Wide Column Stores mit Map/Reduce im Bereich Web Analytics am Beispiel von Apache HBase und Cassandra

René Trauner

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Oktober 2012

© Copyright 2012 René Trauner

Diese Arbeit wird unter den Bedingungen der *Creative Commons Lizenz Namensnennung–NichtKommerziell–KeineBearbeitung Österreich* (CC BY-NC-ND) veröffentlicht – siehe http://creativecommons.org/licenses/by-nc-nd/3.0/at/.

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 4. Oktober 2012

René Trauner

Inhaltsverzeichnis

Erklärung iii

Vorwort vii

Kurzfassung ix

Abstract x

1 Einleitung 1
1.1 Datenbanksysteme im Wandel 1
1.2 Problemstellung, Motivation und Ziele 2
1.3 Abgrenzung 2
1.4 Gliederung der Arbeit 2

2 NoSQL – Theoretische Grundlagen 4
2.1 NoSQL Definition 4
2.2 Problemstellungen relationale Datenbanken 5
2.3 CAP-Theorem 6
2.3.1 Consistency 6
2.3.2 Availability 6
2.3.3 Partition Tolerance 6
2.4 BASE und Eventually Consistent 7
2.5 Consistent-Hashing 8
2.6 Multiversion Concurrency Control 10
2.7 Map/Reduce 11
2.7.1 Map 13
2.7.2 Reduce 13
2.7.3 Beispiel 13
2.8 Vector Clocks 14
2.8.1 Lamport Clocks 14
2.8.2 Versions-Vektoren 16
2.9 Paxos 18

3 NoSQL-Datenbanksysteme 19

3.1 Wide Column Stores/Column Families 19
3.2 Document Stores 20
3.3 Key-Value Stores 20
3.4 Graphdatenbanken 20

4 Analytics Prototyp 22
4.1 Exkurs Web Analytics 22
4.2 Allgemeine Anforderungen 22
4.3 Auswahl Data Storage 23
4.4 Architektur Analytics Prototyp 24
4.4.1 Tracking 25
4.4.2 Datenbank 26
4.4.3 Data Access Objects (DAOs) 26
4.4.4 Service-Schnittstellen 26
4.4.5 Frontend 26
4.5 Projektstruktur 27
4.5.1 Allgemein 27
4.5.2 Datenbankschema Allgemein 27
4.5.3 Wide Column Stores mit Map/Reduce 28

5 Implementierung Apache Cassandra 29
5.1 Apache Cassandra Allgemein 29
5.1.1 Datenmodell 29
5.1.2 Datenoperationen 30
5.1.3 Distribution 31
5.2 Implementierungsdetails Prototyp 32
5.2.1 Datenbankschema Prototyp 32
5.2.2 API Schnittstellen 32
5.2.3 Data Tracking 34
5.2.4 Map/Reduce Integration 36

6 Implementierung Apache HBase 43
6.1 HBase Allgemein 43
6.1.1 Datenmodell 43
6.1.2 Datenoperationen 44
6.1.3 Distribution 45
6.2 Implementierungsdetails Prototyp 45
6.2.1 Datenbankschema Prototyp 45
6.2.2 API Schnittstellen 45
6.2.3 Data Tracking 47
6.2.4 Map/Reduce Framework 49

7 Evaluierung und Ergebnisse 52
7.1 Motivation der Evaluierung 52

7.2 Testumgebung 52
7.3 Vorbereitung und Durchführung des Evaluierungsprozesses . . 53
7.3.1 Testergebnisse Data Tracking 53
7.3.2 Testergebnisse Map/Reduce 54

8 Schlussbemerkungen 57
8.1 Conclusio 57
8.2 Ausblick 58

A Inhalt der CD-ROM/DVD 60
A.1 Masterarbeit 60
A.2 Literaturquellen 60
A.3 Quellcode 60
A.4 Sonstiges 60

Quellenverzeichnis 61
Literatur 61
Online-Quellen 62

Vorwort

Das Thema Datenbanken hat mich seit Beginn meines Studiums begleitet. Bereits der Einführungskurs in die Welt der relationalen Datenbanken erweckte großes Interesse in mir. Es war faszinierend zu erkennen, wie sich unzählige Abläufe und Problemstellungen des täglichen Lebens, mithilfe dieser relationalen Modelle abbilden und lösen lassen.

Als ich vor drei Jahren schließlich erstmals in einer Fachzeitschrift den Begriff „NoSQL“ las, wollte ich sofort mehr über dieses Thema in Erfahrung bringen. Zuerst schien es mir nicht glaubwürdig, dass es vernünftige Alternativen zu den bewährten relationalen DBMS geben kann. Doch dass es sich bei diesen NoSQL-Systemen um mehr als nur einen kurzweiligen Trend handelt, kann man mittlerweile mit ruhigem Gewissen behaupten. Zahlreiche Publikationen und Fachartikel kreisen um das Thema. Nicht zu verachten sind auch die bereits über 100 unterschiedlichen Datenbanksysteme, welche unter der Kategorie NoSQL zu finden sind. Für mich war dies, in jedem Fall, Grund genug einen tieferen Blick in die Materie zu wagen. Zahlreiche Lobgesänge auf diese Datenbanken, welche die Prinzipien der relationalen Systeme „über den Haufen warfen“, weckten meine Neugier. Namhafte Unternehmen wie Facebook, Google oder Amazon betreiben bereits große Teile ihrer Infrastruktur mithilfe von NoSQL-Datenbanken. Alle Probleme, welche durch relationale Datenbanken entstanden, wie beispielsweise die eingeschränkten Skalierungsmöglichkeiten, aufwändige Replikationsmechanismen oder Schemarestriktionen, schienen plötzlich gelöst.

Doch was hatte es damit genau auf sich? Die Gelegenheit dies rauszufinden bot sich mir Ende des Vorjahres. Ein Freund berichtete mir, dass die *Racon Software GmbH* in Linz ein interessantes Projekt im Bereich Web Analytics in Verbindung mit NoSQL-Datenbanken umzusetzen plant. Da sie dazu noch einen Diplomanden suchten, beschloss ich umgehend in das Projekt einzusteigen und so begann bereits Ende des Jahres 2011 die Implementierung eines proprietären Prototypen. Vorerst gestaltete sich die Recherche nach einem geeigneten Datenbanksystem als äußerst aufwändig und schwierig. Die beiden Apache Projekte Cassandra und HBase kristallisierten sich jedoch sehr schnell, als für diesen Anwendungsfall besonders geeignet, heraus. Beide Systeme erscheinen auf den ersten Blick sehr ähnlich, jedoch lassen sich bei genauerer Betrachtung grundlegende Unterschiede ausmachen.

Aus diesem Grund habe ich es mir schließlich zur Aufgabe gemacht, im Rahmen dieser Masterarbeit eine Entscheidungsfindung herbeizuführen. Diese betrifft die Wahl des idealen Datenbanksystems für die Web Analytics Anwendung der *Racon Software GmbH*, eingeschränkt auf die beiden Systeme HBase und Cassandra.

Abschließend möchte ich an dieser Stelle noch jenen Personen Dank aussprechen, die mich bei der Entstehung dieser Masterarbeit und während meines gesamten Studiums tatkräftig unterstützt haben. Dieser gebührt vor allem den Professoren der FH Hagenberg, welche stets bemüht waren, ihr Fachwissen bestmöglich an uns zu vermitteln und unsere Interessensgebiete zu fördern und zu verknüpfen. Besonders DI Martin Harrer danke ich für die hervorragende Betreuung und sein Engagement während der Entstehung dieser Masterarbeit. Weiters bedanke ich mich bei der Firma *Racon Software GmbH*, die es mir ermöglicht hat, die Arbeit in einem produktiven Umfeld umzusetzen. Insbesondere möchte ich hier Roland Thaler erwähnen, dessen konstruktive Kritik und innovative Ideen maßgeblich zu einem erfolgreichen Abschluss beigetragen haben.

Kurzfassung

In Zeiten rasant wachsender Webanwendungen beginnen relationale DBMS an ihre Grenzen zu stoßen. Es gilt riesige Datenmengen zu verwalten und dabei auch noch alle Anfragen an das System in kürzester Zeit zu beantworten. NoSQL-Datenbanken scheinen – was diese Anforderungen betrifft – den traditionellen RDBMS den Rang abzulaufen. Eigenschaften von NoSQL, wie horizontale Skalierbarkeit, einfache Replikationsmechanismen und neuartige Konsistenzmodelle, sind wie geschaffen um den Betrieb großer Webapplikationen auch in naher Zukunft sicherzustellen.

In dieser Masterarbeit wird ein Einblick in die wichtigsten Grundsätze dieser alternativen Datenbanksysteme gegeben. Dabei werden die namhaftesten Vertreter vorgestellt, wobei sich der Fokus auf das Thema Web Analytics in Verbindung mit den beiden Apache Datenbanksystemen Cassandra und HBase verlagert. Mithilfe eines völlig proprietären Prototypen werden beide Systeme evaluiert, um für eine Web Analyse Anwendung ein geeignetes Datenbankmanagementsystem zu finden.

Abstract

In times of fast-growing web applications the limitations of relational DBMS get increasingly exposed. The challenge is to manage vast amounts of data, while still being able to answer all requests to the systems quickly, at the same time. Regarding these requirements the NoSQL databases seem to outstrip traditional RDBMS. Characteristics such as horizontal scalability, simple replication and new consistency-models are ideal to ensure the operation of large web applications in the near future.

This thesis will first give an overview of the the most significant principles of these alternative database systems. Thereafter the most eminent members of NoSQL databases will be presented, with the focus shifting to the subject of Web Analytics in connection with the two Apache database systems: Cassandra and HBase. These two systems are going to be evaluated on the basis of a completely proprietary prototype in order to find the most appropriate database management system for a web analytics application.

Kapitel 1

Einleitung

1.1 Datenbanksysteme im Wandel

Betrachtet man die aktuellen Entwicklungen im Bereich Data Storage, lassen sich in den letzten Jahren viele grundlegende Veränderungen und Umbrüche ausmachen. Um einen genaueren Einblick in jene Thematik zu geben, erfolgt an dieser Stelle vorerst eine Definition des Begriffs „Big Data" [13]:

> *Als Big Data werden besonders große Datenmengen bezeichnet, die mit Hilfe von Standard-Datenbanken und Datenmanagement Tools nicht oder nur unzureichend verarbeitet werden können. Problematisch sind hierbei vor allem die Erfassung, die Speicherung, die Suche, Verteilung, Analyse und Visualisierung von großen Datenmengen. Das Volumen dieser Datenmengen geht in die Terabytes, Petabytes, Exabytes und Zettabytes.*

Durch das rasante Wachstum vieler Unternehmen, welche im Zeichen des Web 2.0 stehen, entstanden auch stetig wachsende, riesige Mengen an Daten. Diese galt es nun zu verwalten, sowie die Applikationen und Plattformen so zu gestalten, dass der Endbenutzer keine Einbußen hinsichtlich Reaktionszeiten des Systems hinnehmen muss und auch sonst keine beträchtlichen Einschränkungen im Bereich User Experience erfährt. Sehr schnell wurde klar, dass diese unumgänglichen Anforderungen mithilfe relationaler DBMS nicht oder nur sehr schwer erfüllt werden können – wodurch ein beträchtlicher Wandel in der Welt der Datenbanksysteme eingeleitet wurde.

Zu Beginn des Jahres 2009 führte schließlich Johan Oskarsson den Begriff „NoSQL" – was per Definition die Abkürzung für „Not Only SQL" ist – ein. Dieser Begriff steht heute in Verbindung mit einer Vielzahl an DBMS, welche hauptsächlich für die Verarbeitung großer Datenmengen in verteilten Systemen konzipiert sind und aufgrund ihrer Vielfalt für zahlreiche Anwendungszwecke geeignet sind.

1.2 Problemstellung, Motivation und Ziele

Eine grundlegende Problemstellung, die diese Vielzahl an neuen Systemen mit sich bringt, ist mit Sicherheit jene, ein geeignetes Datenbanksystem für den jeweiligen Anwendungsfall zu finden.

In dieser Masterarbeit liegt der Fokus auf der Thematik „Web Analytics“. Da dieser Bereich vorwiegend mit der Auswertung und Analyse einer Vielzahl an Daten verbunden ist, wird im Rahmen dieser Arbeit unter Miteinbeziehung des zugehörigen Projektes versucht, ein geeignetes DBMS für jene Art von Anwendungen zu finden. Um das Ganze in einem überschaubaren Rahmen zu halten, wurden aus dem großen Spektrum an NoSQL-Systemen die beiden Wide Column Stores von Apache – Cassandra[1] und HBase[2] – ausgewählt. Diese Datenbanksysteme eignen sich besonders für die Aggregation von Daten, was durchaus einen Kernbereich von Web Analytics darstellt. Zur Realisierung einer direkten Gegenüberstellung wurde ein entsprechender Prototyp implementiert, welcher als Grundlage für die Evaluierung der beiden Datenbanken dient.

1.3 Abgrenzung

Das Ziel dieser Arbeit beschränkt sich auf die Gegenüberstellung der beiden DBMS hinsichtlich Implementierung einer performanten Tracking-Funktion, sowie der Integration eines Map/Reduce Algorithmus. Daher wurde auch auf den Betrieb der Datenbanken in einem Cluster-System verzichtet und auch nicht näher auf die einzelnen Konfigurationsparameter hinsichtlich Distribution eingegangen. Die programmatische Umsetzung erfolgte in Zusammenhang mit einer Single-Node Installation.

1.4 Gliederung der Arbeit

Einleitung

Dieses Kapitel gibt einen kurzen Einblick in die Thematik sowie den Aufbau dieser Masterarbeit mit den einhergehenden Problemstellungen und Zielen.

NoSQL – Theoretische Grundlagen

Die wichtigsten Themen rund um NoSQL werden in diesem Kapitel mithilfe von Grafiken und Beispielen behandelt. Es dient vorwiegend dazu, ein grundlegendes Verständnis für die Materie und die darauf folgenden Kapitel aufzubauen.

[1] http://cassandra.apache.org/
[2] http://hbase.apache.org/

NoSQL-Datenbanksysteme

Um einen kurzen Überblick zu erhalten, beinhaltet das Kapitel „NoSQL-Datenbanksysteme“ eine Auflistung der wichtigsten Kategorien und ihrer Vertreter, sowie eine kurze Beschreibung hinsichtlich Funktionsweise und Anwendungsfälle.

Analytics Prototyp

Das vierte Kapitel dieser Masterarbeit beschreibt den Aufbau und die Umsetzung des Prototypen. Dieser gilt als Grundlage für die Evaluierung der Datenbanksysteme und wurde im Rahmen des Masterprojektes umgesetzt.

Implementierung Apache Cassandra

Dieses Kapitel beinhaltet Details und Beschreibungen zur Implementierung der DAOs (Data Access Objects) und der Map/Reduce Jobs von Apache Cassandra.

Implementierung Apache HBase

Dieses Kapitel beinhaltet Details und Beschreibungen zur Implementierung der DAOs (Data Access Objects) und der Map/Reduce Jobs von Apache HBase.

Evaluierung und Ergebnisse

Die Evaluierung der DBMS und die damit einhergehenden Ergebnisse werden im siebten Kapitel beschrieben. Mithilfe von Diagrammen und Messtabellen erfolgt ein direkter Vergleich der beiden Datenbanken anhand festgelegter Kriterien.

Schlussbemerkungen

Das letzte Kapitel gibt nochmals einen Überblick über die Ergebnisse der Arbeit und enthält ein abschließendes Résumé.

Kapitel 2

NoSQL – Theoretische Grundlagen

Nach einer genauen Erläuterung des Begriffs NoSQL wird in diesem Kapitel ein Überblick über die wesentlichen Unterschiede und neuen Ansätze gegenüber relationalen Datenbanksystemen gegeben. Relevante Aspekte werden auf theoretischer Basis anhand von Beispielen und Grafiken erklärt. Vor allem wird versucht ein allgemeines Verständnis für die Materie aufzubauen und Ansätze einzubringen, welche die Verwendung dieser neuartigen Datenbanksysteme rechtfertigen.

2.1 NoSQL Definition

Beschäftigt man sich etwas genauer mit dem Thema Datenbanken, stößt man seit geraumer Zeit vermehrt auf den Begriff NoSQL. Vor allem seit dem rasanten Wachstum sozialer Netzwerke, cloudbasierter Anwendungen und sonstiger Unternehmen, welche sich dem Web 2.0 verschrieben haben, ist dieser Begriff in aller Munde [21].

Um sich eingehender mit diesem Thema auseinandersetzen zu können, sollte man aber vorerst genau abklären, was es mit diesem Begriff eigentlich auf sich hat. Vor allem stellt sich die Frage, warum NoSQL-Datenbanken eine derartige Innovation mit sich bringen, dass sie bereits bei namhaften Unternehmen wie Google[1] oder Facebook[2] in diversen Bereichen Anwendung finden. Diese – wie bereits erwähnt – als „Not Only SQL“ definierte Abkürzung beschreibt Datenbanksysteme, die als nicht-relational bezeichnet werden können, Daten in unterschiedlicher Art und Weise speichern und dem Benutzer alternative Abfragemechanismen bieten.

Die genaue Definition des Begriffs NoSQL lautet wie folgt: Unter NoSQL wird eine neue Generation von Datenbanksystemen verstanden, die meistens

[1] http://www.google.com/about/company/

[2] https://www.facebook.com/

einige der nachfolgenden Punkte berücksichtigen [5]:

- Das zugrundeliegende Datenmodell ist nicht relational.
- Die Systeme sind von Anbeginn an auf eine verteilte und horizontale Skalierbarkeit ausgerichtet.
- Das NoSQL-System ist Open Source.
- Das System ist schemafrei oder hat nur schwächere Schemarestriktionen.
- Aufgrund der verteilten Architektur unterstützt das System eine einfache Datenreplikation.
- Das System bietet eine einfache API.
- Dem System liegt meistens auch ein anderes Konsistenzmodell zugrunde. Eventually Consistent und BASE, aber nicht ACID[3].

Anhand dieser Aufzählung erkennt man bereits deutliche Unterschiede zu relationalen Datenbanksystemen.

2.2 Problemstellungen relationale Datenbanken

Lange Zeit waren relationale Datenbanken das Maß aller Dinge. Die bewährte Abfragesprache SQL (Structured Query Language), sichere Transaktionsmechanismen und Schemarestriktionen boten eine ideale Grundlage für Unternehmen ihre Daten sicher zu speichern. Jedoch sollte man in Betracht ziehen, dass diese Systeme ursprünglich für reine Business-Anwendungen konzipiert und entwickelt wurden. Deren Transaktionen waren einer strikten Konsistenz untergeordnet. Mit den stetig wachsenden Datenmengen, die von den Anwendungen produziert werden und welche vor allem auf den Erfolg der zahlreichen sozialen Netzwerke zurückzuführen sind, entstand ein Umdenkprozess, aus dem sich die heutigen NoSQL-Systeme entwickelten. Die Anforderungen an das System begannen sich zu ändern. Eigenschaften wie kurze Reaktionszeiten und hohe Ausfallsicherheit sind heute bei vielen Unternehmen nicht mehr wegzudenken und entwickelten sich zu einer Existenzgrundlage. Doch mit der Verwendung relationaler Systeme waren diese Kriterien – vor allem durch das rasante Wachstum der Datenmengen – nicht mehr zu erfüllen und es entwickelten sich zahlreiche Probleme:

- viele schemabedingte Relationen führen oftmals zu langsamen Reaktionszeiten,
- Datenbankdesign nur bedingt auf Skalierung ausgerichtet,
- kostenintensive Lizenzgebühren und Businessmodelle,
- Schemarestriktionen,
- aufwändige Replikationsmechanismen (Atomarität),
- strikte Konsistenz (ACID).

