

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	VII
1 Introduction	1
1.1 Initial goal and contributions	1
1.2 Thesis outline	1
2 Preliminaries	2
2.1 Topic 1	2
2.1.1 Subtopic1	2
2.1.2 Subtopic2	2
2.2 Topic 2	2
2.2.1 Subtopic1	2
3 Cassandra	3
3.1 Datenverwaltung	3
3.1.1 Datenschema	3
3.2 Cassandrareader	4
3.2.1 Architektur	5
3.3 Use Cases	6
3.3.1 Registrierung von Usern	7
3.3.2 Anmelden von Usern	7

3.3.3	Abfragen von Tweets	7
3.3.4	Abfragen von Usern	7
3.3.5	Abrufen der Timeline	7
3.3.6	Absenden/Speichern von Tweets	7
3.4	Webserver und Frontend	7
3.4.1	Frontend	7
3.4.2	Webserver	8
4	Umsetzung von Kommunikationsschemen über Kafka	10
4.1	Grundlagen Kafka	10
4.1.1	Registrieren eines Consumers	10
4.1.2	Nachrichtenverteilung	11
4.2	Realisierung der Kommunikationsschemen	12
4.2.1	Beliebiger Kommunikationspartner	12
4.2.2	Ausgewählter Kommunikationspartner	12
5	Conclusion	15
5.1	Future work	15
A	Glossary	17
B	Cassandra	22
B.1	Daten Model	22
B.2	System	23
B.2.1	Partitionierung	23
B.2.2	Replikation	24
B.2.3	Persistenz	25
B.3	CQL	25
C	Appendix C	27
C.1	Einführung	27
C.2	Datenmodell	27
C.3	Refinments	31
C.4	Performance Auswertung	32

D	Appendix C	33
D.1	Section 1	33
D.2	Section 2	33
E	Appendix C	34
E.1	Section 1	34
E.2	Section 2	34
F	Appendix C	35
F.1	Section 1	35
F.2	Section 2	35
G	Appendix D	36
G.1	Section 1	36

List of Figures

3.1	Cassandra Schema	4
3.2	Technologie Stack des cassandrareaders	5
3.3	Technologie Stack des cassandrareaders	6
4.1	Beispiel für die Zuordnung von <i>Consumern</i> zu <i>Consumer Groups</i> und <i>Topics</i>	11
4.2	Beispiel für die Umverteilung der Partitionen mit dem „roundrobin“ Verfahren: Zwischen Zustand 1 und Zustand 2 wurde der Consumer 2 entfernt.	13
B.1	Beispiel Daten Modell	22
B.2	Consistent-Hashing Ring	24
B.3	CQL Mapping	26
C.1	Tabellen Beispiel	28
C.2	Zugriffs Beispiel mit der BigTable API	29
C.3	Percormance Übersicht	32

List of Tables

Chapter 1

Introduction

Introduction.

You can reference the only entry in the .bib file like this: [1]

1.1 Initial goal and contributions

1.2 Thesis outline

Chapter 2

Preliminaries

2.1 Topic 1

2.1.1 Subtopic1

2.1.2 Subtopic2

2.2 Topic 2

2.2.1 Subtopic1

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

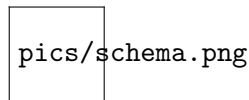


Figure 3.1: Cassandra Schema

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`. 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 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 [?]. 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-

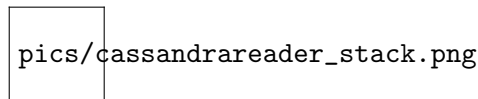


Figure 3.2: Technologie Stack des cassandrareaders

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

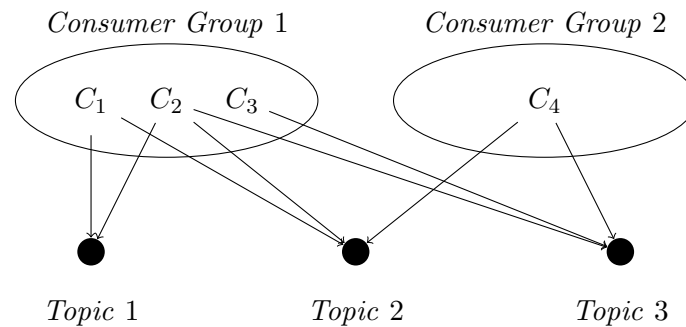
Um Hamaube skalierbar aufsetzen zu können, musste auf die Skalierbarkeit der Kommunikation zwischen den Komponenten geachtet werden. In diesem Kapitel wird beschrieben, welche Techniken eingesetzt wurden, um die Skalierbarkeit der Kommunikation zu gewährleisten. Hierfür werden zunächst die benötigten 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 [?] und Amazons Dynamo [?]. Sie wurde 2010 von Facebook entwickelt, um das Inbox Search Problem zu lösen [?]. 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

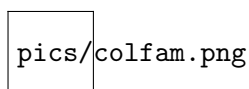


Figure B.1: Beispiel Daten Modell

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 werden 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 Datemodells 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 [?]. Cassandra benutzt genauso wie Dynamo Consistent-Hashing, um Daten auf die verschiedenen 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.

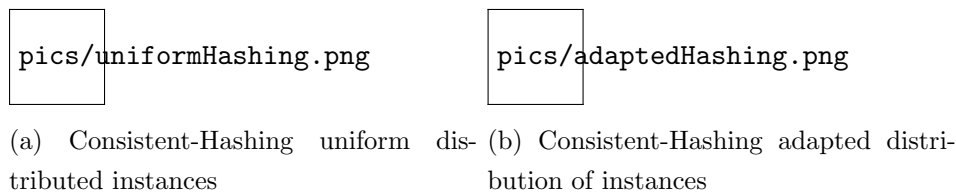


Figure B.2: Consistent-Hashing Ring

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 Netzwerkstruktur einzugehen. Somit beugt sie weniger stark potentiell 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 enthalten Racs bei der Verteilung betrachtet wird. 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 [?]. 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 SSTable, durch einen merge-Prozess immer dann zusammengefasst, wenn eine mindestens halb so groß ist wie die andere. Vorhandene SSTable files können zusätzlich noch komprimiert werden.

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

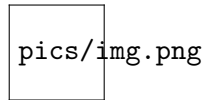


Figure B.3: CQL Mapping

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“

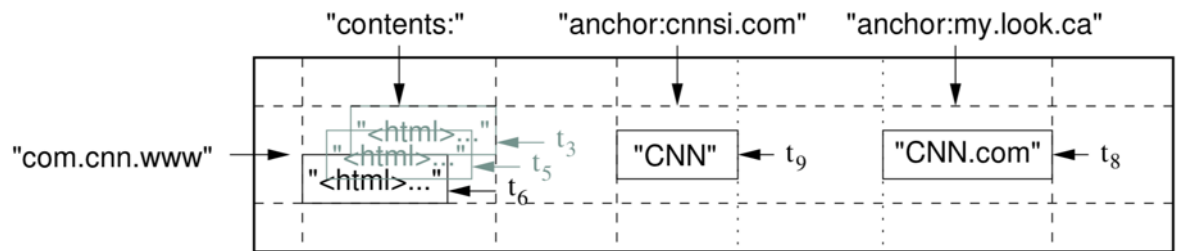


Figure C.1: Tabellen Beispiel

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] Marcelo Arenas, Claudio Gutierrez, and Jorge Pérez. On the semantics of sparql. In *Semantic Web Information Management*, pages 281–307. 2009.