MB/s liegt ($1000 \text{ bytes} * 64 \text{ KB Block size} = 75 \text{ MB/s}$). Damit der Durchsatz bei einem Single tablet server erhöht wird, wird die die SSTable größe in der regel von 64KB auf 8kb gesenkt. Zudem wird erkannt, dass der Durchsatz nicht Linear ansteigt. Bei einer Erhöhung der tablet server von eins auf 500 liegt die Erhöhung des Durchsatzes bei gerade mal dem 300 fachen ($10811 / (500 * 6250) = 350$). Diese Begrenzung liegt wie bei einem tablet server an der CPU der tablet servern.

Appendix **D**

Appendix C

Some text.

D.1 Section 1

D.2 Section 2

Appendix **E**

Appendix C

Some text.

E.1 Section 1

E.2 Section 2

Appendix **F**

Appendix C

Some text.

F.1 Section 1

F.2 Section 2

Appendix G

Appendix D

G.1 Section 1

Bibliography

- [1] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.
- [2] Datastax. Datastax java driver for apache cassandra. <https://docs.datastax.com/en/developer/java-driver/3.3/>. Accessed: 26.08.2018.
- [3] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voss, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.
- [4] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.