[3] http://de.wikipedia.org/wiki/ACID

Doch trotz dieser auftretenden Probleme werden relationale Datenbanksysteme durchaus immer ihre Daseinsberechtigung haben. Vor allem in Anwendungen, die einer strikten Konsistenz unterliegen, können gegenwärtige NoSQL-Systeme keinen vollwertigen Ersatz darstellen.

2.3 CAP-Theorem

Das CAP-Theorem wurde erstmals von Eric Brewer im Jahr 2000 bei einem Vortrag auf dem ACM-Symposium (Association for Computing Machinery) vorgestellt [17]. Dabei handelt es sich um das Verhalten der drei Eigenschaften Konsistenz (Consistency), Verfügbarkeit (Availability) und Ausfalltoleranz (Partition Tolerance) in verteilten Systemen. Diese drei Grundelemente des CAP-Theorems können wie folgt beschrieben werden:

2.3.1 Consistency

Ein vollständig konsistentes System zeichnet sich dadurch aus, dass nach erfolgter Transaktion in einem verteilten System, alle Knoten, auf denen eine Leseoperation ausgeführt wird, jenen aktuellen Wert zurückliefern. Dies kann jedoch nur dadurch bewerkstelligt werden, indem alle replizierenden Knoten in einem Cluster so lange weitere Operationen blockieren (nicht erreichbar sind), bis die aktuellen Transaktionswerte erfolgreich abgespeichert wurden. Vor allem in großen Systemen kann dieser Vorgang durchaus einige Zeit in Anspruch nehmen und zu langsamen Reaktionszeiten führen.

2.3.2 Availability

Die Erreichbarkeit eines Systems definiert sich über die entsprechende Reaktionszeit einer Anfrage. Bei steigender Anzahl an Anfragen – somit höherem Traffic – sollte diese eine für den Endbenutzer ebenfalls noch akzeptable Zeitspanne umfassen.

2.3.3 Partition Tolerance

Die Ausfalltoleranz definiert sich dahingehend, dass bei Störungen, welche zu Ausfällen einer oder mehrerer Knoten in einem verteilten System führen, dieses trotzdem noch bis zu einem gewissen Grad in Betrieb gehalten werden kann und für den Anwender erreichbar bleibt.

Eric Brewer definierte sein Theorem über diese drei Eigenschaften und stellte die Theorie auf, dass in einem verteilten Datenbanksystem maximal zwei dieser drei Anforderungen vollständig erfüllt werden können (Abb. 2.1). Seth Gilbert und Nancy Lynch lieferten schließlich zwei Jahre später (2002) in ihrem Paper „Brewer's conjecture and the feasibility of consistent, availa-

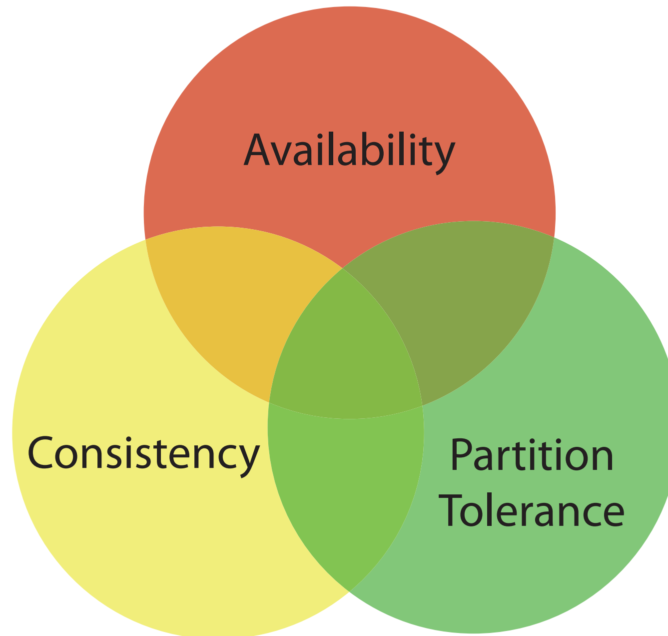


Abbildung 2.1: CAP-Theorem

ble, partition-tolerant web services“ den Beweis für die Existenz des CAP-Theorems [6, S. 24].

2.4 BASE und Eventually Consistent

In relationalen Datenbanksystemen wird stets das Wort Konsistenz groß geschrieben, was beispielsweise bei Geldtransaktionen durchaus Sinn ergibt. Allerdings führte diese Festlegung einer strikten Konsistenz mit wachsender Datenmenge zu längeren Blockaden der Datenbanken und damit auch zu längeren Reaktionszeiten. Angesichts dieser Tatsache entwickelte sich die Idee einer Lockerung der Konsistenz zugunsten der Erreichbarkeit und Ausfallsicherheit von Systemen, woraus eine Alternative zum ACID Konsistenzmodell namens BASE (Basically Available, Soft State, Eventually Consistent) entstand.

Da die Erfüllung der Ausfalltoleranz in verteilten Systemen unumgänglich ist, beschloss man einen Konsens zwischen Erreichbarkeit und Konsistenz zu finden. Dieser Ansatz entwickelte sich weiter und es entstand – basierend auf dem BASE Konsistenzmodell – die Idee der Eventually Consistency. Wie Amazons CTO Werner Vogels in seinem gleichnamigen Paper publizierte [11, S. 3], wird nach dem Grundsatz der Eventually Consistency durchaus in Kauf genommen, dass das System erst nach einem bestimmten Zeitfenster den Status der Konsistenz wieder erreicht. Auf diesem Prinzip bauen vor allem jene

NoSQL-Systeme auf, die im Big Data Bereich angesiedelt sind und schnelle Reaktionszeiten erfordern. Da NoSQL-Datenbanken generell eher im Bereich des BASE Konsistenzmodells zu finden sind und unter diesem Aspekt eine Vielzahl an Ansätzen entstanden sind um die Konsistenz der Daten zu handhaben, wird nun ein kurzer Überblick über diese gegeben [19]:

Causal Consistency Wird von einem Prozess X ein Wert in die Datenbank geschrieben und ein weiterer Prozess Y liest diesen Wert und schreibt wieder in die Datenbank, wird generell davon ausgegangen, dass Prozess Y den von X geschriebenen Wert in seine Berechnungen miteinbezieht und das Ergebnis somit vom Wert, den Prozess X geschrieben hat, abhängig ist. Man spricht daher von Causal Consistency, wenn in jedem Fall Prozess Y erst den Wert erhält, nachdem dieser erfolgreich von Prozess X gespeichert und auf sämtlichen Knoten repliziert wurde.

Read-your-write-Consistency In diesem Fall wird sichergestellt, dass ein Prozess zumindest den zuletzt von ihm geschriebenen Wert erhält und keinen älteren.

Session Consistency Hier wird die oben beschriebene Read-your-write-Consistency für alle Zugriffe innerhalb einer Session auf der Datenbank garantiert. Ist diese abgelaufen und wird eine neue Sitzung gestartet, gilt die Konsistenz nur für jene Operationen in der neuen Session. Für ältere Operationen besteht keine Gewährleistung.

Monotonic Read Consistency Nachdem ein Prozess einen Wert aus der Datenbank gelesen hat, ist es nicht mehr möglich, dass er eine ältere Version dieses Datensatzes erhält.

Monotonic Write Consistency Hier wird sichergestellt, dass alle schreibenden Prozesse in jener Reihenfolge in der sie gestartet wurden auch auf der Datenbank ausgeführt werden. Dies verhindert besonders bei vielen, in kurzer Zeit hintereinander ausgeführten Operationen, das Problem, dass falsche Werte in der Datenbank gespeichert werden, falls bestimmte Befehle nicht in der richtigen Reihenfolge vom System abgearbeitet werden konnten.

2.5 Consistent-Hashing

Consistent-Hashing wird in verteilten Systemen dazu eingesetzt, um für ein Objekt einen Speicherplatz im Netzwerk zu finden, welcher zu einer gleichmäßigen Verteilung der Daten im Cluster führen soll. Vor allem eine dynamische Aufteilung durch Consistent-Hashing ermöglicht das Verkleinern oder Vergrößern eines Clusters ohne die Daten vollständig neu anordnen zu müssen. Als weiteren Punkt kann man auch noch anführen, dass bei diesem Verfahren ebenfalls die unterschiedlichen Kapazitäten der Server in Betracht

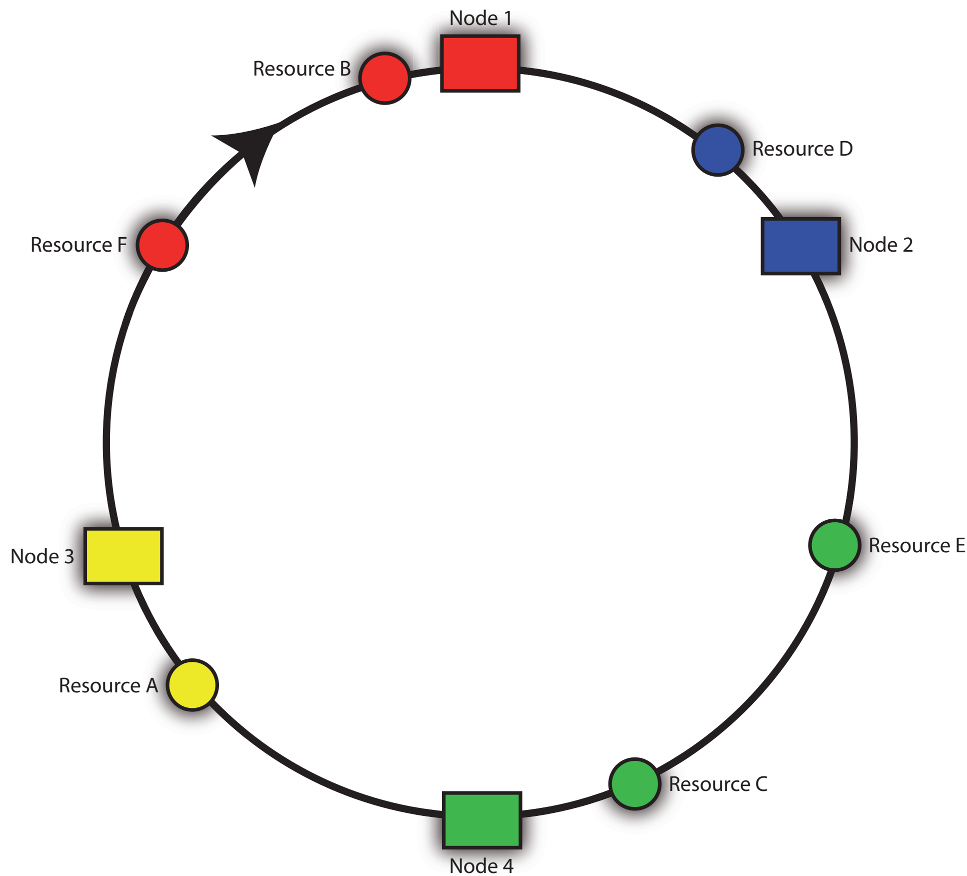


Abbildung 2.2: Verteilungsprinzip Consistent Hashing

gezogen werden. Um dies zu bewerkstelligen, wird jedem Server im Netzwerk mittels einer speziellen Hash-Funktion ein entsprechender Wert zugewiesen. Bei Schreiboperationen werden den einzufügenden Daten, mittels derselben Funktion, Hash-Werte zugewiesen. Danach wird als Speicherort jener Server festgelegt, dessen Wert jenem des einzufügenden Wertes – im Uhrzeigersinn betrachtet – am nächstgelegenen ist (siehe Abb. 2.2). Wobei auch, je nach Kapazität, eine Unterteilung in virtuelle Server vorgenommen wird, denen wiederum entsprechende Hash-Werte zugeordnet werden. Dies führt zu einer dynamischen Speicherung der Objekte in verteilten Datenbanken. Mithilfe des Consistent-Hashing ist ein System ebenfalls in der Lage Serverzugänge oder -ausfälle im Netzwerk zu behandeln, ohne den laufenden Betrieb maßgeblich zu beeinträchtigen.

Bei Amazons Datenbank Dynamo und auch zahlreichen anderen NoSQL-Systemen findet das Verfahren des Consistent-Hashing bereits produktive Anwendung [4].

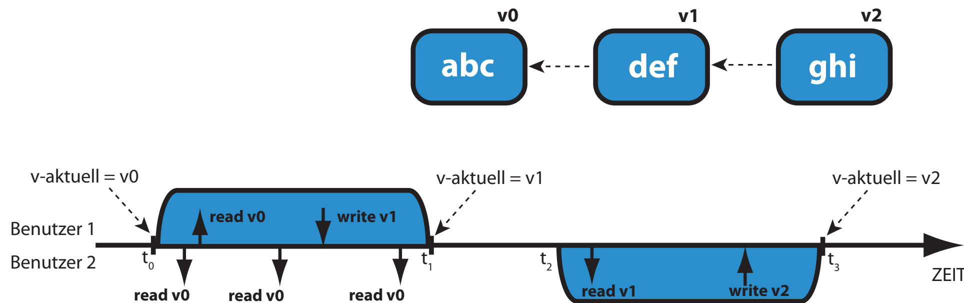


Abbildung 2.3: Leseoperation ohne Blockierung

2.6 Multiversion Concurrency Control

In relationalen Datenbanken kennt man das Problem konkurrierender Zugriffe auf Datensätze. Eine Lösung dieses Problems erfolgt meist in Form von Sperren – sogenannten Locks. Ebenso können aufwändige Replikationsvorgänge dazu führen, dass das System für einen längeren Zeitraum nicht erreichbar ist. Da NoSQL-Datenbanken vor allem in diesem Bereich innovative Ansätze verfolgen, wird hier das Thema Multiversion Concurrency Control behandelt. Das MVCC-Verfahren verzichtet darauf, den Zugriff auf Daten zu sperren, sondern erweitert stattdessen jeden geschriebenen Datensatz um eine Meta-Information. Diese kann sich beispielsweise wie folgt zusammensetzen:

- laufende Transaktionsnummer,
- Zeitstempel des Transaktionsstarts,
- Zeitstempel der aktuellen Schreiboperation,
- Verwendung von Lamport Clocks (siehe Abschn. 2.8.1) oder Versions Vektoren (siehe Abschn. 2.8.2).

Auf diese Art und Weise entstehen von jedem geschriebenen Datensatz mehrere Versionen, welche anhand der obigen Meta-Informationen unterschieden werden können. Der Endbenutzer erhält automatisch vom System immer den zuletzt geschriebenen Wert, ohne dass andere Schreibprozesse mit einer verzögerten Reaktion konfrontiert werden. Bei manchen Datenbanken ist es ebenfalls möglich, auf ältere Versionen eines Datensatzes zuzugreifen und diese zur weiteren Verarbeitung zu nutzen.

Der in Abb. 2.3 dargestellte Ablauf zeigt auf, dass beim MVCC-Verfahren Leseoperationen ohne Verzögerungen durchgeführt werden können. Auftretende Schreibprozesse werden dabei nicht blockiert.

Liegt ein Versionskonflikt wie in Abb. 2.4 dargestellt vor, kann das Datenbanksystem diesen in den meisten Fällen lösen. Es besteht die Möglichkeit die beiden neuen Versionen zu vereinen. Allerdings nur unter der Voraussetzung, dass in beiden Schreibprozessen unterschiedliche Attributwerte ge-

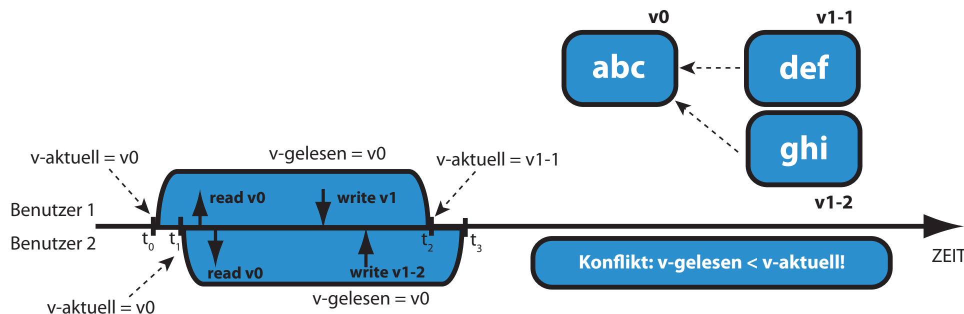


Abbildung 2.4: MVCC: Versionskonflikt

ändert wurden. Ist dies nicht der Fall, und wurde ein Attribut in beiden Versionen geändert, sollte die Transaktion rückgängig gemacht und die Fehlermeldung an den Endbenutzer weitergeleitet werden.

Das MVCC-Verfahren hat zum vermehrten Einsatz der NoSQL-DBMS beigetragen, da es besonders für skalierbare Anwendungen in verteilten Systemen geeignet ist.

2.7 Map/Reduce

Der Begriff Map/Reduce bezeichnet einen Vorgang zur Berechnung und Datenaggregation, der vor allem bei der Verarbeitung großer Datenmengen Anwendung findet. Das von Google patentierte gleichnamige Framework ermöglicht nebenläufige Datenoperationen in großen Cluster-Systemen. Es existieren bereits Implementierungen in den Programmiersprachen Java, C++, Python und Erlang. Map/Reduce entstand durch die in der funktionalen Programmierung häufig angewendeten Funktionen *map* und *reduce*.

Die allgemeine Funktion setzt sich wie folgt zusammen [14]:

$$Map/Reduce : (K \times V)^* \rightarrow (L \times W)^*$$

$$[(k_1, v_1), ..., (k_n, v_n)] \longmapsto [(l_1, w_1), ..., (l_n, w_n)]$$

Definition [15]:

> *Map/Reduce ist ein Programmiermodell, welches assoziiert wird, mit der Verarbeitung großer Mengen von Daten. Der Benutzer definiert eine Map-Funktion, welche aus einer Liste von Schlüssel-Wert-Paaren (Eingabeliste) eine neue Liste von Schlüssel-Wert-Paaren (Ausgabeliste) berechnet. Anschließend werden in einer Reduce-Funktion alle Zwischenergebnisse, deren Schlüssel identisch ist, vereint. Zahlreiche Problemstellungen und Aufgaben lassen sich mithilfe dieses Modells ausdrücken und lösen.*

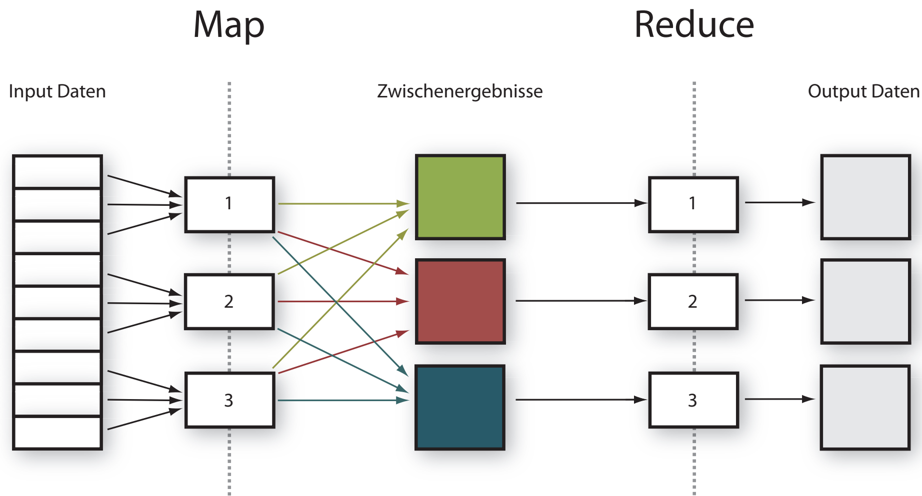


Abbildung 2.5: Map/Reduce Datenfluss

Erläuterung [14]:

- Die Mengen K und L enthalten Schlüssel, die Mengen V und W enthalten Werte.
- Alle Schlüssel $k \in K$ sind vom gleichen Typ, z. B Strings.
- Alle Schlüssel $l \in L$ sind vom gleichen Typ, z. B ganze Zahlen.
- Alle Werte $v \in V$ sind vom gleichen Typ, z. B Atome.
- Alle Werte $w \in W$ sind vom gleichen Typ, z. B Gleitkommazahlen.
- Wenn A und B Mengen sind, so ist mit $A \times B$ die Menge aller Paare (a, b) gemeint, wobei $a \in A$ und $b \in B$ (Kartesisches Produkt).
- Wenn M eine Menge ist, so ist mit M^* die Menge aller endlichen Listen mit Elementen aus M gemeint – die Liste kann auch leer sein.

In Abb. 2.5 werden der Datenfluss und die parallelen Prozesse, die im Laufe dieses Algorithmus stattfinden, grafisch veranschaulicht.

1. Die Input Daten werden auf eine Reihe von Map-Prozessen verteilt, welche die Berechnung für die vom Benutzer definierte Map-Funktion vornehmen.
2. Die Ausführung der Map-Prozesse erfolgt parallel.
3. Jeder dieser Map-Prozesse erzeugt Zwischenergebnisse (in unterschiedlichen Instanzen).
4. Der Übergang von der Map- in die Reduce-Phase findet statt.
5. Für jede Instanz der Zwischenergebnisse berechnet jeweils ein Reduce-Prozess die vom Benutzer definierte Reduce-Funktion. Diese Reduce-Prozesse werden bestenfalls ebenso parallel ausgeführt.
6. Das Resultat aus den Reduce-Prozessen sind die Output Daten.

Im folgenden Abschnitt werden die beiden Funktionen Map und Reduce im Allgemeinen genauer erläutert [14].

2.7.1 Map

Map bildet ein Paar, bestehend aus einem Schlüssel k und einem Wert v, auf eine Liste von neuen Paaren (l_r, x_r) ab, welche die Rolle von Zwischenergebnissen spielen. Die Werte x_r sind vom gleichen Typ wie die Endergebnisse w_m. Bei einem neuen Paar (l, x) verweist der von Map vergebene Schlüssel l dabei auf eine Liste T_l von Zwischenergebnissen, in welcher der von Map berechnete Wert x gesammelt wird. Das Framework ruft für jedes Paar in der Eingabeliste die Funktion Map auf.

$$Map : K \times V \rightarrow (L \times W)^*$$

$$(k, v) \longmapsto [(l_1, x_1), ..., (\mathrm{l}_{rk}, \mathrm{x}_{rk})]$$

All diese Map-Berechnungen sind voneinander unabhängig, so dass man sie nebenläufig und verteilt auf einem Cluster ausführen kann.

2.7.2 Reduce

Sind alle Map-Aufrufe erfolgt bzw. liegen alle Zwischenergebnisse in T_l vor, so ruft das Framework für jede Zwischenwertliste die Funktion Reduce auf, welche daraus eine Liste von Ergebniswerten w_m berechnet, die vom Framework in der Ausgabeliste als Paare (l, w_m) gesammelt werden.

$$Reduce : L \times W^* \rightarrow W^*$$

$$(l, [y_1, ..., \mathrm{y}_{sl}]) \longmapsto [w_1, ..., \mathrm{w}_{ml}]$$

Auch die Aufrufe von Reduce können unabhängig auf verschiedene Prozesse im Cluster verteilt werden.

2.7.3 Beispiel

Ein typisches Beispiel für eine Map/Reduce Anwendung ist die Analyse von Worthäufigkeiten in einer Dokumentensammlung [3, S. 2].

```
1  map(String key, String value):
2    // key: document name
3    // value: document contents
4    for each word w in value:
5      EmitIntermediate(w, "1");
```

```
1  reduce(String key, Iterator values):
2    // key: a word
3    // values: a list of counts
4    int result = 0;
5    for each v in values:
6      result += ParseInt(v);
7    Emit(AsString(result));
```

Die Map-Funktion erhält in diesem Beispiel als Schlüssel den Namen des jeweiligen Dokuments und als Wert dessen Inhalt. Danach wird für jedes Wort(w) ein Zwischenergebnis (1) gespeichert. In der Reduce-Funktion wird anschließend die Werteliste der einzelnen Wörter summiert und der Map/Reduce Prozess abgeschlossen.

2.8 Vector Clocks

Vor allem bei NoSQL-Systemen, die sich durch hohe Skalierbarkeit auszeichnen, können Komplikationen bei der Datensynchronisation auftreten. Durch fehlende Lock-Mechanismen und auftretende inkonsistente Zustände innerhalb des Systems bedarf es einer Lösung, welche eine gewisse Form von Ordnung erzeugt. Eine mögliche Lösung solcher Konflikte ist die zeitbasierte Synchronisation der Daten. In diesem Fall bedarf es jedoch einer voll funktionsfähigen Synchronisation der Uhren. Das Network Time Protokoll [4] kann beispielsweise zu diesem Zweck herangezogen werden. Die Entwicklung und genaue Funktionsweise dieses Timekeepers in verteilten Systemen wird in dem Paper „A Brief History of NTP Time: Confessions of an Internet Timekeeper“ von David L. Mills beschrieben [9].

2.8.1 Lamport Clocks

Um diese Art der uhrenbasierten Synchronisation und die damit einhergehenden Abhängigkeiten und Einschränkungen zeitbasierter Systeme zu vermeiden, entwickelte der bereits durch LaTeX[5] bekannte Leslie Lamport einen – unter dem Begriff Lamport Clocks bekannten – Lösungsansatz für diese Problemstellung. Lamport Clocks verzichten vollkommen auf die Verwendung von Echtzeit-Uhren. Stattdessen wird jedes Event mit einem Zähler oder Zeitstempel versehen, mit dessen Hilfe eine chronologische Ordnung der Ereignisse in einem verteilten System durchgeführt wird.

Funktionsweise

Beim Senden einer Nachricht wird diese um einen Zähler oder Zeitstempel erweitert.

Beim Empfang der Nachricht wird dieser Zähler mit dem vorhandenen Zähler des Empfängers – welcher vorerst noch um eins inkrementiert wird – verglichen. Wird dabei festgestellt, dass der erhaltene Wert des Zählers größer ist als der Vorhandene, wird dieser um eins erhöhte Wert als aktueller Indikator gespeichert.

Somit wird sichergestellt, dass dieser in jedem Fall höher ist als jener Wert des Absenders, sowie aller vorangegangenen Prozesse.

[4]http://de.wikipedia.org/wiki/Network_Time_Protocol
[5]http://www.latex-project.org/

In der Abb. 2.6 wird der Algorithmus der Lamport Clocks anhand eines einfachen Beispiels dargestellt.

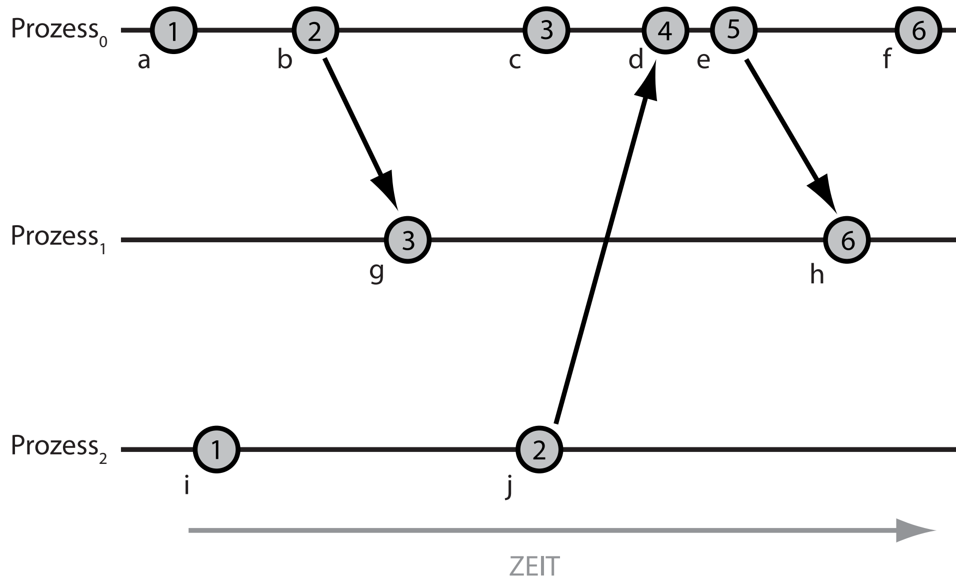


Abbildung 2.6: Lamport Clocks Beispiel

Erläuterung Lamport Clocks Beispiel

In Abb. 2.6 erhält das Event h (Prozess 1) eine Nachricht vom Event e aus Prozess 0. Das Event h erhält normalerweise – würde der Prozess 1 lokal ausgeführt – den Zähler 4. Da aber der empfangene Zähler von Event e (5) größer ist als 4, wird dieser als Wert angenommen und um eins erhöht. Event h erhält somit den Zähler 6 (5+1).

Wie in seinem, im Jahre 1978 vorgestellten, ACM-Paper „Time, Clocks, and the Ordering of Events in a Distributed System" legt er zwei grundsätzliche Konsistenzkriterien fest [8].

Schwaches Konsistenzkriterium

Wissensbasis: Event1 war Ursache von Event2
Schlussfolgerung:

$$Zeitstempel/ZählerEvent1 < Zeitstempel/ZählerEvent2$$

Starkes Konsistenzkriterium

Wissensbasis:

$$Zeitstempel/ZählerEvent1 < Zeitstempel/ZählerEvent2$$

Schlussfolgerung: Event1 war Ursache von Event2

Leslie Lamport belegt in seinem Paper, dass mithilfe dieser logischen Uhren lediglich das schwache Konsistenzkriterium erfüllt werden kann. Mithilfe der Lamport Clocks ist es also nicht möglich, Ereignisse herauszufiltern, welche kausal unabhängig stattgefunden haben. Um das Kriterium der starken Konsistenz zu erfüllen, bedarf es einer erweiterten Form: den sogenannten Versions-Vektoren.

2.8.2 Versions-Vektoren

Im Gegensatz zu den von Leslie Lamport entwickelten Lamport Clocks, erfüllen Versions-Vektoren sehr wohl das Kriterium der starken Konsistenz. Da hier nicht nur ein zustandsloser Zähler mitgespeichert wird, sondern von jedem Prozess auch ein Zeitstempel. Diese sich sammelnden Werte werden als Vektoren bei jedem Ereignis mitgesendet, womit auch kausale Zusammenhänge nachvollziehbar werden. Der zuletzt gespeicherte Wert wird automatisch zurückgeliefert.

Funktionsweise

Versions-Vektoren bestehen aus einer Liste bzw. Vektoren deren Elemente sich wie folgt zusammensetzen:

```
SenderID x Zeitstempel
```

Die SenderID kann entweder eine IP-Adresse, MAC-Adresse oder eine Prozess-ID sein.

In NoSQL-Datenbanksystemen wie Riak oder Amazons Dynamo werden Versions-Vektoren eingesetzt, um dem Client trotz häufiger und mehrfacher Replikationsvorgänge und Updates im Datenbank-Cluster eine Möglichkeit zu geben, die aktuelle Version eines Datensatzes zu erhalten. Dazu kann der Algorithmus als Pseudo-Code wie folgt beschrieben werden.

Nachricht senden

```
1 Uhr[PID]= Uhr[PID]+1;
2 Zeitstempel= Uhr;
3 sende(Nachricht,Zeitstempel);
```

Wird eine Nachricht gesendet, wird das Element mit der eigenen SenderID um eins erhöht. Zu Beginn werden alle Zähler des Vektors mit Null initialisiert. Danach wird der Vektor, mit dem eigenen Timestamp der zu verschickenden Nachricht, hinzugefügt.

Nachricht empfangen

```
1 (Nachricht,Zeitstempel)= empfange();
2 Uhr[PID]= Uhr[PID]+1;
3
```

```
4 for (Prozesse P) do begin
5     Uhr[P]= max(Uhr[P],Zeitstempel[P]);
6 end;
```

Beim Erhalt einer Nachricht erhöht der Empfänger seinen Zähler um eins. Danach wird aus dem eigenen und dem empfangenen Vektor, für jedes enthaltene Element das Maximum gebildet und anschließend der eigene Vektor wieder auf den aktuellen Stand gebracht. Jene Elemente der Uhr, von deren Prozessen noch keine Nachricht erhalten wurde, werden mit Null gekennzeichnet.

Dieser Algorithmus beinhaltet lediglich die Annahme, dass eine Nachricht erst zu einem späteren Zeitpunkt beim Empfänger ankommt, als sie gesendet wurde. Versions-Vektoren lassen es dadurch zu, sich weitestgehend von der Synchronisation realer Uhren zu lösen.

Wichtig zu erwähnen ist aber auch, dass Versions-Vektoren keine auftretenden Update Konflikte lösen können, sondern lediglich auf diese hinweisen. Die Problemlösung bedarf einer Begutachtung der Clientseite, von welcher entsprechende Schritte in die Wege geleitet werden müssen.

In Abb. 2.7 wird dieser Algorithmus mit einem auftretenden Konflikt dargestellt.

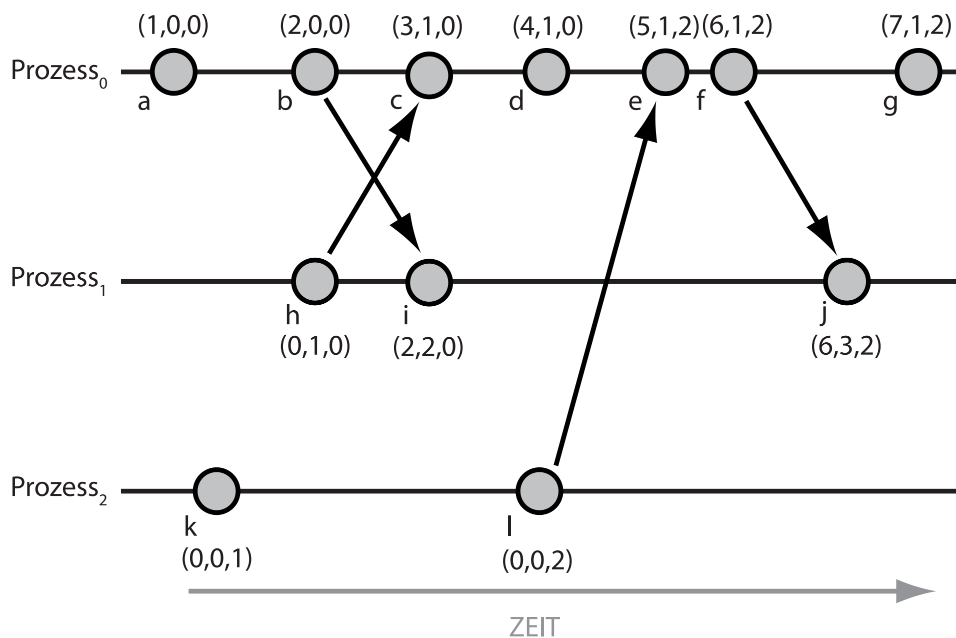


Abbildung 2.7: Vector Clocks Beispiel

Erläuterung Vector Clocks Beispiel

Anhand der Darstellung 2.7 lässt sich beispielsweise eindeutig ausmachen, dass das Event e vor dem Event l stattgefunden hat, da jeder Wert des Vektors e größer oder gleich jenem in Vektor l ist:

$$0 \leq 6, 0 \leq 1, 2 \leq 2$$

Betrachtet man jedoch die Events c und i kann man hier einen Konflikt feststellen. Die Tatsache, dass diese beiden Events konkurrieren, lässt sich durch folgenden Vergleich feststellen:

$$c \geq i \Rightarrow 3 \geq 2$$

$$i \geq c \Rightarrow 2 \geq 1$$

Daher kann man auch nicht mit Sicherheit definieren, in welcher Reihenfolge die Events c und i stattgefunden haben. Dieser Konflikt sollte an den Client mittels einer Fehlermeldung weitergeleitet und entsprechend behandelt werden.

2.9 Paxos

Um in einem verteilten System – trotz Teilausfällen – ein gewisses Maß an Stabilität zu gewährleisten, wird oftmals das fehlertolerante, skalierbare Commit-Protokoll Paxos eingesetzt. Vor allem in Key-Value-Stores wie Scalaris, aber auch in der Google Datenbank BigTable wird dieser Algorithmus bereits verwendet. Paxos wurde dahingehend konzipiert, dass es einen teilweisen Ausfall replizierender Knoten im Cluster toleriert und versucht die Konsistenz mittels eines Konsens zu erhalten. Diese Consensus-Protokolle führen dazu, dass – auch bei Störungen im System – Transaktionen erfolgreich durchgeführt werden können. Grundvoraussetzung ist allerdings, dass eine Mehrheit der Knoten im System läuft.

Kapitel 3

NoSQL-Datenbanksysteme

Es wurden bereits mehr als 100 verschiedene Datenbanken unter dem Paradigma von NoSQL entwickelt. Diese Systeme können in die vier Hauptkategorien

- Wide Column Stores/Column Families,
- Document Stores,
- Key-Value Stores,
- und Graphdatenbanken

gegliedert werden.

In dieser Masterarbeit wird vorwiegend auf Datenbanken eingegangen, die der Kategorie der Wide Column Stores/Column Families zugeordnet werden. Der Überblick über die verschiedenen Systeme und Kategorien dient lediglich dazu, die Entwicklung und den Fortschritt von NoSQL im Laufe der letzten Jahre aufzuzeigen.

3.1 Wide Column Stores/Column Families

Die NoSQL-Datenbanksysteme der Kategorie Wide Column Stores/Column Families zeichnen sich dadurch aus, dass sie Daten spaltenweise ablegen. Diese spaltenorientierte Arbeitsweise unterscheidet sich grundlegend von den herkömmlichen relationalen Systemen, welche einem zeilenorientierten Schema folgen. Aufgrund dieser Tatsache eignen sich Wide Column Stores besonders zur Aggregation großer Datenmengen – nur eine oder wenige Spalten betreffend. Vor allem im Bereich Web Analytics ist dies durchaus zutreffend, weshalb für die Implementierung des Prototyps auf zwei dieser Datenbanksysteme zurückgegriffen wurde. Ein spaltenorientiertes Schema bringt allerdings nicht nur Vorteile mit sich. Vor allem spezielle Datenabfragen können sich sehr schnell als durchaus komplex herausstellen. Man ist gut beraten, sich bereits bei der Erstellung des Datenbankschemas ausreichend Gedanken über anstehende Schreib- und Leseoperationen zu machen. Die generelle Ar-

beitsweise, welche mit diesen Systemen einhergeht, unterscheidet sich grundlegend von jener der SQL-Datenbanken. Hier wird stets im Vorhinhein das Datenmodell (Schema) genau definiert und meist sogar mit speziellen SQL-Modelling-Tools ein Entity-Relationship-Diagramm für die jeweilige Anwendung erstellt. Apache Cassandra, HBase, LucidDB und BigTable sind nur einige Vertreter dieser Kategorie.

3.2 Document Stores

Die ersten Ansätze Daten nicht-relational zu speichern, sondern in einer dokumentenbasierten und strukturierten Form, findet man bereits in den 80er Jahren. Blickt man auf die Entstehungsgeschichte von NoSQL, stößt man häufig auf die verteilte, dokumentenbasierte Datenbank Lotus Notes [16]. Eng angelehnt an das Prinzip der Key-Value Stores werden, zu einem definierten Schlüssel, strukturierte Dokumente jeglicher Form hinterlegt. Dazu zählen beispielsweise bekannte Formate wie XML, JSON, YAML oder RDF. Document Stores verzichten vollkommen auf referentielle Integritäten oder Schemarestriktionen. Es wird hier die Verantwortung vollkommen auf die Seite der Applikation verlagert. Besonders in Cloud-Infrastrukturen oder hoch skalierbaren Web 2.0 Anwendungen bieten sich zahlreiche Anwendungsfälle für diese NoSQL-Datenbanken. Bekannte Vertreter dieser Sparte sind CouchDB, MongoDB, Terrastore, ThruDB, OrientDB und RavenDB.

3.3 Key-Value Stores

Vor allem das sehr rudimentär aufgebaute Schema von Key-Value Stores deutet auf eine hohe Skalierbarkeit hin – was durchaus eines der ausschlaggebendsten Argumente für die Verwendung dieser Datenbanken ist. Wie dem Namen bereits zu entnehmen, setzt sich das Schema lediglich aus einer Ansammlung von Key-Value-Paaren zusammen. Unternehmen wie Facebook, Amazon, Twitter oder LinkedIn, die Ihre rasch wachsenden Datenmengen auf riesige Cluster-Systeme über globale Rechenzentren verteilen, bevorzugen für manche Anwendungen Key-Value Stores gegenüber relationalen Systemen. Besonders der Reiz eines unkomplizierten, horizontalen Skalierungsmechanismus, hat diese Datenbanksysteme sehr beliebt gemacht und daher existieren auch bereits zahlreiche Vertreter dieser Sparte. Zu den bekanntesten Key-Value Stores gehören Redis, Chordless, Riak, Membase, Amazon (Dynamo und S3), Voldemort und Scalaris.

3.4 Graphdatenbanken

Beim Versuch baumartige Strukturen in relationalen Datenschemen darzustellen, stößt man oftmals an seine Grenzen. Es ist zwar durchaus möglich,

Daten, die als Graphen repräsentiert werden, in einem RDBMS zu speichern, jedoch gestaltet sich die Abfrage als sehr aufwändig. Zahlreiche join-Verknüpfungen gehen zu Lasten der Performance und es muss mit längeren Reaktionszeiten gerechnet werden. Um diesem Problem entgegenzuwirken, entwickelte man die sogenannten Graphdatenbanken. Diese verwenden eigene Traversionsalgorithmen, um beispielsweise Pfadanalysen durchzuführen oder den kürzesten Weg zu einem Punkt zu finden. Vor allem in den Bereichen Location Based Services, Semantic Web, Page Ranking und Artificial Intelligence werden Graphdatenbanken häufig eingesetzt. Neo4j und SonesDB zählen zu den namhaftesten DBMS dieser Kategorie.

Zwar existieren noch vergleichsweise wenige Datenbanken dieser Kategorie, aber aufgrund ihrer vielseitigen und sehr nutzbringenden Einsatzmöglichkeiten entwickelten sie sich mittlerweile zu den fast wichtigsten Vertretern der NoSQL-Bewegung. Dies ist vor allem auch auf die rasche Entwicklung der Smartphones und der damit verbundenen ortsbasierten Services zurückzuführen.

Kapitel 4

Analytics Prototyp

4.1 Exkurs Web Analytics

In Zeiten, in denen der Webauftritt eines Unternehmens immer mehr an Bedeutung zunimmt, beginnt auch die Analyse dieser Präsenzen im World Wide Web ein wichtiger Bestandteil von Geschäftsmodellen zu werden. Es existieren unter dem Begriff Web Analytics bereits zahlreiche Tools, die es jedem ermöglichen das Besucherverhalten auf seinen Websiten zu analysieren und auszuwerten (z. B in Form von Diagrammen). Zahlreiche Unternehmen, aber auch Privatpersonen nutzen bereits diese Werkzeuge, um anhand der Ergebnisse einer solchen Web Analyse ihr Online-Angebot zu optimieren oder sogar effizientere Geschäftsmodelle zu schaffen.

4.2 Allgemeine Anforderungen

Im Zuge dieser Masterarbeit entstand ein völlig proprietärer Prototyp, der einige Grundfunktionalitäten der Web Analyse bietet. Das Hauptaugenmerk lag durchaus nicht auf der Implementierung einer Vielzahl an Features. Viel wichtiger war es, herauszufinden, wie man diese sogenannten Tracking-Daten – Daten die vom Besucher einer Website gesammelt werden – optimal speichert und effizient auswertet. Die Problematik der zur Verfügung stehenden Web Analytics Tools besteht darin, dass sie meist für einzelne Anwendungsfälle überdimensioniert sind oder einfach spezielle Anforderungen nicht erfüllen können.

Diese Arbeit beleuchtet hauptsächlich den Part der Implementierung in Verbindung mit einer enstprechenden Datenbanklösung. Dabei wird, abgesehen von den üblichen Kennzahlen einer Web Analytics Anwendung (Besucheranzahl, Drop-Out-Quote, Conversion-Funnel), ebenfalls versucht, individuelle Analysen vorzunehmen. In weiterer Folge sollen, anhand dieser Auswertungen, Berichte erstellt werden, die Aufschluss über Entwicklungen und Trends – den Webauftritt betreffend – geben können.

Dabei erstrecken sich die Anforderungen an den Analytics Prototypen über mehrere Ebenen. Zum einen bedarf es einer kostengünstigen Hardware-Lösung, damit ein längerfristiger Betrieb der Anwendung gesichert werden kann. Die Analyse hochfrequentierter Webseiten sollte ebenso gewährleistet sein. Diese Punkte verlangen eine gut durchdachte Lösung betreffend Data Storage. Auch die Erzeugung eines starren Systems sollte weitestgehend vermieden werden, da der Bedarf nach Erweiterungen oder Adaptierungen der zu analysierenden Themen durchaus gegeben ist. Der Zugriff und die Analyse der Daten soll schnellstmöglich durchführbar sein und keine langen Wartezeiten oder regelmäßige Ausfälle mit sich bringen. Bestimmte Kennzahlen erfordern eine echtzeitnahe Auswertung der Daten, um den Reaktionszeitraum des Benutzers, bestimmte Veränderungen betreffend, auf ein Minimum einzugrenzen. Wobei diese Analysen nicht nur auf Webseiten eingeschränkt werden sollten, sondern bis zu Systemeigenschaften oder internen Daten ausgedehnt werden können.

Die Ergebnisse der Analysen spiegeln sich in Form von Diagrammen oder Kennzahlentabellen wider und werden vom Benutzer anhand diverser Filtermechanismen angepasst.

4.3 Auswahl Data Storage

Bei der Auswahl einer geeigneten Data Storage Lösung für den Prototypen wurde sehr schnell klar, dass herkömmliche relationale DBMS nicht unbedingt für diesen Anwendungsfall geeignet sind. Das strikte ACID Prinzip und die einhergehenden Schemarestriktionen werfen einfach zu viele Widersprüche auf. Die Analytics Anwendung soll jederzeit erweiterbar sein. Auch was nachträgliche Adaptierungen am Datenbankschema betrifft, dürfen keine Einschränkungen entstehen. Die Tatsache der großen Datenmengen im Web Analyse Bereich lässt sofort den Gedanken einer verteilten Datenbanklösung aufkommen. Allerdings sollte man ebenfalls in Betracht ziehen, dass der Betrieb einer solchen Anwendung durchaus zu erheblichen Komplikationen und administrativen Aufwänden führen kann. Aufgrund der bereits erwähnten Restriktionen relationaler Datenbanksysteme wurden diese für die Verwendung in dieser Masterarbeit ausgeschlossen.

Ziel war es nun eine geeignete Alternative zu finden, welche erheblich weniger Einschränkungen mit sich bringt und sich generell besser für die Implementierung des Prototypen eignet. Somit wurde die Tür zur großen Gegenbewegung der relationalen Datenbankmanagementsysteme aufgestoßen, die mittlerweile unter dem Begriff NoSQL in aller Munde ist. Unter den bereits über 100 existierenden NoSQL-Systemen galt es nun ein geeignetes DBMS herauszufiltern.

Das Datenschema des Prototypen soll einfach gehalten werden und – zugunsten der Performance – ohne Relationen auskommen. Dazu kommt noch

die Tatsache, dass es sich um eine Analyse-Anwendung handelt, was vor allem die Stichworte Datenaggregation und Big Data aufwirft. Dadurch wurde die Auswahl der NoSQL-DBMS auf die beiden Kategorien Wide Column Stores und Key-Value Stores eingeschränkt. Zu den bekanntesten Vertretern der Wide Column Stores kann man wohl Googles BigTable zählen. Diese NoSQL-Datenbank ist im Stande mehrere Milliarden Suchanfragen pro Tag zu verarbeiten und das bei sehr schnellen Reaktionszeiten und nahezu ständiger Verfügbarkeit. Ein zweiter großer Name, den man unter den NoSQL-DBMS findet, ist Amazon. Diverse Web-Services des E-Commerce Anbieters verwenden für die Speicherung der Daten einen Key-Value Store namens Dynamo.

Die genannten Systeme sind aufgrund ihrer Architektur besonders dazu geeignet, große Mengen an Daten in relativ kurzen Zeiträumen zu verarbeiten. Argumente wie horizontale Skalierung, effiziente Replikationsmechanismen, hohes Maß an Ausfallsicherheit und schnelle Reaktionszeiten führten schließlich zu der Entscheidung, ein System zu verwenden, welches architektonisch jenem von Google oder Amazon ähnelt. Die Datenbanksysteme der beiden Unternehmen sind jedoch nicht für den freien Markt zugänglich und stehen lediglich als cloudbasierte Dienste zur Verfügung. Daher galt es eine Lösung zu finden, die völlig frei in der prototypischen Implementierung einsetzbar ist. Nach eingehender Recherche fiel die Auswahl auf zwei Apache Incubator Projekte namens Cassandra und HBase. Cassandra kann man als eine Art Hybrid betrachten. Das Datenmodell ähnelt im Aufbau Googles Big Table [2], hat jedoch bei Themen wie Partitionierung, Replizierung und Konsistenz, Amazons Dynamo zum Vorbild [4]. Diese Hybrid-Lösung bietet ideale Voraussetzungen zur Speicherung der Tracking-Daten. Neben Cassandra wurde noch die, auf dem Apache Hadoop Projekt[1] basierende, Datenbanklösung HBase zur Umsetzung des Prototypen herangezogen. Da Hadoop eine Open Source Implementierung von BigTable darstellt und diese in Kombination mit dem Map/Reduce Framework und HBase ebenfalls eine ideale Grundlage zur Analyse von Daten bildet, wurde die Auswahl um HBase erweitert.

Die Anbindung des Prototypen an die Datenbanken erfolgt in Form von Data Access Objects (DAOs)[2]. Diese fungieren als Ausgangspunkt für die Evaluierung von Cassandra und HBase.

4.4 Architektur Analytics Prototyp

Der Aufbau und die Kommunikation der einzelnen Komponenten des Prototypen ist in Abb. 4.1 dargestellt.

In diesem Abschnitt werden die einzelnen Komponenten und Schnittstel-

[1] http://hadoop.apache.org/

[2] http://de.wikipedia.org/wiki/Data_Access_Objects

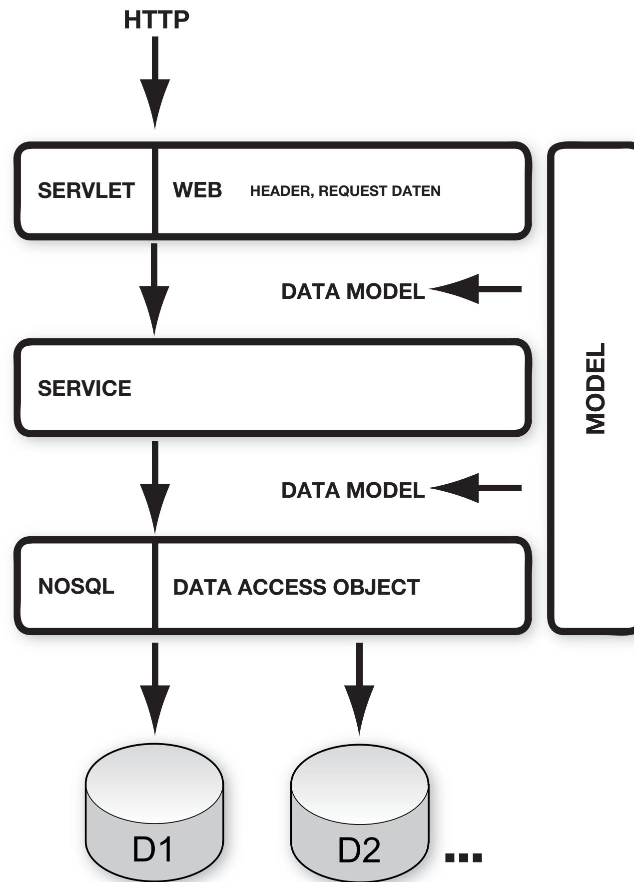


Abbildung 4.1: Architektur Analytics Prototyp

len des Analytics Prototypen im Detail erläutert.

4.4.1 Tracking

Um Daten für den Analytics Prototypen zu sammeln wurde ein entsprechender Tracking-Code erstellt, der auf einer Website eingefügt wird und – je nach Definition – die entsprechenden Daten sammelt und in der Datenbank speichert. Dabei wird zwischen clientseitigen und serverseitigen Tracking-Daten unterschieden. Die Informationen der Clientseite werden in einen Request verpackt, welcher an ein entsprechendes Java Servlet übermittelt wird. Dort werden diese Parameter ausgelesen und ein Tracking-Objekt erstellt. Dieses wird in weiterer Folge um die serverseitigen Tracking-Daten erweitert und letztendlich in der Datenbank abgelegt.

4.4.2 Datenbank

In der jeweiligen NoSQL-Datenbank (Cassandra/HBase) werden zunächst die vom Tracker gesammelten Daten im Rohformat gespeichert. Dies bedeutet, dass jeder Request an den Server einen eigenen Eintrag in der Datenbank erhält. Allgemein ist das Schema so definiert, dass diese Einträge unter einem eindeutigen Zeilenschlüssel abgerufen werden können und für jede Eigenschaft eine entsprechende Spalte in der Datenbank erstellt wird. Der Name der jeweiligen Spalte kann als Schlüssel angesehen werden, dem ein Wert zugeordnet wird. Zugriff auf diese Werte erhält man durch eine Kombination aus Schlüssel und Spaltenname. In dieser Masterarbeit wird jedoch vorwiegend das Map/Reduce Framework von Hadoop zur Datenabfrage und -aggregation herangezogen.

4.4.3 Data Access Objects (DAOs)

Data Access Objects, oder kurz DAOs genannt, stellen die Schnittstelle zur Datenbank dar. Um diese flexibel zu gestalten, wurde ein Interface erstellt, welches sämtliche Funktionen enthält. Dessen Implementierung erfolgt mittels der jeweiligen API individuell und ist somit einfach austausch- und erweiterbar. Durch die Vielzahl an Schnittstellen der Datenbanken ist dieser Aufbau sinnvoll und bietet zahlreiche Möglichkeiten hinsichtlich der Durchführung von Tests und Evaluierungen. Die Hauptfunktion eines Data Access Objects besteht somit einzig und allein in der direkten Kommunikation mit der Datenbank. Dies betrifft vor allem die sogenannten CRUD-Operationen. Alle weiteren Funktionalitäten und Datenmanipulationen wurden in den Servicebereich ausgelagert.

4.4.4 Service-Schnittstellen

Die Services des Analytics Prototypen umfassen sämtliche Operationen an den Daten, welche von den Data Access Objects zur Verfügung gestellt werden. Dabei unterteilt sich der Servicebereich in zwei Teile. Der erste Teil beinhaltet das Backend und das Web-Modell wurde in den zweiten Part ausgelagert.

4.4.5 Frontend

Das Frontend des Analytics Prototypen wurde mittels HTML, CSS und Javascript erstellt und dient einzig der grafischen Darstellung der Datenauswertungen. Obwohl von großem Wert für die gesamte Applikation, bedarf es im Zuge dieser Masterarbeit keiner genaueren Erläuterung der betreffenden Technologien und Implementierungen. Der Inhalt dieses Werkes ist auf den Bereich rund um die beiden NoSQL-Datenbanken Cassandra und HBase und deren Einsatzmöglichkeiten im Bereich Web Analytics fokussiert.

4.5 Projektstruktur

4.5.1 Allgemein

Die Implementierung des Analytics Prototypen erfolgte anhand der Programmiersprache Java. Das Deployment wurde auf einem Apache Tomcat Server (Version 6.0) durchgeführt.

In diesem Abschnitt erfolgt eine Aufstellung der Projektstruktur.

Build Projekt Dieses Projekt umfasst den Build und das Deployment der gesamten Applikation.

MapReduce Hier findet die Implementierung des Map/Reduce Algorithmus statt. Zur effizienten Datenanalyse wurden entsprechende Jobs mithilfe der jeweiligen Map/Reduce Integration erstellt, welche in regelmäßigen chronologischen Abständen eine Aggregation der gesammelten Tracking-Daten vornehmen.

StatisticServices Dieser Part beinhaltet die Implementierung der DAOs und Service-Schnittstellen der Datenbanken Cassandra und HBase.

WebAnalytics Das Frontend des Analytics Prototypen umfasst die Darstellung der Datenauswertungen anhand von Diagrammen und Kennzahlentabellen.

WebPublic Dieser Part der Applikation beinhaltet den Tracking-Code und das Java Servlet, welches die einzelnen Request-Parameter einliest und ein entsprechendes Tracking-Objekt erstellt.

Im folgenden Abschnitt werden nun der Projektaufbau und das Datenbankschema im Detail erläutert.

4.5.2 Datenbankschema Allgemein

Wie bereits in der theoretischen Einführung beschrieben, unterscheidet sich das Datenbankschema von NoSQL-Systemen meist grundlegend von jenem der relationalen Datenbanken. Man kann sogar so weit gehen, zu behaupten, dass vor allem bei Wide Column Stores das Datenbankschema erst sinnvoll erstellt werden kann, sobald man sich darüber Klarheit verschafft hat, welche Daten in welcher Form abzufragen sind. Wogegen bei relationalen Datenbanken, dies genau der umgekehrte Prozess ist. Für den Analytics Prototypen wurden daher zuerst folgende Abfragekriterien definiert:

- Hauptaugenmerk auf Aggregation der Daten mittels Map/Reduce,
- Daten auf einen bestimmten Zeitraum einschränkbar,
- Daten auf die definierten Eigenschaften einschränkbar,
- Daten individuell auf einen Benutzer einschränkbar,
- Daten individuell auf eine Website einschränkbar.

4.5.3 Wide Column Stores mit Map/Reduce

Durch die Integration des Map/Reduce Framworks in Apache Hadoop besteht die Möglichkeit, diesen effizienten Algorithmus auch in Verbindung mit NoSQL-Datenbanken wie Cassandra oder HBase zu nutzen. Die Aggregation von Daten kann dadurch um einiges schneller vollzogen werden, da mithilfe von Map/Reduce nebenläufige Berechnungen über große Datenmengen durchgeführt werden. Im Folgenden werden die einzelnen Phasen des Algorithmus nochmals am Beispiel eines Wide Column Stores erklärt.

- Eine oder mehrere Spalten der Datenbank werden als Input Daten defniert.
- Diese Input Daten werden auf eine Reihe von Map-Prozessen verteilt, welche die Berechnung für die vom Benutzer implementierte Map-Funktion vornehmen.
- Die Ausführung der Map-Prozesse erfolgt parallel.
- Jeder dieser Map-Prozesse erzeugt Zwischenergebnisse (in unterschiedlichen Instanzen).
- Der Übergang von der Map in die Reduce-Phase findet statt.
- Für jede Instanz der Zwischenergebnisse berechnet jeweils ein Reduce-Prozess die vom Benutzer implementierte Reduce-Funktion. Diese Prozesse werden bestenfalls ebenso parallel ausgeführt.
- Das Resultat aus den Reduce-Prozessen sind die Output Daten, welche wiederum in der Datenbank gespeichert werden.

Der Ansatz von Map/Reduce erfordert ein gezieltes Umdenken, da es sich hier um Datenanalysen in verteilten Systemen handelt, die parallel ausgeführt werden. Um dies zu bewerkstelligen und eine angemessene Integrationsmöglichkeit in Anwendungen zu finden, bedarf es spezieller Tools wie Apache Hadoop, Pig oder Hive [12].

Kapitel 5

Implementierung Apache Cassandra

5.1 Apache Cassandra Allgemein

Eines der bekanntesten NoSQL-Datenbankverwaltungssysteme, das den Wide Column Stores zugeordnet wird, ist Apache Cassandra. Ursprünglich entwickelt von Avinash Lakshman und Prashant Malik bei Facebook, wurde das Projekt im März 2009 von der Apache Software Foundation als Unterprojekt in den Apache Incubator aufgenommen. Mittlerweile in der Version 1.1.5 veröffentlicht, hat es den Status eines Top-Level-Projektes erreicht und wird bereits von namhaften Unternehmen wie Reddit, Twitter oder Digg eingesetzt. Cassandra ist hervorragend für den Big Data Bereich geeignet, was vorwiegend auf die verteilte Architektur zurückzuführen ist. Wie in ihrem Blog angeführt[1], verwendet die Micro-Blogging Plattform Twitter Cassandra beispielsweise für Real-Time Analysen, Geolocation und diverse Data-Mining Aktivitäten – die gesamte Community betreffend. Diese Aufgaben erfordern die Verarbeitung einer riesigen Menge an Daten. Aufgrund der verteilten Architektur und der daraus entstehenden Ausfallsicherheit scheint Cassandra hier die idealen Voraussetzungen mitzubringen. Darüber hinaus zielte Apache mit dieser Datenbank darauf ab, den Betrieb mit kostengünstiger Standard Hardware zu sichern. Cassandra zeichnet sich vor allem durch effiziente Schreiboperationen aus, die Performance der Lesezugriffe lässt noch etwas zu wünschen übrig [7].

5.1.1 Datenmodell

Obwohl unter die Kategorie Wide Column Store einzuordnen, unterscheidet sich Cassandra grundlegend vom Prinzip der Spaltenorientierung. Einfluss auf die Datenorganisation hat vor allem Googles BigTable genommen, des-

[1] http://blog.twitter.com/

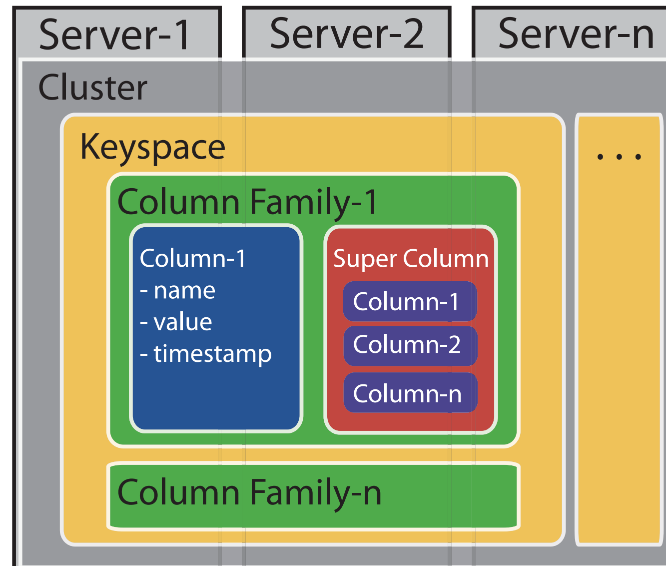


Abbildung 5.1: Cassandra Datenmodell

sen Architektur als „sparse, distributed multidimensional sorted map“ beschrieben werden kann. Darunter versteht man mehrdimensionale Tabellen mit folgendem Aufbau [5]:

$$n*[Domain/Keyspace] \times [Item/ColumnFamily] \times [Keyx]n*[Key+Value]$$

Wie in Abb. 5.1 dargestellt, werden bei Cassandra Datenbanken als Keyspaces bezeichnet, welche sogenannte Column Families beinhalten. Die Daten werden, den Eigenschaften entsprechend, spaltenweise abgelegt und über einen Zeilenschlüssel eindeutig identifiziert. Diese Spalten werden, wie in obiger Definition angegeben, nach einem Key-Value Prinzip abgelegt. Zusätzlich zu dem Wert eines Keys wird auch noch ein Zeitstempel automatisch generiert und abgespeichert. Eine Gliederung der Spalten in Super Columns erlaubt darüber hinaus eine speziellere Definition des Schemas und die Einführung einer flachen, hierarchischen Ordnung. Einstellungen wie Consistency-Level oder Replikationsfaktor können über die entsprechende Konfigurationsdatei oder bereits bei der Erstellung des Datenbankschemas vorgenommen werden.

5.1.2 Datenoperationen

Cassandra ist – wie bereits erwähnt – für eine Integration in verteilten Systemen geeignet und bietet vor allem durch die Eigenschaften der horizontalen Skalierbarkeit, hoher Fehlertoleranz, sowie performanter und effizienter

Schreib- und Lesevorgänge, die idealen Voraussetzungen für den Betrieb einer Big Data Anwendung. Zur Ausführung der üblichen CRUD-Operationen stellt Cassandra standardmäßig eine Low-Level API namens Thrift zur Verfügung. Das Erstellen eines Datenbank-Schemas, sowie herkömmliche Datenbankoperationen können mithilfe dieser API über eine Konsole kommuniziert werden. Aufgrund der Tatsache, dass Cassandra in der Programmiersprache Java implementiert wurde, ist die Entwicklung von Anwendungen in einer IDE wie Eclipse ebenfalls ohne großen Aufwand möglich. Für den Einsatz von Cassandra in komplexeren Anwendungen ist es jedoch anzuraten auf eine High-Level API wie beispielsweise Hector zurückzugreifen. Ab der Version 0.8 steht außerdem ein, der Abfragesprache SQL sehr ähnliches, Konstrukt namens CQL (Cassandra Query Language), sowie der Zugriff auf Werte mittels sogenannter Secondary Indexes, zur Verfügung. Besonders der Punkt Datenaggregation wird bei Cassandra groß geschrieben. Zusätzlich zum spaltenorientieren Schema wurde ab der Version 0.6 das bekannte Big Data Framework Hadoop integriert, welches nebenläufige Berechnungen mithilfe eines Map/Reduce Algorithmus ermöglicht und somit einen großen Fortschritt im Umgang mit großen Datenmengen und deren Verarbeitung mit sich bringt.

5.1.3 Distribution

Der Einsatz von Cassandra als verteiltes System in einem großen Cluster wurde bereits erwähnt. Replikationsmechanismen und Datenkonsistenz sind frei konfigurierbar und im Gegensatz zu SQL-Datenbanken ohne großen Administrationsaufwand zu bewerkstelligen. Diese Einstellungen sollten jedoch wohldurchdacht sein und den Anforderungen der jeweiligen Anwendung gerecht werden. So ist es beispielsweise durchaus denkbar, eine strikte Konsistenz in einem Cluster einzuführen, jedoch wird dies durchaus negative Folgen für die Erreichbarkeit des Systems haben. Abhängig vom Replikationsfaktor blockiert Cassandra solange, bis alle Updates erfolgreich auf den gesamten Knoten im System durchgeführt wurden. Bezugnehmend auf das Cap-Theorem von Eric Brewer bedarf es in vielen Fällen eines Trade-Offs. Um einen Knoten im Cluster hinzuzufügen, reicht es aus, diesen in der Konfigurationsdatei (cassandra.yaml) bekanntzugeben. Das Erweitern oder auch Verkleinern des Netzwerks kann während des Betriebs erfolgen und erfordert auch keinen Neustart des Systems. Die Bekanntmachung des neuen Cluster-Knotens erfolgt nach dem Gossip Prinzip, welches die Informationen über das neue Mitglied im Netzwerk von Knoten zu Knoten weiterverteilt und bekanntgibt, bis alle Einheiten des Clusters dies vollständig registriert haben. Cassandra verzichtet in seiner Verteilungsstrategie auf das herkömmliche Master/Slave Prinzip und stellt jeden Knoten im Netzwerk völlig gleich. Durch diese Strategie vermeidet man die Entstehung eines möglichen Single Point Of Failure. Darunter versteht man Systemausfälle eines Master-Knoten, welche dazu führen, dass auch die ihm untergeordneten Einheiten

nicht mehr erreichbar sind und es zu einem Totalausfall kommt. Um eine gleichmäßige Aufteilung der Daten in Cassandra zu erreichen, werden diese nach dem Prinzip des Consistent-Hashing verteilt. Zur Administration und Überwachung des Cassandra-Clusters empfiehlt es sich auf die integrierten Monitoring-Tools JMX oder nodetool zurückzugreifen. Folgende Eigenschaften sind dem Monitoring zuzurechnen:

- Dumping,
- Im- und Export von Daten,
- Latenzzeiten,
- Zeiten pro Lese- und Schreiboperation,
- Verteilung der Daten im Cluster,
- Verfügbarkeit der Knoten,
- Auslastung der Knoten.

5.2 Implementierungsdetails Prototyp

5.2.1 Datenbankschema Prototyp

Anhand der festgelegten Kriterien wurde für die Datenbank Apache Cassandra folgendes Schema entworfen 5.2:

Keyspace: Statistics
Column Families:

tracking_request Enthält sämtliche Rohdaten, welche sich aus einem Zeileneintrag pro abgesetztem Request zusammensetzen.

all_visits Eine Auswertung aller Besuche nach Zeitraum, Benutzer und Website sowie einer definierten Eigenschaft (z. B Browser, Betriebssystem).

tracked_visits Die Auswertung aller eindeutigen Besucher nach Zeitraum, Benutzer und Website sowie einer definierten Eigenschaft (z. B Browser, Betriebssystem).

visit_times Eine Timeline, die die Kalkulation zur durchschnittlichen Besuchszeit eines Benutzers ermöglicht.

dashboard Die Monitoring Kennzahlen, welche Auskunft über die Anzahl der aktuell angemeldeten und aktiven Benutzer geben.

5.2.2 API Schnittstellen

Das Fehlen einer wohldefinierten Abfragesprache, wie man sie bei relationalen Datenbanken in Form der Structured Query Language (SQL) findet, verlangt nach einem Umdenkprozess, was NoSQL-DBMS betrifft. Daher werden in diesem Abschnitt die zur Verfügung gestellten APIs vorgestellt, welche es erlauben, ein Cassandra Schema zu erstellen und Datenbankoperatio-

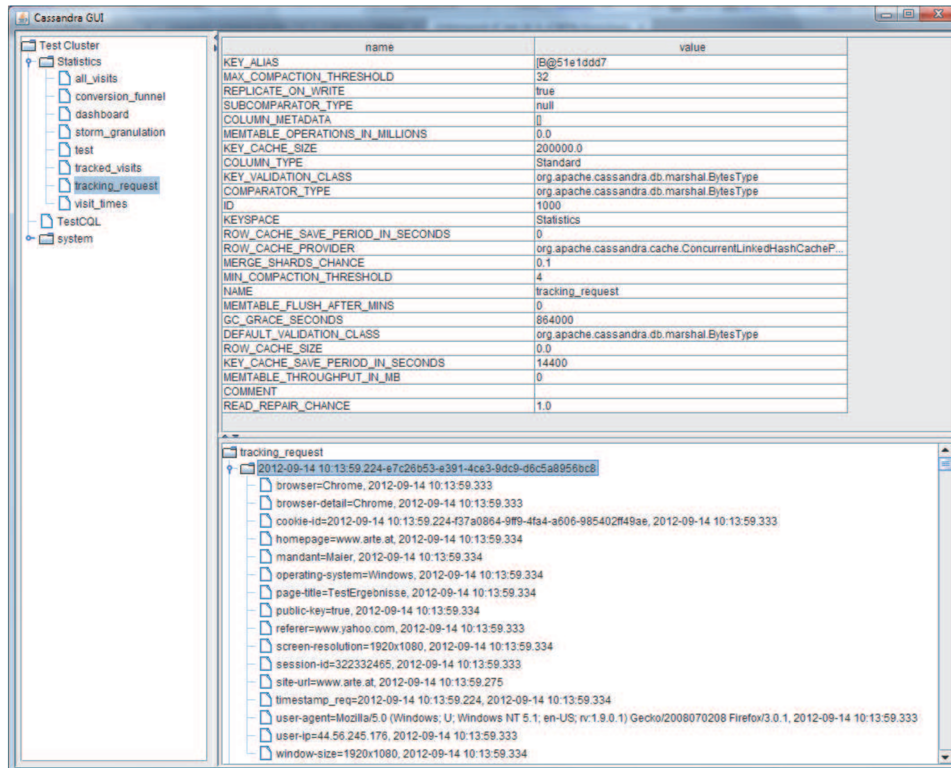


Abbildung 5.2: Cassandra Datenbankschema

nen durchzuführen. Weiters wurden diese Schnittstellen zur Implementierung der Data Access Objects verwendet. Kurze Code-Snippets dienen zur Veranschaulichung, wobei hier lediglich ein Auszug aus der Schemaerstellung sowie der Datenbankoperationen gegeben wird.

Cassandra Thrift API

Die Low-Level API stellt eine Java Anbindung zur Verfügung, kann aber ebenfalls über ein interaktives Interface namens CLI für Datenbankoperationen verwendet werden. Bei der Implementierung des Prototypen wurde Thrift lediglich zur Erstellung des Datenbankschemas verwendet. Aufgrund der eingeschränkten Funktionalitäten und umständlicher Serialisierungsvorgänge fiel der Entschluss, auf die High-Level API Hector zurückzugreifen. Die wichtigsten Funktionen zur Erstellung des Cassandra Datenbankschemas [1]:

```
//Verbindung mit Datenbank herstellen
$<cassandra_home> /bin/cassandra-cli -host localhost -port 9160

//Keyspace Statistics erzeugen
```

```
[default@unknown] create keyspace Statistics;

//Verbindung zu Keyspace herstellen
[default@unknown] use Statistics;

//Column Family tracking_request erzeugen
[default@Statistics] create column family tracking_request;
```

Hector API

Da man mit Thrift sehr schnell an seine Grenzen stößt, wurde aufgrund der zahlreichen Features die Hector API für die Implementierung der Cassandra DAOs verwendet.

Die Hector API kann mit zahlreichen Features aufwarten [18]:

- einfaches, objektorientiertes Interface für Cassandra,
- clientseitige Fehlerbehandlung,
- Connection Pool für verbesserte Performance und Skalierbarkeit,
- Monitoring und Management mittels JMX[2],
- Load-Balancing Konfiguration und Erweiterung,
- automatische Erkennung hinzugefügter Knoten im Cluster,
- einfacher ORM Mapper,
- typsicheres Cassandra Datenmodell.

5.2.3 Data Tracking

Um Informationen über Besucher einer Webseite zu erhalten, bedarf es einer effizienten Tracking-Methode, da man durchaus mit hohem Traffic und damit zahlreichen Schreiboperationen auf der Datenbank rechnen muss. Für simple Tracking-Operationen im Zusammenhang mit Cassandra, die nur wenige Parameter speichern, wurde folgende Methode implementiert:

```
1  public void insertSingleRowSingleColumn(String key, String columnName,
       String columnValue) {
2    Mutator<String> mutator = HFactory.createMutator(keyspace,
     serializer);
3    mutator.insert(key, this.columnFamily, HFactory.createStringColumn(
     columnName, columnValue));
4  }
```

Da jedoch im Falle des Analytics Protoypen eine Vielzahl an Informationen über den Besucher gespeichert werden, verlangt dies nach einer effizienteren Methode. Mithilfe der bereits erwähnten Hector API ist es möglich, Batch-Operationen durchzuführen, um eine Kollektion an Daten in Form von Key-Value-Paar zu speichern. Diese Methode wurde entsprechend dem

[2]http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html

Tracking-Objekt angepasst und wird auch in der Implementierung angewendet.

```
1  public void insertMultipleColumns(TrackingRequest statistic, String
     columnFamily){
2    Mutator<String> mutator = HFactory.createMutator(keyspace,
     serializer);
3    String uid = statistic.getUniqueId();
4    if (statistic.getSiteUrl() != null) {
5      mutator.addInsertion(uid, columnFamily, HFactory.
     createStringColumn(TrackingRequest.SITE_URL_PROPERTY, statistic.
     getSiteUrl()));
6    }
7    if (statistic.getIpUser() != null) {
8      mutator.addInsertion(uid, columnFamily, HFactory.
     createStringColumn(TrackingRequest.USER_IP_PROPERTY, statistic.
     getIpUser()));
9    }
10   if (statistic.getReferer() != null) {
11     mutator.addInsertion(uid, columnFamily, HFactory.
     createStringColumn(TrackingRequest.REFERER_PROPERTY, statistic.
     getReferer()));
12   }
13   if (statistic.getCookieId() != null) {
14     mutator.addInsertion(uid, columnFamily, HFactory.
     createStringColumn(TrackingRequest.COOKIE_PROPERTY, statistic.
     getCookieId()));
15   }
16   if (statistic.getSessionId() != null) {
17     mutator.addInsertion(uid, columnFamily, HFactory.
     createStringColumn(TrackingRequest.SESSION_PROPERTY, statistic.
     getSessionId()));
18   }
19   if (statistic.getUserAgent() != null) {
20     mutator.addInsertion(uid, columnFamily, HFactory.
     createStringColumn(TrackingRequest.USER_AGENT_PROPERTY, statistic.
     getUserAgent()));
21   }
22   if (statistic.getBrowser() != null) {
23     mutator.addInsertion(uid, columnFamily, HFactory.
     createStringColumn(TrackingRequest.BROWSER_PROPERTY, statistic.
     getBrowser()));
24   }
25   if (statistic.getBrowserDetail() != null) {
26     mutator.addInsertion(uid, columnFamily, HFactory.
     createStringColumn(TrackingRequest.BROWSER_DETAILS_PROPERTY,
     statistic.getBrowserDetail()));
27   }
28   if (statistic.getOperatingSystem() != null) {
29     mutator.addInsertion(uid, columnFamily, HFactory.
     createStringColumn(TrackingRequest.OPERATING_SYSTEM_PROPERTY,
     statistic.getOperatingSystem()));
30   }
31   if (statistic.getWindowSize() != null) {
```

```
32        mutator.addInsertion(uid, columnFamily, HFactory.
      createStringColumn(TrackingRequest.WINDOW_SIZE_PROPERTY, statistic.
      getWindowSize()));
33      }
34      if (statistic.getScreenResolution() != null) {
35        mutator.addInsertion(uid, columnFamily, HFactory.
      createStringColumn(TrackingRequest.SCREEN_RESOLUTION_PROPERTY,
      statistic.getScreenResolution()));
36      }
37      if (statistic.getPageTitle() != null) {
38        mutator.addInsertion(uid, columnFamily, HFactory.
      createStringColumn(TrackingRequest.PAGE_TITLE_PROPERTY, statistic.
      getPageTitle()));
39      }
40      if (statistic.getTimestamp() != null) {
41        mutator.addInsertion(uid, columnFamily, HFactory.
      createStringColumn(TrackingRequest.TIMESTAMP_PROPERTY, statistic.
      getTimestamp().toString()));
42      }
43      if (statistic.getHomepage() != null) {
44        mutator.addInsertion(uid, columnFamily, HFactory.
      createStringColumn(TrackingRequest.HOMEPAGE_PROPERTY, statistic.
      getHomepage()));
45      }
46      if (statistic.getMandant() != null) {
47        mutator.addInsertion(uid, columnFamily, HFactory.
      createStringColumn(TrackingRequest.CLIENT_PROPERTY, statistic.
      getMandant()));
48      }
49      if (statistic.getPublicKey() != null) {
50        mutator.addInsertion(uid, columnFamily, HFactory.
      createStringColumn(TrackingRequest.PUBLIC_KEY_PROPERTY, statistic.
      getPublicKey()));
51      }
52      mutator.execute();
53    }
```

5.2.4 Map/Reduce Integration

Die Anforderung an den Analytics Prototypen verlangen eine echtzeitnahe Abfrage der Daten. Durch das Schemadesign und die Aggregation der Daten war in diesem Fall jedoch die Verwendung der vorhanden APIs nicht möglich. Ein weiteres Problem besteht in der stetig steigenden Menge an Daten. Um trotzdem akzeptable Reaktionszeiten zu erreichen, wurde entschieden, eine Hadoop Integration des Map/Reduce Frameworks zu verwenden und die in Cassandra gespeicherten Daten diesem Framework als Input zur Verfügung zu stellen. Der Output der Map/Reduce Jobs soll wiederum als eigenständige Column Family in Cassandra gespeichert werden. Die Hadoop Integration von Cassandra setzt sich aus den Hauptphasen *setup*, *map* und *reduce* zusammen.

Die vollständige Implementierung eines Map/Reduce Jobs, welcher im Analytics Prototypen Anwendung findet, wird hier anhand der einzelnen Phasen mit dazugehörigen Code-Snippets beschrieben. Dabei ist die Implementierung sehr dynamisch gestaltet, wodurch eine Analyse unterschiedlicher Eigenschaften – wie beispielsweise Browser, Bildschirmauflösung, Betriebssystem – möglich ist.

Setup

Das Setup definiert sich in diesem Fall über eine sogenannte run-Methode, welche sämtliche Konfigurationsparameter des Map/Reduce Prozesses festlegt. Dabei werden die Input Daten, welche aus der Cassandra Datenbank stammen, sowie der Output definiert. Dieser besteht aus den Ergebnissen der Reduce-Phase, welche in einer eigenständigen Column Family von Cassandra gespeichert werden.

```
1   public int run() throws Exception {
2     CURRENT_DAY = MapReduceUtils.currentDayOfYear();
3     LOGGER.info("CURRENT DAY:" + CURRENT_DAY);
4     super.setConf(new Configuration());
5     for (int j = 0; j < OUTPUT_COLUMN_FAMILIES.length; j++){
6       String columnFamily = OUTPUT_COLUMN_FAMILIES[j];
7       getConf().set(OUTPUT_COLUMN_FAMILY, columnFamily);
8
9       for (int i = 0; i < COLUMNS.length; i++) {
10        String columnName = COLUMNS[i];
11        getConf().set(CONF_COLUMN_NAME, columnName);
12
13        Job job = new Job(getConf(), JOBNAME);
14        job.setJarByClass(AllVisitsJob.class);
15        job.setMapperClass(AllVisitsMapper.class);
16        job.setReducerClass(ReducerToCassandra.class);
17
18        job.setMapOutputKeyClass(Text.class);
19        job.setMapOutputValueClass(IntWritable.class);
20        job.setOutputKeyClass(ByteBuffer.class);
21        job.setOutputValueClass(List.class);
22
23        job.setOutputFormatClass(ColumnFamilyOutputFormat.class);
24        ConfigHelper.setOutputColumnFamily(job.getConfiguration(),
      KEYSPACE,
25                              getConf().get(OUTPUT_COLUMN_FAMILY));
26
27        job.setInputFormatClass(ColumnFamilyInputFormat.class);
28
29        ConfigHelper.setRpcPort(job.getConfiguration(), PORT);
30        ConfigHelper.setInitialAddress(job.getConfiguration(), HOST);
31        ConfigHelper.setPartitioner(job.getConfiguration(), PARTITIONER)
      ;
32        ConfigHelper.setInputColumnFamily(job.getConfiguration(),
      KEYSPACE, INPUT_COLUMN_FAMILY);
33
```

```
34
35          SlicePredicate predicate = new SlicePredicate().
36              setColumn_names(Arrays.asList(ByteBufferUtil.bytes(COLUMNS[i
        ]),
37                              ByteBufferUtil.bytes("mandant"),
38                              ByteBufferUtil.bytes("homepage"),
39                              ByteBufferUtil.bytes("cookie-id"),
40                              ByteBufferUtil.bytes("session-id"),
41                              ByteBufferUtil.bytes("timestamp_req")));
42
43          ConfigHelper.setInputSlicePredicate(job.getConfiguration(),
        predicate);
44
45          job.waitForCompletion(true);
46        }
47
48      }
49      return 0;
50
51    }
52
53
```

Map

Die Map-Phase hat die Aufgabe entsprechende Schlüssel-Wert-Paare zu erstellen. Diese werden über die Input Columns von Cassandra abgerufen und entsprechend aufbereitet. Der definierte Schlüssel wird anschließend als Parameter an das Context Objekt übergeben und erhält zusätzlich ein IntWritable(1) Objekt. Die Ergebnisse der Map-Phase werden anschließend an den Reducer weitergeleitet.

```
1       public void map(ByteBuffer key, SortedMap<ByteBuffer, IColumn>
        columns,Context context)
2                             throws IOException, InterruptedException {
3
4         String entryValue = "undefined";
5         String mandant = "undefined";
6         String homepage = "undefined";
7         String sessionId = "undefined";
8         String cookieId = "undefined";
9         int day = 0;
10
11
12        for (Map.Entry<ByteBuffer, IColumn> entry : columns.entrySet()) {
13          IColumn column = entry.getValue();
14          if (column == null)
15            continue;
16
17          String columnName = ByteBufferUtil.string(column.name());
18
19
```

```
20          if (columnName.equalsIgnoreCase(confColumnName)) {
21            entryValue = ByteBufferUtil.string(column.value());
22          }
23
24          else if (columnName.equalsIgnoreCase("timestamp_req")) {
25            String columnValue = ByteBufferUtil.string(column.value());
26            day = MapReduceUtils.getDayofYear(columnValue);
27          }
28
29          else if (columnName.equalsIgnoreCase("mandant")) {
30            mandant = ByteBufferUtil.string(column.value());
31          }
32
33          else if (columnName.equalsIgnoreCase("homepage")) {
34            homepage = ByteBufferUtil.string(column.value());
35          }
36
37
38          else if (columnName.equalsIgnoreCase("session-id")) {
39            sessionId = ByteBufferUtil.string(column.value());
40          }
41
42          else if (columnName.equalsIgnoreCase("cookie-id")) {
43            cookieId = ByteBufferUtil.string(column.value());
44          }
45
46          if (day == CURRENT_DAY) {
47            if(confColumnFamily.equalsIgnoreCase("tracked_visits")){
48              word.set(mandant + "/" + homepage + "/" + entryValue + "/" +
         cookieId);
49            }
50            else{
51              word.set(mandant + "/" + homepage + "/" + entryValue + "/" +
         sessionId);
52            }
53
54            context.write(word, one);
55          }
56        }
57      }
58    }
```

Reduce

Die abschließende Reduce-Phase summiert die erhaltenen Werte und führt eine Aggregation der Schlüssel-Wert-Paare durch. Die daraus entstehenden Ergebnisse werden über Mutator-Objekt der Java API in der Cassandra Datenbank gespeichert und können von dort beliebig abgerufen und weiterverwendet werden. Auch eine Kombination unterschiedlicher Map/Reduce Jobs ist möglich.

```
1      public void reduce(Text word, Iterable<IntWritable> values,
```

```
2            Context context) throws IOException, InterruptedException
     {
3      String input = word.toString();
4      String delimiter = "\\/";
5      String [] temp = input.split(delimiter);
6
7      outputKey = ByteBufferUtil.bytes(temp[0] + "/" + temp[1]);
8
9      int sum = 0;
10     for (IntWritable val : values)
11       sum += val.get();
12
13     word.set(context.getConfiguration().get(CONF_COLUMN_NAME) + "/" +
     temp[2] + "/" + temp[3]);
14
15     context.write(outputKey,Collections.singletonList(getMutation(word
     , sum)));
16   }
```

Problemstellung Data Input

Der Prototyp wurde so konzipiert, dass jeder Aufruf einer Website mittels des Tracking-Codes erfasst wird und somit einen Eintrag in der Datenbank erzeugt. Dies führt bei hochfrequentierten Webseiten zu besonders großen Datenmengen.

Diese Tracking-Daten dienen wiederum als Input für die Map/Reduce Jobs, wodurch es eines Filtermechanismus bedarf, der den Input an Daten reduziert. Im Rahmen dieser Masterarbeit wurden hierzu drei unterschiedliche Lösungswege erarbeitet.

Lösung 1: Super Columns Dieser Lösungsansatz dient dazu eine chronologische Ordnung in das Datenbankschema von Cassandra zu integrieren und effiziente Abfragen zu ermöglichen. Durch die Verwendung von sogenannten Super Columns wird eine flache hierarchische Gliederung der Daten eingeführt.

Im Analytics Prototypen werden die Daten täglich analysiert. Daher sollten die Map/Reduce Jobs ebenso, als Input, nur die jeweils für den aktuellen Tag relevanten Daten erhalten. Um dies auch im Datenbankschema dementsprechend zu repräsentieren, wird für jeden Kalendertag eine Super Column erstellt, welche die jeweiligen Spalten und Werte beinhaltet.

Das Problem bei diesem Lösungsansatz sind große Einbußen in der Performance, wobei mittlerweile sogar von der Verwendung von Super Columns abgeraten wird und stattdessen auf effizientere Alternativen zurückgegriffen werden sollte – welche in den folgenden Lösungsansätzen Anklang finden. Weiters wurde durch die Verwendung des Random Partitioners von Cassandra, welcher aber aufgrund des idealen Load Balancing dringendst empfohlen wird, der ständig wachsende Input an Daten keineswegs minimiert. Durch

die zufällige Verteilung der Zeileneinträge im Cluster anhand einer Hash-Funktion, ist es nicht möglich ein Pre-Filtering der Input Daten durchzuführen. Zusätzliche Abfragen in der Map-Phase einzuführen wäre hier die Alternative, allerdings würde das wiederum erhebliche Einbußen in der Performance zur Folge haben. Im Großen und Ganzen kann man daher den Lösungsweg über die Super Columns eindeutig als nicht zufriedenstellend klassifizieren.

Lösung 2: Reference Columns Aufgrund der Performance-Probleme, welche mit der Verwendung von Super Columns einhergingen, bedurfte es eines weiteren Lösungsweges, um das Problem des vermehrten Daten Inputs in den Griff zu bekommen. Dieser beinhaltet die Verwendung von Referenzen und somit einen Verstoß gegen das Paradigma der nicht-relationalen Datenbank. Während des eigentlichen Tracking-Prozesses analysiert der Algorithmus einen aktuellen Zeitstempel und speichert, zusätzlich zu den erfassten Rohdaten, den definierten Zeilenschlüssel als Referenz in einer eigenen Column Family. Deren Schlüssel wird mit einem Datumswert festgelegt und enthält die erwähnten Referenzen in Form von Spalten. Mithilfe einer einfachen Iteration über diese Spalten werden die Rohdaten des Tracking-Prozesses abgerufen.

Dieser Lösungsansatz bietet zwar die Möglichkeit, Daten chronologisch nach Tagen zu sortieren, jedoch enthält die Map/Reduce Integration von Hadoop keine Konfigurationsparameter, um diese entsprechend als Input für die Map-Prozesse zu verwenden. Besonders bei hochfrequentierten Webseiten kann sich das Tracking der Daten, aufgrund der zusätzlichen Referenzierung, negativ auf die Performance auswirken. Diese Problematik führt dazu, dass dieser Lösungsansatz ebenfalls für die programmatische Umsetzung des Prototypen unbrauchbar ist.

Lösung 3: Index Expressions Cassandra bietet seit der Version 1.1 die Möglichkeit, sogenannte Secondary Indexes auch in der Map/Reduce Integration von Hadoop zu nutzen. Dazu wird eine Spalte mit einem Index versehen, wodurch eine Filterung der Daten mittels Vergleichsoperatoren vorgenommen werden kann. Die Spalte der Rohdaten, welche den Zeitstempel des Tracking-Prozesses enthält, wird daher mit einem Secondary Index versehen. Danach wird bei der Konfiguration der Map/Reduce Jobs eine Index Expression erstellt, welche die Rohdaten scannt und eine Filterung entsprechend der definierten Angaben vornimmt. Diese gefilterten Daten werden anschließend an die Map-Prozesse weitergeleitet. Im Falle des Prototypen definiert die Expression eine Filterung der Daten für den aktuellen Tag, wodurch die Map-Phase mit keinem zusätzlichen Overhead mehr belastet wird und in dieser Phase auch keine Filterung der Daten mehr durchgeführt wird.

Um dies genauer zu veranschaulichen, wird hier ein Auszug aus der Kon-

figuration gegeben.

```
1  IndexExpression expr = new IndexExpression(
2                            ByteBufferUtil.bytes(CURRENT_DAY_TIMESTAMP),
3                            IndexOperator.GTE,
4                            ByteBufferUtil.bytes(0)
5                        );
6
7  ConfigHelper.setInputRange(
8    job.getConfiguration(),
9    Arrays.asList(expr)
10 );
```

Diese Funktionalität des Pre-Filtering erweist sich, ohne Zweifel, als die effizienteste Methode Map/Reduce Jobs für die Analyse der Tracking-Daten zu implementieren und zu konfigurieren. Eine Gegenüberstellung von Cassandra und HBase erfolgt ebenfalls in dieser Masterarbeit (siehe dazu Kapitel 7).

Kapitel 6

Implementierung Apache HBase

6.1 HBase Allgemein

HBase ist, ebenfalls wie Cassandra, ein verteiltes Datenbanksystem der Kategorie Wide Column Store, das sich durch die Eigenschaft der horizontalen Skalierbarkeit auszeichnet. Ursprünglich Ende 2006 von Chad Walters und Jim Kellerman der Firma Powerset ins Leben gerufen, wurde es im Jahr 2008 von der Apache Software Foundation auch als Top-Level-Projekt angeführt. HBase kann man durchaus als Nachbau von Googles BigTable bezeichnen. Nachdem dieses – wie bereits erwähnt – jedoch als proprietäres System gehandelt wird, haben es sich die Entwickler von HBase zur Aufgabe gemacht, ein quelloffenes Duplikat der Google Datenbank zu erzeugen und der Öffentlichkeit frei zugänglich zu machen. Dieses Vorhaben hat sich inzwischen durchaus bewährt, da HBase bereits von Facebook, sowie zahlreichen weiteren namhaften Unternehmen wie Yahoo, IBM oder AOL eingesetzt und gefördert wird. Darüber hinaus existiert mit dem Unternehmen Cloudera bereits ein Anbieter für professionellen Support. Im Gegensatz zu Cassandra kann man HBase als Teil des Apache Frameworks Hadoop betrachten, welches eine Umsetzung von Google's proprietären Technologien für die Open Source Community repräsentiert. HBase ist vollständig in Java implementiert und speichert die gesamten Daten und Log-Files in einem verteilten Dateisystem namens HDFS (Hadoop File System) [10].

6.1.1 Datenmodell

Wie in Abb. 6.1 zu erkennen, setzt sich das HBase Schema aus Tabellen zusammen, welche Daten in Form von Zeilen und Spalten enthalten. Ähnlich dem Prinzip von Cassandra wird eine Zeile über einen eindeutigen Schlüssel identifiziert, über den der Zugriff auf die einzelnen Spalten und deren Einträ-

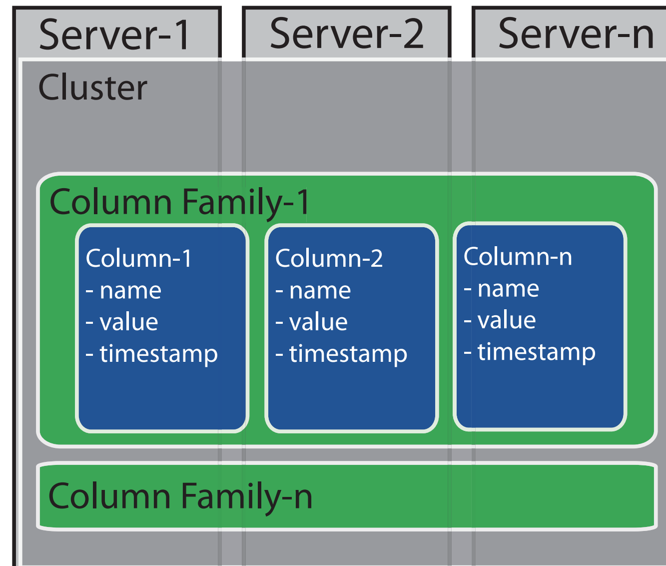


Abbildung 6.1: HBase Datenmodell

ge erfolgt. Dieser Schlüssel ist in Form eines Byte-Arrays in der Datenbank hinterlegt und bietet dem Programmierer somit zahlreiche Möglichkeiten in Punkto Definition. Da auch sämtliche andere Werte in HBase als Byte-Arrays vorhanden sind, muss das Thema der Typsicherheit in die Logik der Applikation verlagert werden.

Um eine Strukturierung des Datenbankschemas zu ermöglichen, wurden die bereits von Cassandra bekannten Column Families eingeführt, welche eine Zusammenfassung ähnlicher Spaltentypen darstellen. Da sich HBase Datenmodell jedoch nicht grundlegend von jenem von Cassandra unterscheidet, wird an dieser Stelle nicht näher auf die Thematik eingegangen.

6.1.2 Datenoperationen

Als ersten Schritt widmet man sich der Erstellung des Datenbank-Schemas. Diese erfolgt über eine integrierte JRuby-Shell oder eine Java API, die ebenfalls die nötigen Funktionen für CRUD-Operationen zur Verfügung stellt. Ein weiterer Ansatz ist die Verwendung von HBase mittels einer REST-API, über die mit einem Java Servlet namens Stargate kommuniziert werden kann. Für fortgeschrittene Abfragemechanismen, die nebenläufige Berechnungen implementieren, sei an dieser Stelle das über Hadoop integrierte Framework Map/Reduce angeführt, welches einen wichtigen Teil dieser Masterarbeit ausmacht und auch im Analytics Prototypen seine Anwendung findet. Weitere Ausführungen zum Map/Reduce Framework finden sich in weiterer Folge.

6.1.3 Distribution

Da HBase für den Betrieb in großen Cluster-Systemen mit einer Vielzahl an Knoten vorgesehen ist, wurde bei der Entwicklung besonderer Wert auf einfache Skalierung und Konfiguration gelegt. Im Gegensatz zu Cassandra, wo jeder Knoten im Netzwerk gleichwertig ist, wird ein HBase Cluster nach dem 1-Master-/N-Slaves-Prinzip aufgebaut. Die als Region Server bezeichneten Slaves speichern die Daten ab und stellen diese bei Bedarf wieder zur Verfügung. Die Verteilung der einzelnen Daten und Tabellen erfolgt automatisch über den Master-Knoten, wodurch der Administrationsaufwand, im Gegensatz zu verteilten relationalen Datenbanksystemen, deutlich geringer ausfällt. Das Monitoring des gesamten Clusters hinsichtlich Synchronisation, Auslastung und Verfügbarkeit übernimmt ein sogenannter ZooKeeper-Cluster. Das Thema Datensynchronisation verlangt bei HBase nach einem NTP-Server, um die Daten entsprechend ihrem Zeitstempel in der chronologisch richtigen Reihenfolge zu speichern. In Punkto Replikation existiert in HBase bereits ein integrierter Mechanismus, der sich allerdings noch in einem frühen Beta-Stadium befindet und nicht für den produktiven Einsatz geeignet ist. Eine weitere Möglichkeit besteht darin, die von Hadoop zur Verfügung gestellte Replikation mittels des verteilten Dateisystems HDFS zu verwenden.

6.2 Implementierungsdetails Prototyp

6.2.1 Datenbankschema Prototyp

Das Datenbankschema von HBase mit dem von Cassandra weitestgehend gleichgestellt werden kann, wurde dieses bei der Implementierung des Prototypen beibehalten. Darüber hinaus erweisen sich die identen Strukturen als wertvoll, um im Zuge der Evaluierung von Cassandra und HBase anschauliche Ergebnisse zu erhalten und diese auch dementsprechend interpretieren zu können.

6.2.2 API Schnittstellen

HBase API

HBase beinhaltet eine vollständig integrierte Java API und bedarf somit keiner zusätzlichen Bibliotheken. Dies vermindert den Overhead und erleichtert die Implementierung der Data Access Objects.

Die Methoden für die Erstellung des Datenbankschemas wurden für den Prototypen ebenfalls vollständig in Java implementiert. Es steht zwar für einfache Operationen eine Shell zur Verfügung, jedoch wurde diese im Zuge der Umsetzung des Projektes nicht verwendet. Die wichtigsten Funktionen zur Erstellung des HBase Datenbankschemas:

```
1   private static Configuration conf = null;
```

```
2   private static HTable table = null;
3   private static final String TABLE = "Statistics";
4   private static final String FAMILY = "tracking_request";
5
6   /**
7    * Initialization
8    */
9   static {
10    conf = HBaseConfiguration.create();
11    try{
12      table = new HTable(conf, TABLE);
13    }
14
15    catch(IOException e){
16      LOGGER.warn(e.toString());
17    }
18
19  }
```

```
1   /**
2    * Create a table
3    */
4   public void createTable(String tableName, String[] familys) {
5     try {
6       HBaseAdmin admin = new HBaseAdmin(conf);
7       if (admin.tableExists(tableName)) {
8         LOGGER.info("table already exists!");
9       } else {
10        HTableDescriptor tableDesc = new HTableDescriptor(tableName);
11        for (int i = 0; i < familys.length; i++) {
12          tableDesc.addFamily(new HColumnDescriptor(familys[i]));
13        }
14        admin.createTable(tableDesc);
15        LOGGER.info("create table " + tableName + " ok.");
16      }
17    } catch (Exception e) {
18      LOGGER.info(e.toString());
19    }
20  }
```

```
1   /**
2    * Create a column family
3    */
4   public void createFamily(String tableName, String family){
5     try {
6       HBaseAdmin admin = new HBaseAdmin(conf);
7       admin.disableTable(tableName);
8       admin.addColumn(tableName, new HColumnDescriptor(family));
9       LOGGER.info("add family " + family + " ok.");
10      admin.enableTable(tableName);
11    } catch (MasterNotRunningException e) {
12      e.printStackTrace();
13    } catch (IOException e) {
14      e.printStackTrace();
```

```
15     }
16
17   }
```

6.2.3 Data Tracking

Das Data Tracking wurde ebenfalls zum besseren Vergleich – vor allem in Punkto Performance – dem von Cassandra angeglichen. Es wurden zwei unterschiedliche Methoden zum Schreiben auf die Datenbank implementiert. Die Methode für singuläre Einträge setzt sich wie folgt zusammen:

```
1   /**
2    * Put (or insert) a singlerow
3    */
4   public void addRecord(String tableName, String rowKey,
5       String family, String qualifier, String value){
6     try {
7       Put put = new Put(Bytes.toBytes(rowKey));
8       put.add(Bytes.toBytes(family), Bytes.toBytes(qualifier),
9           Bytes.toBytes(value));
10      table.put(put);
11      LOGGER.info("insert record " + rowKey + " to table "
12                  + tableName + " ok.");
13    } catch (IOException e) {
14      e.printStackTrace();
15    }
16  }
```

Auch in HBase besteht die Möglichkeit der Batch-Verarbeitung von Schreiboperationen. Da dies aufgrund der zahlreichen Tracking-Properties durchaus ein Vorteil hinsichtlich Performance ist, fand diese Methode Einzug in die Implementierung.

```
1 public void addMultipleRecords(TrackingRequest statistic){
2     List<Put> puts = new ArrayList<Put>();
3     puts.add(new Put(Bytes.toBytes(statistic.getUniqueId())).add(Bytes.
      toBytes(FAMILY),
4                         Bytes.toBytes(TrackingRequest.SITE_URL_PROPERTY)
      , Bytes.toBytes(statistic.getSiteUrl())));
5     puts.add(new Put(Bytes.toBytes(statistic.getUniqueId())).add(Bytes.
      toBytes(FAMILY),
6         Bytes.toBytes(TrackingRequest.USER_IP_PROPERTY), Bytes.toBytes(
      statistic.getIpUser())));
7     puts.add(new Put(Bytes.toBytes(statistic.getUniqueId())).add(Bytes.
      toBytes(FAMILY),
8         Bytes.toBytes(TrackingRequest.REFERER_PROPERTY), Bytes.toBytes(
      statistic.getReferer())));
9     puts.add(new Put(Bytes.toBytes(statistic.getUniqueId())).add(Bytes.
      toBytes(FAMILY),
10        Bytes.toBytes(TrackingRequest.COOKIE_PROPERTY), Bytes.toBytes(
      statistic.getCookieId())));
11    puts.add(new Put(Bytes.toBytes(statistic.getUniqueId())).add(Bytes.
      toBytes(FAMILY),
```

```
12          Bytes.toBytes(TrackingRequest.SESSION_PROPERTY), Bytes.toBytes(
        statistic.getSessionId())));
13        puts.add(new Put(Bytes.toBytes(statistic.getUniqueId())).add(Bytes.
        toBytes(FAMILY),
14          Bytes.toBytes(TrackingRequest.USER_AGENT_PROPERTY), Bytes.
        toBytes(statistic.getUserAgent())));
15        puts.add(new Put(Bytes.toBytes(statistic.getUniqueId())).add(Bytes.
        toBytes(FAMILY),
16          Bytes.toBytes(TrackingRequest.BROWSER_PROPERTY), Bytes.toBytes(
        statistic.getBrowser())));
17        puts.add(new Put(Bytes.toBytes(statistic.getUniqueId())).add(Bytes.
        toBytes(FAMILY),
18          Bytes.toBytes(TrackingRequest.BROWSER_DETAILS_PROPERTY), Bytes.
        toBytes(statistic.getBrowserDetail())));
19        puts.add(new Put(Bytes.toBytes(statistic.getUniqueId())).add(Bytes.
        toBytes(FAMILY),
20          Bytes.toBytes(TrackingRequest.OPERATING_SYSTEM_PROPERTY), Bytes.
        toBytes(statistic.getOperatingSystem())));
21        puts.add(new Put(Bytes.toBytes(statistic.getUniqueId())).add(Bytes.
        toBytes(FAMILY),
22          Bytes.toBytes(TrackingRequest.WINDOW_SIZE_PROPERTY), Bytes.
        toBytes(statistic.getWindowSize())));
23        puts.add(new Put(Bytes.toBytes(statistic.getUniqueId())).add(Bytes.
        toBytes(FAMILY),
24          Bytes.toBytes(TrackingRequest.SCREEN_RESOLUTION_PROPERTY), Bytes
        .toBytes(statistic.getScreenResolution())));
25        puts.add(new Put(Bytes.toBytes(statistic.getUniqueId())).add(Bytes.
        toBytes(FAMILY),
26          Bytes.toBytes(TrackingRequest.PAGE_TITLE_PROPERTY), Bytes.
        toBytes(statistic.getPageTitle())));
27        puts.add(new Put(Bytes.toBytes(statistic.getUniqueId())).add(Bytes.
        toBytes(FAMILY),
28          Bytes.toBytes(TrackingRequest.TIMESTAMP_PROPERTY), Bytes.toBytes
        (statistic.getTimestamp().toString())));
29        puts.add(new Put(Bytes.toBytes(statistic.getUniqueId())).add(Bytes.
        toBytes(FAMILY),
30          Bytes.toBytes(TrackingRequest.HOMEPAGE_PROPERTY), Bytes.toBytes(
        statistic.getHomepage())));
31        puts.add(new Put(Bytes.toBytes(statistic.getUniqueId())).add(Bytes.
        toBytes(FAMILY),
32          Bytes.toBytes(TrackingRequest.CLIENT_PROPERTY), Bytes.toBytes(
        statistic.getMandant())));
33        puts.add(new Put(Bytes.toBytes(statistic.getUniqueId())).add(Bytes.
        toBytes(FAMILY),
34          Bytes.toBytes(TrackingRequest.PUBLIC_KEY_PROPERTY), Bytes.
        toBytes(statistic.getPublicKey())));
35        try{
36          table.put(puts);
37          LOGGER.info("insert multiple recoreds " + statistic.getUniqueId()
        + " to table " + TABLE
38              + " ok.");
39        }
40        catch(IOException e){
41          LOGGER.info(e.toString());
```

```
42      }
43   }
```

6.2.4 Map/Reduce Framework

HBase stellt bereits ein vollwertiges Map/Reduce Framework zur Verfügung, da die Datenbank auf dem Apache Hadoop Projekt basiert. Dies bringt zahlreiche Vorteile mit sich, da kein zusätzlicher Overhead entsteht und auch hinsichtlich der Implementierung einige Vorteile gegenüber Cassandra bietet. Diese werden hier anhand der einzelnen Phasen genauer beschrieben. Wie auch in Cassandra besteht der Map/Reduce Prozess von HBase aus den Phasen *setup*, *map* und *reduce*.

Um wiederum einen genauen Vergleich der beiden Systeme ziehen zu können, wurde dieser Map/Reduce Job weitestgehend ähnlich wie in Cassandra implementiert.

Setup

Die im Zuge der run-Methode durchgeführte Setup-Phase definiert hier wiederum die Eingangs- und Ausgangsparameter. Ein deutlicher Vorteil gegenüber Cassandra ist allerdings die standardmäßige Filterfunktion, welche zur Lösung des Data Input Problems herangezogen wird.

```
1    public int run() throws Exception {
2        CURRENT_DAY = MapReduceUtils.currentDayOfYear();
3        Configuration config = HBaseConfiguration.create();
4        Job job = new Job(config,"ExampleSummary");
5        job.setJarByClass(AllVisitsJobHBase.class);     // class that
       contains mapper and reducer
6
7        Scan scan = new Scan();
8        scan.setCaching(500);        // 1 is the default in Scan, which
       will be bad for MapReduce jobs
9        scan.setCacheBlocks(false);  // don't set to true for MR jobs
10       // set other scan attrs
11       SingleColumnValueFilter filter = new SingleColumnValueFilter
12                     (Bytes.toBytes(INPUT_COLUMN_FAMILY),Bytes.toBytes("
       timestamp_req"),
13                      CompareOp.GREATER_OR_EQUAL, Bytes.toBytes(new Date(
       CURRENT_DAY).getTime()) );
14       scan.setFilter(filter);
15
16
17       TableMapReduceUtil.initTableMapperJob(
18         SOURCE_TABLE,        // input table
19         scan,               // Scan instance to control CF and attribute
        selection
20         MyMapper.class,     // mapper class
21         Text.class,         // mapper output key
22         IntWritable.class,  // mapper output value
```

```
23        job);
24      TableMapReduceUtil.initTableReducerJob(
25        SOURCE_TABLE,        // output table
26        MyTableReducer.class,    // reducer class
27        job);
28      job.setNumReduceTasks(1);   // at least one, adjust as required
29
30      boolean b = job.waitForCompletion(true);
31      if (!b) {
32        throw new IOException("error with job!");
33      }
34      return 0;
35  }
```

Map

Die Phase der Map-Prozesse bietet in HBase die Möglichkeit, direkt auf einzelne Spalten und deren Werte zuzugreifen. Dies erleichtert die Definition eines geeigneten Schlüsselwertes. Mithilfe des Context Objektes werden, analog zu Cassandra, die Schlüssel-Wert-Paare sortiert und an die Reduce-Phase weitergeleitet.

```
1   public static class MyMapper extends TableMapper<Text, IntWritable>  {
2
3     private final IntWritable ONE = new IntWritable(1);
4       private Text text = new Text();
5
6       public void map(ImmutableBytesWritable row, Result value, Context
      context) throws IOException, InterruptedException {
7             String val = new String(value.getValue(Bytes.toBytes(INPUT_
      COLUMN_FAMILY), Bytes.toBytes(INPUT_COLUMN)));
8               text.set(val);
9
10            context.write(text, ONE);
11      }
12  }
```

f

Reduce

Die Reduce-Phase definiert sich auch in HBase über eine Iteration über die erhaltenen Werte aus den Map-Prozessen. Mithilfe eines Put-Objektes werden die Ergebnisse in der Datenbank gespeichert.

```
1   public static class MyTableReducer extends TableReducer<Text,
      IntWritable, ImmutableBytesWritable>  {
2
3     public void reduce(Text key, Iterable<IntWritable> values, Context
      context) throws IOException, InterruptedException {
4           int i = 0;
5           for (IntWritable val : values) {
```

```
 6            i += val.get();
 7          }
 8          Put put = new Put(Bytes.toBytes(key.toString()));
 9          put.add(Bytes.toBytes(OUTPUT_COLUMN_FAMILY), Bytes.toBytes("
       count"), Bytes.toBytes(i));
10
11          context.write(null, put);
12      }
13  }
```

Problemstellung Data Input

Das Problem des ansteigenden Daten Inputs erweist sich mithilfe von HBase und dem integrierten Map/Reduce Framework als durchaus einfach zu lösen. Die Konfiguration der Jobs in der Setup-Phase bietet standardmäßig eine effiziente Filterfunktion, um den Einfluss irrelevanter Daten auf die Map-Prozessen zu vermeiden. Für den Anwendungsfall dieser Masterarbeit wurde ein SingleColumnValueFilter verwendet, mit dessen Hilfe die Tracking-Daten für den aktuellen Tag Map-Prozessen als Input übergeben werden. Die Integration einer solchen Funktionalität setzt sich wie folgt zusammen:

```
1  SingleColumnValueFilter filter = new SingleColumnValueFilter
2        Bytes.toBytes(INPUT_COLUMN_FAMILY),Bytes.toBytes("timestamp_req
     "),
3        CompareOp.GREATER_OR_EQUAL, Bytes.toBytes(new Date(CURRENT_DAY).
     getTime()) );
4  scan.setFilter(filter);
```

Dieser Filter über eine einzelne Spalte bietet nur eine Möglichkeit, das Datenvolumen vor dem eigentlichen Map- und Reduce-Prozess zu minimieren. HBase bietet hier eine Vielzahl an Typen, sowie die Einführung von Regular Expressions [20].

Kapitel 7

Evaluierung und Ergebnisse

7.1 Motivation der Evaluierung

Die im Rahmen dieser Arbeit durchgeführten Testläufe wurden mittels einer Single-Node Installation von HBase und Cassandra durchgeführt. Da der Betrieb in einem Cluster-System für diese Art der Auswertungen mit zu hohem Aufwand verbunden ist, wurde auf jegliche Evaluierung diesbezüglich verzichtet. Der Fokus dieser Diplomarbeit liegt einzig und allein auf der Implementierung mittels der zur Verfügung gestellten APIs und die damit verbundenen Möglichkeiten, welche sich aus den beiden Datenbanksystemen hinsichtlich Web Analytics ergeben. Die Erfassung der Tracking-Daten, die Schemaerstellung und die nach sich ziehende Datenauswertung anhand effizienter Map/Reduce Algorithmen sind von Relevanz und finden sich auch in den Testergebnissen wieder. Diese sollen Aufschluss darüber geben, welches der beiden Datenbanksysteme (Cassandra/HBase) besser für den vom Prototypen repräsentierten Anwendungsfall geeignet ist.

7.2 Testumgebung

Um realistische Ergebnisdaten zu erzielen, wurde die Testumgebung auf einen Linux Server ausgelagert. Es fand ein vollständiges Deployment des Prototypen statt. Hier die wichtigsten Eckdaten – die Testumgebung und Datenbanksysteme betreffend:

Tabelle 7.1: Detailinformationen Testumgebung Server

Betriebssystem	Ubuntu Lucid Version 10.04
CPU	Intel(R) Core(TM) i7 CPU 2.93GHz
Apache Cassandra	Version 1.1.5
Apache HBase	Version 0.94.1

7.3 Vorbereitung und Durchführung des Evaluierungsprozesses

Für die Durchführung des Evaluierungsprozesses wurde ein Testdatensatz angelegt. Dieser umfasst zufällig generierte Tracking-Objekte, welche in der Datenbank gespeichert werden. Hauptaugenmerk liegt auf der Verarbeitung unterschiedlicher Datenmengen im Bereich Tracking, sowie der Integration des Map/Reduce Frameworks von Cassandra und HBase.

Die Testdurchläufe wurden mithilfe automatisierter Unit-Tests des Java Frameworks JUNIT (Version 4.8.2) durchgeführt, welches aufgrund der Implementierung der Data Access Objects sehr gut für diesen Anwendungsfall geeignet ist. Die Performance-Messergebnisse entstanden mittels, direkt im Algorithmus generierter, Zeitstempel.

```
1   @Test
2   public void testAddMultipleRecords(){
3     initTestCases();
4     testCreateInstances();
5     Timestamp start = HttpUtils.createTimestamp();
6     LOGGER.info("start: " + start);
7     for (int i = 0; i < trackingRequests.size(); i++){
8       trackingRequestHBaseDAO.addStatistic(trackingRequests.get(i));
9     }
10    Timestamp end = HttpUtils.createTimestamp();
11    LOGGER.info("end: " + end);
12    long difference = end.getTime() - start.getTime();
13    LOGGER.info("Time: " + difference + " Milliseconds");
14  }
```

7.3.1 Testergebnisse Data Tracking

Der erste Part der Testergebnisse umfasst das Data Tracking der Requests. Die Datenbanken wurden mit bis zu 10.000 Datensätzen beschrieben und die Durchlaufzeit in Millisekunden gemessen.

Tabelle 7.2: Erfassen der Tracking-Requests in der Datenbank

Data Tracking	Cassandra	HBase
insert 1	63 ms	11 ms
insert 10	98 ms	56 ms
insert 100	264 ms	435 ms
insert 1.000	873 ms	1225 ms
insert 10.000	4.323 ms	6.870 ms
insert 100.000	61.000 ms	56.445 ms

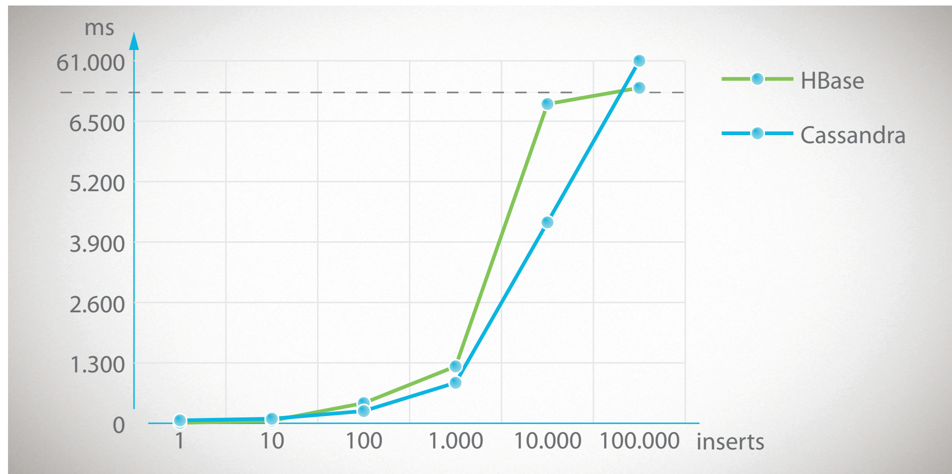


Abbildung 7.1: Diagramm Data Tracking Tests

Auswertung Data Tracking

Wie sich anhand des Auswertungsdiagramms feststellen lässt, ist die Datenbank Cassandra deutlich performanter, was Schreibvorgänge bis zu 10.000 Datensätze anbelangt. Wird diese Grenze überschritten – was bei Webseiten mit sehr hohem Traffic durchaus der Fall sein kann – drängt sich HBase klar in den Vordergrund. Besonders das Testergebnis von 100.000 Schreiboperationen zeigt deutlich auf, dass HBase diese große Menge an Daten in kürzerer Zeit verarbeiten kann.

Weiters wurde auch ein Testszenario erstellt, in dem nur ein einziger Datensatz in der Datenbank persistiert wird. Auch hier kann HBase diese Operation deutlich schneller durchführen als Cassandra.

Aufgrund dieser Testergebnisse kann man darauf schließen, dass HBase grundsätzlich besser für den Umgang mit großen Datenmengen geeignet ist. Vor allem in der Verarbeitung steigender Mengen an Request-Daten, zeigte die Datenbank hohes Potential. Hier sollte man in Betracht ziehen, dass bei der Analyse hochfrequentierter Webseiten die performante Verarbeitung von Tracking-Daten unumgänglich ist. Längere Ladezeiten und Ausfälle des Systems sind negative Faktoren, die es unbedingt zu vermeiden gilt.

7.3.2 Testergebnisse Map/Reduce

Diese Ergebnisse beinhalten die Datenaggregation der in der Datenbank befindlichen Tracking-Requests mittels Map/Reduce. Die Funktionalität der Jobs ist in Cassandra und HBase identisch und liefert die entsprechenden Performancedaten ebenfalls in Millisekunden. Der zu diesen Testzwecken implementierte Map/Reduce Job wertet die getrackten Browsertypen aus.

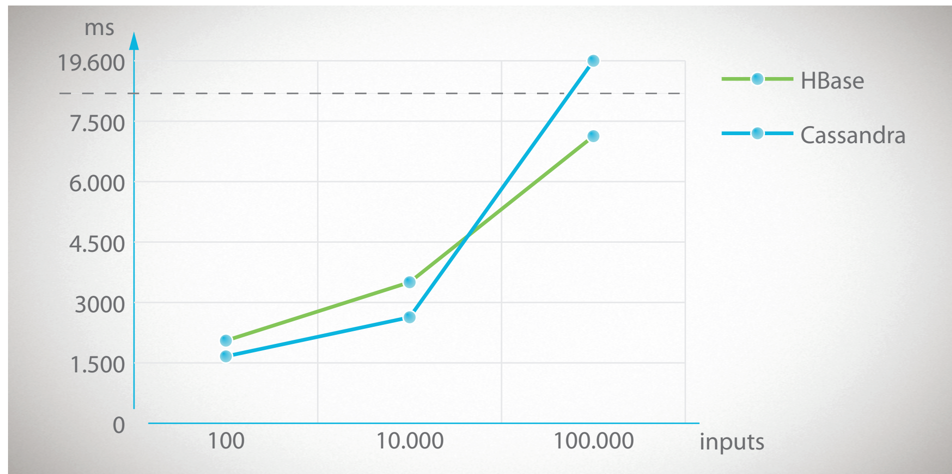


Abbildung 7.2: Diagramm Map/Reduce Tests

Eine fertige Aggregation, welche in der Datenbank gespeichert wird, sieht wie folgt aus:

Tabelle 7.3: Auswertung der Map/Reduce Jobs

Map/Reduce	Cassandra	HBase
input 1.000	1.662 ms	2.052 ms
input 10.000	2.632 ms	3.502 ms
input 100.000	19.621 ms	7.129 ms

Auswertung Map/Reduce

Die Auswertungen der Map/Reduce Abfragen erfolgten nach dem selben Schema wie jene der Tracking-Daten. Die Implementierung sieht vor, sämtliche Input Daten täglich zu aggregieren. Bei beiden Datenbanksystemen gestaltete sich dies durchaus sehr unterschiedlich, aufgrund der individuellen Hadoop Integrationen, welche das Map/Reduce Framework beinhalten. Betrachtet man die Testergebnisse, sprechen diese auf den ersten Blick für das Cassandra DBMS, da es – vor allem mit kleineren Datenmengen – die Abarbeitung der einzelnen Map/Reduce Phasen schneller erledigt.

Bei genauerer Betrachtung bietet HBase jedoch die entscheidenden Vorteile bei der Anwendung von Map/Reduce. Besonders das letzte Testergebnis – welches die Verarbeitung von 100.000 Datensätzen betrifft – spricht für HBase. Die Datenbank konnte die Ausführung des Jobs in deutlich weniger als der Hälfte der Zeit, die Cassandra benötigte, erledigen. Da Hadoop die Verwendung von Map/Reduce hauptsächlich für Datenmengen im Tera-

und Petabyte-Bereich empfiehlt, kann man hier eindeutig HBase gegenüber Cassandra den Vorzug geben. Die Parallelisierung in der Durchführung der einzelnen Phasen des Map/Reduce Frameworks ist ebenfalls ein Indiz für den Einsatz im Big Data Bereich. Wie die Testergebnisse zeigen, erlebt Cassandra doch markante Performance-Einbußen mit steigender Datenmenge. Hier bewähren sich deutlich die Filterfunktionen von HBase, welche den Input für die Map-Prozesse auf das wesentliche reduzieren kann. Mittels definierter Vergleichsoperatoren ist es möglich, diese Filterfunktionen auf die Daten in HBase anzuwenden und somit eine optimale Konfiguration des Map/Reduce Jobs zu erreichen. Hinsichtlich Cassandra bietet die Integration von Hadoop deutlich weniger Konfigurationsmechanismen. Erst seit der Version 1.1 gibt es eine Möglichkeit mittels Expressions Daten im Vorab zu filtern und somit die Datenmenge im Zuge der Konfiguration zu reduzieren. Allerdings muss die jeweilige Spalte, auf die eine Abfrage ausgeführt werden soll, vorher mit einem Secondary Index versehen werden. Sogenannte Range-Scans, bei denen nur ein Teil der Zeilen in der Datenbank abgefragt wird, sind aufgrund des Random Partitioners von Cassandra nicht durchführbar. Dieser wendet eine Hash-Funktion auf jede eingetragene Zeile an, um diese dann dementsprechend im Cluster zu verteilen – was zu einer zufälligen Anordnung der Datenbankeinträge führt. Zwar ist die Funktionalität des Random Partitioners durchaus vorteilhaft, was das Load Balancing betrifft – jedoch ist es nicht möglich nachzuvollziehen, wie die Daten im Cluster angeordnet sind. Somit bleibt meist nur die Abfrage über eine gesamte Spalte.

Aufgrund dieser Tatsachen und der eingeschränkten Konfigurationsmöglichkeiten der Map/Reduce Integration von Cassandra, ist HBase deutlich einen Schritt voraus und auch im Falle des Analytics Prototypen die bessere Wahl.

Kapitel 8

Schlussbemerkungen

8.1 Conclusio

Im Zuge dieser Masterarbeit wurde der Prototyp einer Web Analytics Anwendung implementiert. Da der Einsatz relationaler Datenbanken bei Analysen und Auswertungen, besonders in Zusammenhang mit großen Datenmengen, nicht empfehlenswert ist, fiel die Entscheidung auf die Verwendung eines Wide Column Stores. Apache Cassandra und HBase waren hier die erste Wahl. Sie bieten die Möglichkeit, eine Form des Map/Reduce Algorithmus zu implementieren und somit eine Aggregation gespeicherter Datenbankeinträge durchzuführen. Da sich diese Art der Datenabfrage als sehr effizient herausstellte, wurde Map/Reduce zu einem Kernthema dieser Arbeit.

Nach einem theoretischen Einblick in die Welt der NoSQL-Systeme und einem Überblick über die unterschiedlichen Kategorien dieser Datenbanken, wurden die beiden Implementierungsvorgänge erläutert und ein abschließender Evaluierungsprozess durchgeführt. Dieser soll darüber Aufschluss geben, welches System – für den durch den Prototyp repräsentierten Anwendungsfall – besser geeignet ist.

Im Laufe der Umsetzung wurde sehr schnell klar, dass die Erstellung eines optimalen Datenbankschemas entscheidenden Einfluss auf die Performance der Datenanalyse hat. Das relativ simpel aufgebaute Datenmodell von Cassandra und HBase ist grundsätzlich sehr gut geeignet für die Analyse der Tracking-Daten. Allerdings sollte man sich bereits vor der Erstellung des Datenbankschemas Klarheit darüber verschaffen, welche Aggregationen man durchführen möchte. Dies fordert einen Umdenkprozess im Hinblick auf relationale Systeme, wo die Erstellung des Datenbankschemas unabhängig von den folgenden Operationen stattfindet.

Grundlegende Differenzen hinsichtlich Cassandra und HBase ergeben sich bei der Implementierung des Map/Reduce Algorithmus. Da die Integrationen des Hadoop Frameworks gewisse Einschränkungen mit sich bringen, war besonders das Finden eines angemessenen Lösungsvorschlags ausschlagge-

bend für die erfolgreiche Umsetzung des Prototypen. Cassandra scheint – besonders was die Konfiguration der Map/Reduce Jobs anbelangt – noch etwas Nachholbedarf zu haben. Insbesondere die Filterung der Input Daten kann sich als sehr mühsam gestalten. HBase bietet hier, aufgrund der diversen Filtermechanismen, bei größeren Datenmengen eine eindeutig bessere Performance.

Ziel dieser Arbeit war es eine Entscheidung hinsichtlich der Verwendung eines geeigneten DBMS für den Analytics Prototypen zu finden. Eingeschränkt auf die beiden Datenbanken Cassandra und HBase wurden diverse Vergleiche – vor allem die Implementierungsmöglichkeiten betreffend – gezogen. Um eine sinnvolle Entscheidung herbeizuführen und annähernd gleiche Bedingungen zu schaffen, wurde versucht das Schema der Datenbanken identisch zu gestalten. Darüber hinaus sollte vor allem das Stichwort Big Data nicht außer Acht gelassen werden, da das Thema Web Analytics meist mit der Ansammlung großer Datenmengen verbunden ist. In Anbetracht dieser Tatsachen kann man – vorwiegend aufgrund der Evaluierungsergebnisse der Map/Redue Jobs – eindeutig Apache HBase den Vorzug geben. Obwohl Cassandra bei kleineren Datenmengen noch mit den besseren Performance-Ergebnissen aufwarten kann, ist bei der Aggregation größerer Datenmengen jenseits von 100.000 Datensätzen Apache HBase eindeutig die bessere Wahl.

8.2 Ausblick

Ohne Zweifel sind NoSQL-Systeme bereits im Vormarsch und können in vielen Bereichen den herkömmlichen relationalen DBMS den Rang ablaufen. Vor allem im Umfeld von Datenanalysen – wie sie im Zuge des Analytics Prototypen stattgefunden haben – überwiegen klar die Vorteile bei der Verwendung eines solchen Systems. Nichtsdestotrotz werden relationale Datenbanksysteme immer ihre Daseinsberechtigung haben, vor allem im Bereich von Business-Anwendungen, welcher einer strikten Konsistenz unterliegen. Ausschlaggebend für den Erfolg von NoSQL ist die große Menge an Daten, welche sich im Laufe der Zeit ansammelt und für deren Verarbeitung und Analyse diese Systeme sehr gut geeignet sind.

Doch nicht nur der auf diese Arbeit beschränkte Part der Web Analyse mit HBase und Cassandra kann mithilfe von NoSQL-Systemen sehr gut bewerkstelligt werden. Die Anwendungsfelder sind aufgrund der mittlerweile großen Auswahl an DBMS sehr vielfältig. Vor allem der Bereich Datenanalyse sollte sich nicht nur auf das World Wide Web beschränken. Algorithmen wie Map/Reduce können beispielsweise auch zur Verarbeitung unternehmensinterner Daten verwendet werden und möglicherweise dazu beitragen, bestimmte Arbeitsprozesse zu optimieren oder Einsparungsmaßnahmen zu konzipieren. Da die nötigen Werkzeuge zur Umsetzung solcher Vorhaben bereits in einer großen Vielfalt zur Verfügung stehen, ist es durchaus auch von

Priorität das optimale Tool für den individuellen Anwendungsfall zu finden.

Anhang A

Inhalt der CD-ROM/DVD

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 Masterarbeit

Pfad: /Masterarbeit

/LaTeX . . . . LaTeX-Quelldateien dieser Masterarbeit
DA.pdf . . . . Masterarbeit (Gesamtdokument)

A.2 Literaturquellen

Pfad: /Literaturquellen

/Online . . . . Online-Quellen

A.3 Quellcode

Pfad: /Quellcode

/AnalyticsPrototyp . . . Quellcode des Analytics Prototypen

A.4 Sonstiges

Pfad: /Abbildungen

*.eps . . . . Bilder und Grafiken im EPS-Format

Quellenverzeichnis

Literatur

[1] Edward Capriolo. *Cassandra High Performance Cookbook*. Packt Publishing Ltd., Juli 2011.

[2] Fay Chang u. a. *Bigtable: A Distributed Storage System for Structured Data*. Englisch. Techn. Ber. Google, Inc., 2006. URL: http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/de//archive/bigtable-osdi06.pdf.

[3] Jeffrey Dean und Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. Techn. Ber. Google, Inc., 2004. URL: http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/de//archive/mapreduce-osdi04.pdf.

[4] Giuseppe DeCandia u. a. *Dynamo: Amazon's Highly Available Key-value Store*. Techn. Ber. Amazon, 2007. URL: http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf.

[5] Stefan Edlich u. a. *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. Carl Hanser Verlag GmbH & CO. KG, Okt. 2010.

[6] Seth Gilbert und Nancy Lynch. „Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *ACM SIGACT News* 33/2 (Juni 2002).

[7] Avinash Lakshman und Prashant Malik. *Cassandra - A Decentralized Structured Storage System*. Englisch. Techn. Ber. Facebook, 2009. URL: http://www.cs.cornell.edu/projects/ladis2009/papers/lakshman-ladis2009.pdf.

[8] Leslie Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System*. Techn. Ber. Massachusetts Computer Associates, Inc., 1978. URL: http://cnlab.kaist.ac.kr/~ikjun/data/Course_work/CS642-Distributed_Systems/papers/lamport1978.pdf.

[9] David L. Mills. *A Brief History of NTP Time: Confessions of an Internet Timekeeper*. Techn. Ber. University of Delaware, 2000. URL: http://www.eecis.udel.edu/~mills/database/papers/history.pdf.

[10] Konstantin Shvachko u. a. *The Hadoop Distributed File System*. Englisch. Techn. Ber. Yahoo-Inc.com, 2012. URL: http://storageconference.org/2010/Papers/MSST/Shvachko.pdf.

[11] Werner Vogels. „Eventually Consistent“. In: *ACM Queue* 52/1 (Jan. 2009).

[12] Pete Warden. *Big Data Glossary*. O’Reilly Media, Inc., 2011.

Online-Quellen

[13] URL: http://de.wikipedia.org/wiki/Big_Data (besucht am 25.08.2012).

[14] URL: http://de.wikipedia.org/wiki/MapReduce (besucht am 24.06.2012).

[15] URL: http://en.wikipedia.org/wiki/MapReduce (besucht am 24.06.2012).

[16] URL: http://de.wikipedia.org/wiki/Lotus_Notes (besucht am 12.06.2012).

[17] Eric A. Brewer. *Towards robust distributed systems*. Juli 2000. URL: http://www.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.

[18] Patricio Echague und Nate McCall. *Hector - A high level Java client for Apache Cassandra*. URL: http://hector-client.github.com/hector/build/html/index.html.

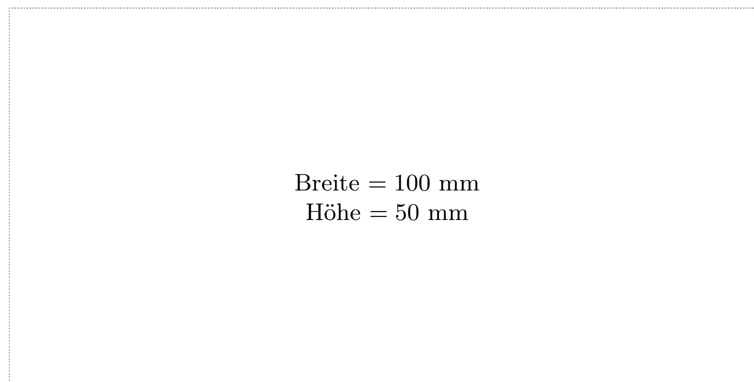
[19] Campus Gummersbach FH Koeln. *BASE*. URL: http://wikis.gm.fh-koeln.de/wiki_db/Datenbanken/BASE.

[20] Apache Software Foundation. *HBase API - Client Request Filters*. URL: http://hbase.apache.org/book/client.filter.html.

[21] Andreas Hartmann. *NoSQL in der Cloud - Business Value jenseits des Hypes*. URL: http://blog.adesso.de/nosql-in-der-cloud-business-value-jenseits-des-hypes/.

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —