FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Ein Technologieentwurf für eine Klasse skalierender Webapplikationen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Masterstudium Software Engineering/Internet Computing

eingereicht von

Paul Panhofer
Matrikelnummer 9725248

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Gerald Futschek, Univ. Doz.
Mitwirkung: -

Wien, 01.10.2013

(Unterschrift Verfasser) (Unterschrift Betreuer)

„Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.“

Kirchberg am Wagram, 3rd October 2013

Abstrakt

Mit dem Aufkommen von **Web 2.0** sind wir Zeuge einer dramatischen Änderung der geforderten **Anforderungen** an Web Applikationen und Softwaresysteme im Allgemeinen geworden. Wir erfahren eine neue Art von Applikationen, die einem veränderten Katalog nicht funktionaller Anforderungen gerecht werden muß. Im Zentrum der Arbeit steht der Versuch, eine im Sinne der technischen Umsetzung **skalierende Architektur** für eine Klasse von komplexen Softwareapplikationen zu finden. In diesem Zusammenhang ist die Differenzierung zwischen **horizontal** und **vertikal skalierenden** Lösungen essentiell und zu beachten. Als Stellvertreter für diese Klasse wird ein spezifisches Projekt gewählt, eine **Biber Webapplikation**.

Im Zuge der Arbeit schenken wir der **Datenverwaltung** und damit dem **Konsistenzmodell** besondere Aufmerksamkeit. Das **CAP Theorem** [12][16] dient in diesem Zusammenhang als fundamentaler Fokuspunkt, um die komplementären Größen und deren Beziehungen im Umfeld der Architekturdiskussion zu verstehen. Der Vergleich diverser bestehender **Technologien** und **Paradigmen** soll ein allgemeines Verständnis für das Problem und dessen Lösung schaffen. Die gefundenen Ergebnisse dienen als Basis zur Erstellung der **Softwarearchitektur**. Zur Evaluierung der Ergebnisse werden 2 **Prototypen** erstellt. Die Prototypen werden im Sinne widersprüchlicher Paradigmen entworfen und gegeneinader verglichen. Das Resultat hilft, die gewünschten Ergebnisse zu etablieren.

Die **Prototypen** weisen eine grundlegende Differenzierung in ihrem **Daten-**, **Konsistenz-** und **Replikationsmodell** auf. Ein Prototyp wird den Prinzipien der **NoSQL** Ansätze folgend gestaltet, der andere nutzt die Theorie der verteilten **relationalen Datenbanken** für die Datenverarbeitung. **Lastentests** werden eingesetzt um das Laufzeitverhalten der Prototypen zu bewerten und zu verstehen. Das Laufzeitverhalten ist in diesem Sinne als die Summe der verschiedenen umgesetzten Paradigmen für Datenkonsistenz, Replikation und Datenmodell zu beurteilen. Im Anschluß an die Lastentests werden die generierten Meßergebnisse ausgewertet und gesichtet. Die Meßergebnisse dienen als Grundlage zur Beurteilung der eingesetzten Modelle.

Wir haben ein **performantes Laufzeitverhalten** und die **Skalierbarkeit** des Systems als primäre **nicht funktionale Anorderung** an das System gestellt. Die zentrale Frage ist ob es möglich ist ein solches System unter der Verwendung eines **RDBMS** Systems als Kern umzusetzten. Betrachten wir die Ergebnisse für die **Skalierbarkeit** des MySQL Prototypen bemerken wir daß das System nur **horizontal skaliert**. Unser Anspruch besteht jedoch darin ein System zur Verfügung zu stellen, daß **linear skaliert**. Diesen Anspruch erfüllt aber nur der **NoSQL Prototyp**. Wir müssen feststellen daß eine **RDBMS** Lösung für unser Problem nicht in Frage kommt. □

Abstract

Web 2.0 is becoming the Buzz Word in the **Distributed Systems community**. The technological standards for Web Applications are continually changing resulting in a new stack of **non functional** requirements. The focus of the thesis is to find a **scaleable architecture** for a special class of distributed softwaresystems.

The processing of large amounts of data in a timely manner is key. The **CAP Theorem** helps us to establish a **classification** of dataprocessing distributed systems. This classification is key to understand the different **properties** and **requirements** in the process of developing the **software architecture** in question.

The comparison of different classes of dataprocessing systems gives us an idea of the problem. **RDBMS** and **NoSQL Systems** are the current dataprocessing technologies. We have created two **Softareprototypes**. One **prototype** has been created as a **NoSQL System** the other one is using **RDBMS** for dataprocessing. We have been running different tests to learn about the **performance, scalability** and **elastic speed up** of the prototypes.

We have learned that **RDBMS** as a dataprocessing technology is not able to give us the **scalability** we need. On the other hand **NoSQL** solutions are a tool that helps us to develop a **scalable** architecture. □

Contents

List of Figures

CHAPTER 1

Einführung

Problemstellung und Kontext der Arbeit

Web 2.0 Mit dem Aufkommen von **Web 2.0** sind wir Zeuge einer dramatischen Änderung der geforderten Anforderungen an **Web Applikationen** und **Softwaresysteme** im Allgemeinen geworden. Wir sehen eine neue Art von Applikationen, die einem verändertem Katalog nicht funktionaler Anforderungen gerecht werden muß.

Viele der heute weitverbreitesten Anwendungen, finden sich in der Liste dieser Anwendungen:

- **Facebook**
- **Amazon**
- **Twitter**
- **YouTube**
- ... □

kritische nichtfunktionale Anforderungen Das Laufzeitverhalten von Client Requests wird zum entscheidenden Thema. Anwendungen sehen sich mit der Herausforderung konfrontiert Hunderttausende und Millionen von Anwendern gleichzeitig zu verarbeiten.

Das Laufzeitverhalten der Anwendung versteht sich in diesem Zusmmenhang als Resultat folgender **Grössen** und **Konzepte**:

- **Datenverarbeitung**
- **Replikationsmodell**
- **Konsistenzmodell**
- **Datenschema**
- **Skalierbarkeit** □

Grundlegende Konzepte Gleichzeitig erkennen wir ein verändertes Verständnis dieser Konzepte im Lichte von **Web 2.0**.

- **Datenverarbeitung:** Im Vordergrund steht Datenverarbeitung im Tera und Peta Datenbereich. Das **Scale up** Potential relationaler Systeme stößt hier schnell an seine Grenzen. Erschwerend kommt hinzu daß Zugriffe nicht gleichverteilt über ein entsprechendes Zeitintervall sondern in extremen Lastspitzen auftreten.
- **Konsistenzmodell:** Der Stellenwert der strikten Konsistenzanforderungen und der damit verbundenen **ACID** Eigenschaften von Transaktionen tritt deutlich in den Hintergrund. Der User nimmt **eventual Consitency** in Kauf und erkauft sich damit ein wesentlich verbessertes Laufzeitverhalten und lineare Skalierbakeit des Systems.

  Neue Prinzipien im Datenverarbeitungsbereich werden erdacht und entwickelt um das enge Korsett der **ACID** bedingenten **Konsistenzanforderungen** abzulegen. Die grundlegende Idee ist hier, die Restriktionen relationaler Datenbanken aufzubrechen und andere Wege in der Datenverarbeitung zu beschreiten.
- **Datenschema**: In traditionellen Datenbanksystemen wird beim Einspielen von Daten ein Schema erzwungen. Passen die Daten nicht zum Schema werden diese verworfen. Dieser Ansatz wird auch **Schema On-Write** genannt da die Daten beim Schreiben gegen das Schema geprüft werden.

  Verstärkt ist der Wunsch von Software Entwicklern zu bemerken die Konventionen eines **Schema On-Write** zugunsten eines aufgelockerten Schemaverständnisses aufzugeben. □

Alternative Ansätze Um den veränderten nichtfunktionalen Anforderungen des **Web 2.0** gerecht zu werden müßen Lösungen gefunden werden die uns helfen uns über die gegebenen technischen Restriktionen und Paradigmen hinwegzusetzen.

Zum besseren Verständnis dieses Ansatzes sollen 2 Beispiele aus der Praxis diskutiert werden:

- **SETI@home:** **SETI@home** ist eine Anwendung die eine immense Rechenleistung aufbringt, in dem es seine Rechenlast auf beliebige Rechner im Internet verteilen kann. (siehe[20])
- **Hadoop:** **Hadoop** ist eine Framework, das in der Lage ist, komplexe Operationen auf Datenmengen beträchtlichen Umfangs durchzuführen

Trotz der Ähnlicheit der Anforderungen an die Anwendungen, erkennen wir zwei Lösungsstrategien, die unterschiedlicher nicht sein könnten. SETI@home muß als Resultat seiner Softwarearchitektur wiederholt beträchtliche Datenmengen zwischen dem Server und den Clients austauschen. Im Gegensatz dazu verfolgt Hadoop den Ansatz, den Anwendungscode zu den Clients zu bringen und die Netzwerklast damit so gering wie möglich zu halten.

Im Zuge dieser Abhandlung soll ein **Technologieententwurf** für eine bestimmte Klasse von Webapplikation entwickelt werden. Stellvertrettend für diese Klasse von Webapplikationen wird ein System im Kontext des Wettbewerbs des **Bibers der Informatik** entwickelt und getestet. Im Sinne eines wissenschaftlichen Diskurses wird zuerst versucht, die Anforderungen an das System zu erarbeiten. Das Verständnis der Anforderungen soll helfen, die kritischen Bereiche der Architektur zu erkennen und entsprechend zu entwerfen. Mehrere Technologien werden in diesem Zusammenhang verglichen und ihre Tauglichkeit für das System bewertet. □

Methodisches Vorgehen

Zur Diskussion und Etablierung unserer Ergebnisse zum **Technologieentwurf** des gewünschten Systems werden die folgenden Schritte durchlaufen werden:

1. **Definition der gewünschten Klasse von Softwareanwendungen**
2. **Entwicklung der Prototypen**
3. **Durchführung der Lastentests**
4. **Etablierung und Diskussion der Ergebnisse** □

Klasse von Softwareanwendungen

Die für uns interessante **Klasse von Softwareanwendungen** möchten wird durch die für sie charakteristischen Eigenschaften beschreiben. Die gewünschte Klasse findet sich in der **Durchschnittsmenge** der durch folgende Eigenschaften und Anforderungen beschriebenen Softwaresysteme:

- □ **verteilte System**
- □ **basierend auf dem HTTP Protokoll**
- □ **Unterstützung für HTML 5**
- □ **hoch skalierbar**
- □ **Datenverarbeitung im Terrabyte Bereich**
- □ **performantes Laufzeitverhalten** □

Prototypen

Prototypen Stellvertretend für die im Zuge der Diplomarbeit interessante Klasse von Softwareanwendungen, sollen 2 Prototypen erstellt werden. Die Prototypen werden unterschiedlichen Konzepten und Paradigmen folgend entworfen und implementiert. Ein Prototyp wird als **RDBMS** (**Realtional Database Management System**) umgesetzt der andere verwendet **NoSQL** als Datenverarbeitungsschicht. Beide Lösungen müßen einem bestimmten Katalog **nicht funktionaler** Anforderungen gerecht werden. Der Vergleich dieser unterschiedlichen Technologien und Paradigmen soll ein allgemeines Verständnis für das Problem und dessen Lösung schaffen. □

Prototypenarchitektur - Schichtenmodell Um eine Vergleichbarkeit der Prototypen zu gewährleisten, wird die Architektur der beiden Programme dem **Schichtenmodell** folgend gestaltet. Gelingt es uns, in beiden Prototypen einen Großteil der Schichten identisch zu halten, kann ein legitimer Vergleich der Programme durchgeführt werden.

Das **Schichtenmodell** etabliert einer Reihe von Prinzipien, die die Durchführung eines Vergleichs begünstigen:

- **Trennung der Funktionalität:** Jede Schicht implementiert unterschiedliche Funktionalität. Die Idee ist, die Funktionalität der einzelnen Schichten paarweise disjunkt zu halten.
- **Interfaces:** Die Funktionalität jeder Schicht wird dem Client über ein Interface zur Verfügung gestellt. Eine strikte Trennung der Interfaces von der Implementierung hilft uns einen generische Softwarearchitektur aufzubauen.
- **Zugriff:** Jede Schicht kann nur auf das Interface der benachbarten Schichten zugreifen. Dieser hierarchische Aufbau etabliert eine strikte Trennung der Funktionalität und Modularisierung des Systems.
- **Modularisierung:** Die Unterteilung der Anwendung in ein System hierarchischer Schichten etabliert gleichzeitig eine Modularisierung der Anwendung.
- **Abstraktion:** Mit jeder Schicht im Schichtenmodell steigt der Abstraktionsgrad der Software. Der durch die Abstraktion geschaffene Kontext erlaubt eine triviale Implementierung komplexer Aufgabenstellungen. □

Implementierung

MVC Modell Das **MVC Modell** stellt eine Interpretation des Schichtenmodells dar, eingesetzt zur Strukturierung bestimmter Klassen von Anwendungen.

Die einzelnen Schichten des Modells sind:

- **Model:** Das **Model** als Schicht ist für die Datenverarbeitung der Anwendung verantwortlich
- **View:** Das **View** dient als Anwenderschnittstelle. Diese Schicht erlaubt den Zugriff des Anwenders auf die Funktionalität des Programmes.
- **Controller:** Der **Controller** dient als Mediator zwischen **Model** und **View**.

Beide Prototypen implementieren das **MVC Modell**. Die Herausforderung in der Gestaltung beider Programme ist es sowohl das **View** als auch den **Controller** für

beide identisch zu halten. Lediglich in der Model Schicht streben wir eine grundlegende Differenzierung im Sinne der Implementierung an.

Die Prototypen verwenden unterschiedliche Ansätze zur Implementierung ihrer **Daten-**, **Konsistenz-** und **Replikationsmodelle**. Ein Prototyp wird den Prinzipien der **NoSQL** Ansätze folgend gestaltet, der andere nutzt die Theorie der verteilten relationalen Datenbanken für die Datenverarbeitung. □

Lastentest

Laufzeitverhalten Lastentests werden eingesetzt, um das **Laufzeitverhalten** der Prototypen zu bewerten und zu verstehen. Das Laufzeitverhalten ist in diesem Sinne als die Summe der unterschiedlichen Paradigmen und Konzepte für **Datenkonsistenz**, **Replikation** und des **Datenmodells** zu beurteilen.

Beide Prototypen werden denselben Lastentests unterzogen. Der Vergleich der Ergebnisse der Lastentests gibt Auskunft über die Beschaffenheit des eingesetzten **Technologieentwurfs**. Die gefundenen Ergebnisse helfen uns, die gewünschten Eigenschaften zu prüfen und die gewünschten Architekturentscheidung zu treffen. □

Resultat

Die durch die Lastentests generierten Ergebnisse dienen uns als Grundlage für den gewünschten **Technologieentwurf**. Zu diesem Zeitpukt soll gleichzeitig das Ergebnis geprüft werden und der Einfluß der unterchiedlichen Konzepte auf das Ergebnis besprochen werden. □

Technologievergleich

CAP Theorem

CAP Theorem **Das CAP Theorem etabliert, dass es für ein verteiltes System unmöglich ist, die 3 CAP Eigenschaften (Konsistenz, Verfügbarkeit und Robust gegenüber Knotenausfällen) gleichzeitig zu erfüllen.**

Brewer zeigt mit seinem bekannten **CAP-Theorem** die **Entwurfsgrenzen** für ein verteiltes System auf. Das **CAP Theorem** [12][16] dient in diesem Zusammenhang als fundamentaler **Fokuspunkt** um die komplementären Größen und deren Beziehungen im Umfeld der Architekturdiskussion verteilter Systeme zu verstehen. Gleichzeitig wird eine **Klassifizierung** für verteilte Systeme im Datenverarbeitungsbereich gegeben. Die Erkenntnisse des CAP Theorems spielen für die Gestaltung verteilter Datenverarbeitungssysteme damit eine zentrale Rolle. □

Konsistenz, Verfügbarkeit und Partitionstoleranz

- **Consistency** - **Konsistenz** aller Netzwerkknoten: Alle Knoten sehen zur selben Zeit dieselben Daten. In verteilten Systemen mit replizierten Daten muß sichergestellt sein, dass nach Abschluß einer Transaktion auch alle Replikate des manipulierten Datensatzes aktualisiert werden.
- **Availability** - **Verfügbarkeit**, alle Anfragen an das System können beantwortet werden.
- **Partition Tolerance** - **Partitionstoleranz**, das System arbeitet auch bei Verlust von einzelnen Knoten oder dem Ausfall von Teilsystemen, weiter. Das gesamte System muß stabil laufen unabhängig davon, ob ein Knoten geplant oder ungeplant ausfällt. □

CAP Theorem - Diskussion Es läßt sich leicht erkennen, daß je nach eingesetztem System, entweder die **Verfügbarkeit** oder die **Konsistenz** der Daten beim Eintreten eines **Partition Events** nicht vollständig gewährleistet werden kann. Es sind damit immer nur 2 der 3 geforderten Eigenschaften **gleichzeitig** erfüllbar.

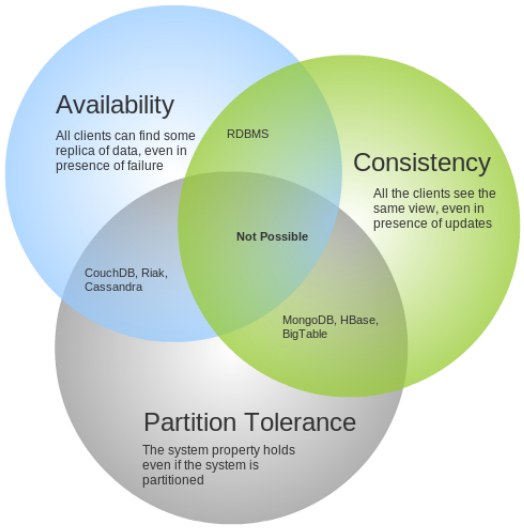
Ein System, das alle drei Eigenschaften zur selben Zeit erfüllt, müsste demnach bei einer Unterbrechung der Verbindung zwischen den Knoten die separierten Knoten verfügbar halten und in diesem Sinne konsistent auf Anfragen reagieren. Wenn nun Daten auf einem der Knoten geändert werden, können offenbar nicht alle restlichen Knoten konsistent gehalten werden, da ja manche Knoten von den anderen Knoten aus nicht

mehr erreichbar sind. Das System kann nun entweder INKONSISTENT oder GAR NICHT REAGIEREN, müsste also entweder **Consistency** oder **Availability** aufgeben. □

Klassifizierung von Applikationen

Da stets nur 2 der 3 Eigenschaften gleichzeitig erfüllt sein können, finden wir eine Klassifizierung von Anwendungen in 3 disjunkte Klassen.

- **CA Systeme:** **Konsistenz** und **Verfügbarkeit** zur selben Zeit gewährleistet
- **CP Systeme:** **Konsistenz** und **Partitionstoleranz** zur selben Zeit gewährleistet
- **AP Systeme:** **Verfügbarkeit** und **Partitionstoleranz** zur selben Zeit gewährleistet



CA Systeme Beschreibt die Klasse der **RDBMS** Systeme.

- **Traits:** **2 Phase Commit**, **Cache Validation Protocolls**, **strenge Konsistenz**
- **Verzicht auf Partition Tolerance:** Kann **Partition Tolerance** nichstt mehr gewährleistet werden, muß das System auf einer einzelnen Serverinstanz ausgeführt werden.
- **Vertreter:**
  - **RDBMS Systeme**

  - **Singel-Site DB**
  - **Cluster DB**
  - ...□

CP Systeme Beschreibt eine Klasse von **NoSQL** Systemen.

- **Verzicht auf Availability:** Ereignet sich ein Partition Event müssen betroffene Services warten, bis Daten konsistent sind und sind erst nachher wieder verfügbar.
- **Vertreter:**
  - **BigTable**
  - **MongoDB**
  - **HBase**
  - ...□

AP Systeme: Beschreibt eine Klasse von **NoSQL** Systemen.

- **Verzicht auf Datenkonsistenz**
- **Vertreter:**
  - **Cassandra**
  - **CouchDB**
  - **DNS**
  - ...□

Technologieentscheidung Bei der Erstellung der Prototypen stellen wir die **NoSQL Systeme** **(CP- und AP Systeme)** den **RDBMS Systemen** **(CA Systeme)** gegenüber.

RDBMS vs NoSQL

Wir isolieren das **Laufzeitverhalten** und die **Skalierbarkeit** der Prototypen als die entscheidenden nichtfunktionalen Anforderungen in unserer Klasse von Softwareystemen. Ein performantes **Laufzeitverhalten** in Verbindung mit der Notwendigkeit, **Millionen von Benutzerrequests** zur selben Zeit zu bearbeiten, ist die Quadratur des Kreises, die es zu meistern gilt.

	NoSQL Systeme	RDBMS Systeme
Skalierbarkeit:	horizontale Skalierbarkeit	vertikale Skalierbarkeit
Konsistenzmodell:	**ACID**	**BASE**
Datenschema:	schemaless	Schema
Datastore:	**RDBMS**	**NoSQL**

Figure 1.1: Konzeptvergleich

Eine Vielzahl von Requests gleichzeitig zu bearbeiten, birgt die Notwendigkeit, die Anwendung auf einen Cluster von Rechnern zu verteilen. Zur nichtfunktionalen Anforderung des Laufzeitverhaltens gesellt sich die Forderung nach einem skalierenden System.

Das **Laufzeitverhalten** und die **Skalierbarkeit** des Systems sind Größen, die in engem Zusammenhang mit anderen Grössen und Konzepten stehen. Zur Etablierung unserer Ergebnisse entwickeln wir einen **CP-** und **CA Prototypen**. Die beiden Prototypen werden nach grundlegend unterschiedlichen Paradigmen und Technologiekonzepten entwickelt.

Skalierbarkeit

Skalierbarkeit wird eingesetzt, um die Verfügbarkeit von Anwendungen im industriellen Umfeld gewährleisten zu können. Ein Sofwaresystem wird als **skalierbar** bezeichnet, wenn das System in der Lage ist gleichzeitig zur wachsenden **Systemlast** den vereinbarten Dienst zu gewährleisten.

Generell wird unterschieden zwischen

- **horizontale/lineare Skalierung: Horizontale Skalierung** ist die komplexere der beiden Skalierungsformen. Ein **horizontal skalierendes** Softwaresystem ist in der Lage bei beliebiger Systemlasten den Dienst des Systems zur Verfügung zu stellen. Die Lösung wird nicht durch teurere Hardware erkauft, sondern durch einen entsprechenden **Technologieentwurf** erreicht.

  Bei dieser Art des Entwurfs wird die Software des Systems an die Hardware angepasst.

- **vertikaler Skalierung: Vertikale Skalierung** ist die einfachere der beiden Formen der Skalierung. Im Grunde besteht die Lösung darin die Applikation auf leistungsfähigere Computer zu verlagern. Die Skalierung wird damit

eine Frage der Finanzierung. Teurere Harware erkauft eine temporäre Lösung mit dem Vorbehalt, daß der erfolgreichen Verarbeitung in letzter Instanz physikalische Grenzen auferlegt sind.

Bei dieser Art von Softwaresystemen wird die Hardware der Software angepasst

Konsistenzmodell: ACID vs BASE

Die Entscheidung für ein **Konsistenzmodell** ist fundamental für das Verhalten einer Applikation und für die Gestaltung ihrer Architektur.

Klassifizierung von Konsistenzmodellen

- **Strict Consistency:** Alle Leseoperationen liefern den Wert, der die letzte abgeschlossene Schreiboperation geschrieben hat, egal auf welchem Node die Operationen ausgeführt werden.

  Bei verteilten Systemen braucht es dafür ein verteiltes Transaktions - Protokoll.

- **Eventual Consistency:** **Eventual Consistency** gehört zur Kategorie der schwachen Konsistenzmodelle. Eine Leseoperation liefert eventuell den zuletzt geschriebenen Wert. Je nachdem, an welchem Node der Client die Anfrage schickt, sieht er einen inkonsistenten Zustand.

- **Functional Partitioning:** **Functional Partitioning** ist eine Möglichkeit eine **skalierbare Lösung** für ein Softwaresystem umzusetzen. **Functional Partitioning** besteht darin, das Datenschema als logische Distribution der Daten in funktionale Einheiten zu strukturieren. Daten die in funktionaler Weise zusammenhängen, werden dabei in einer Serverinstanz gruppiert. Die Datenverarbeitung innerhalb von Transaktionen kann in diesem Sinne schnell und effizient abgewickelt werden. □

ACID Je nachdem für welche Art von **Konsistenz** wir uns entscheiden können wir eine Zahl von Eigenschaften fordern. Entscheiden wir uns für **Strict Consistency**, werden die **ACID** Eigenschaften gefordert:

- **Atomicity:** Abgeschlossenheit - Jede Operation innerhalb einer Transaktion wird vollständig abgeschlossen (oder alle Operationen werden wieder rückgängig gemacht)

- **Consistency:** Konsistenz - Bei Transaktionsstart und -ende sind die Daten in einem **konsistenten** Zustand.

- **Isolation:** Abgrenzung - Die Transaktion verhält sich so, als wäre sie die einzige Operation, die auf dem Datenbestand ausgeführt wird.
- **Durability:** Dauerhaftigkeit - Ist eine Transaktion einmal abgeschlossen, sind die Veränderungen dauerhaft im System gespeichert.

Ein Konsistenzmodell, das die **ACID** Eigenschaften erfüllt, ist in der Lage, komplexe Datenshemen und deren Abhängigkeiten abzubilden. □

BASE **Web 2.0** Systeme müssen in erster Linie performant und zuverlässig sein.

Eigenschaften:

- **Basically available:** Die Applikation ist grundsätzlich immer verfügbar
- **Soft state:** Die Applikation muss nicht immer in einem **konsistenen Zustand** sein
- **Eventual Consistency:** Nach dem Verständnis von **BASE** wird die **Konsistenz** der Daten als ein Zustand betrachtet, der irgendwann erreicht wird. Das ist die Idee des **Eventual Consistency**. Es wird damit in Kauf genommen, dass sich bis zum Erreichen der Konsistenz, die Daten in einem **inkonsistenten Zustand** befinden.

Die Hochverfügbarkeit der Daten geht zu Lasten der **Konsistenz**. Die Daten sind im Anschluß an einen Schreibvorgang nur **eventuell konsistent**. Erst nach dem Verstreichen einer gewissen Zeit wird die Änderungen an alle anderen Replikate weitergegeben. □

Vergleich ACID vs BASE

ACID	BASE
strenge Konsistenz	schwache Konsistenz
Isolation	Verfügbarkeit
Fokus auf commit	Inetwa richtige Antworten
verschachtelte Transaktionen	schnell
konservativ/pessimistisch	aggressiv/optimistisch
komplizierte Evolution	einfache Evolution

Datenverarbeitung

DB Verteilte Datenbanksysteme kommen zum Einsatz wenn ein Rechner allein, als Datenbankserver, die anfallenden Aufgaben nicht mehr bewältigen kann.

Stand Alone Datenbanksysteme stoßen schnell an ihre Grenzen falls:

- **Volumen:** Das Volumen der zu speichernden Daten eine kritische Grenze übersteigt
- **Zahl:** Die Zahl der Requests an den DB Server eine kritische Grenze übersteigt
- **Replikation:** Die Daten im Sinne von Datensicherheit über mehrere Rechner repliziert werden □

Datenverarbeitung Vertreter für die, für uns interessante Klasse von Softwaresystemen müssen in der Lage sein, in kürzester Zeit immense Datenmengen **zu bewegen** und **zu verarbeiten**. Wir erkennen, daß die **Datenverarbeitung** eine kritische Größe für das Verständnis der Anforderungen an den **Technologieentwurf** darstellt.

Finden wir eine effiziente Architektur, können wir **Salierbarkeit** und ein **performates Laufzeitverhalten** garantieren. □

NoSQL Eine Entwicklung im Datenverarbeitungsbereich, bekanntgeworden als **NoSQL** Datenbanken, wird vorangetrieben, um ein performantes **Laufzeitverhalten** und **Skalierbarkeit** im Industriebereich zu schaffen. Die grundlegende Idee ist hier, die Restriktionen relationaler Datenbanken zu umgehen und andere Wege in der Datenverarbeitung zu beschreiten.

Die Abkehr von etablierten Datenverarbeitungstechniken hin zu neuen, unkonventionellen Lösungen steht im Mittelpunkt dieses Ansatzes. **ACID** als Ankerpunkt des **Konsistenzmodell** wird durch die neu formulierten **BASE** Kriterien abgelöst. **NoSQL** verfolgt den Ansatz daß eine zeitliche Lockerung der Konsistenzkriterien die gewünschten Ergebnisse bringt, ohne geforderte Softwareeigenschaften zu verletzen. □

Fragestellung der Arbeit

Vertikale Skalierung vs. horizontale Skalierung Die bestimmende technologische Herausforderung der letzten Jahre in der Softwareentwicklung war die Eskalierung der zu bearbeitenden Datenmengen in einer entspechenden vorgegebenen Zeit. Etablierte verteilte RDBMS Systeme stossen in diesem Zusammenhang zunehmend an ihre Grenzen. Neue Konzepte müssen entwickelt werden, um den geänderten Anforderungen gerecht zu werden.

Die grundlegende Frage in diesem Sinne ist, ob **vertikal skalierende** Systeme in diesem Bereich, verworfen werden müssen und durch **horizontal Skalierende** Lösungen ersetzt werden sollen? □

Fragestellung Die grundlegende Frage in diesem Zusammenhang ist damit:

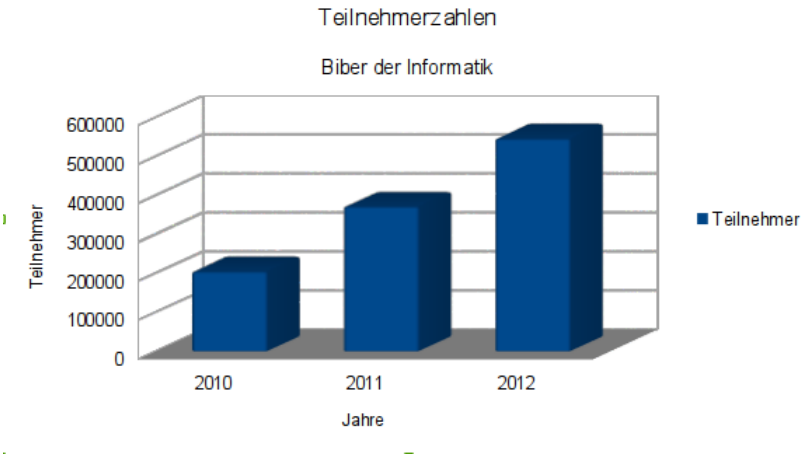
Soll zur Etablierung eines verteilten webbasierten Client-Server Systems mit Millionen von gleichzeitigen Anwenderzugriffen und der Notwendigkeit Daten im Peta Byte Bereich zu bearbeiten bei gleichzeitig performantem Laufzeitverhalten, ein vertikal oder ein horizontal skalierendes System eingesetzt werden? □

Das Biber Softwaresystem - Eine Beschreibung

Beschreibung und Geschichte des "Bibers"

Der Informatik Biber (Bebras International Contest on Informatics and Computer Literacy) ist ein jährlich stattfindender Informatik Wettbewerb. Der Wettbewerb wird durchgeführt in Volks- und Mittelschulen. Am Computers lösen die Teilnehmer 15 bis 21 Aufgaben. Die zu lösenden Aufgaben umfassen dabei interaktive und Multiplechoice Aufgaben.

Die Zahl der am Wettbewerb teilnehmenden Länder wächst jährlich. Mit der zunehmenden Zahl an Teilnehmern wächst die Notwendigkeit nach einer skalieren Lösung für das System.



Geschichte des Bibers

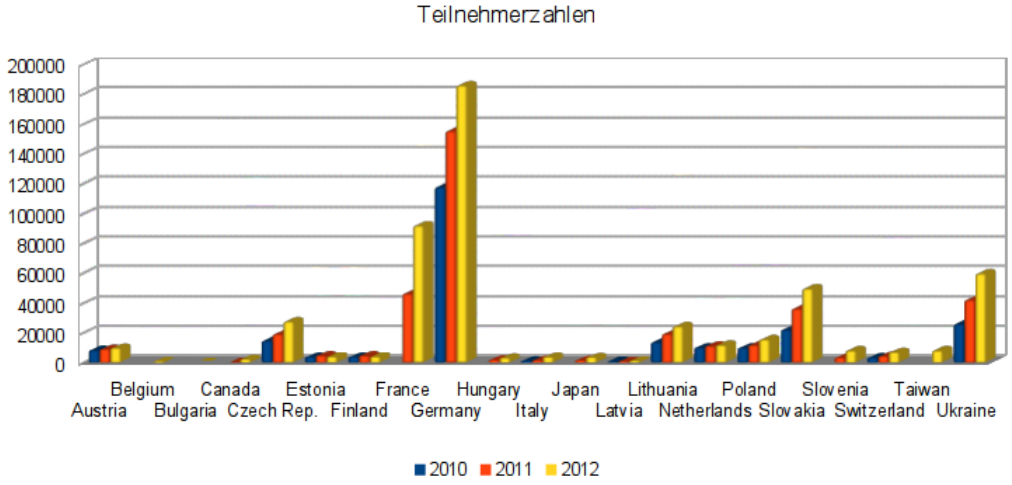
- Der Wettbewerb wurde zum ersten Mal **2004** unter dem Namen Bebras in Litauen durchgeführt.
- Im Jahre **2005** wurde Beaver in
  - Estland,
  - Lettland,
  - Litauen,
  - den Niederlanden,

  - Polen durchgeführt.

- **2006** wurde der Informatik-Biber im Rahmen des Informatikjahres unter dem Namen EI:SPIEL blitz! erstmals in Deutschland durchgeführt.

- **2007** fand der Informatik-Biber in Deutschland, Österreich, Polen, Litauen und den Niederlanden statt.

- Im Jahr **2010** fand der Informatik-Biber zum ersten Mal auch in der Schweiz statt.



Usecasebeschreibung

Zur transparenten Darstellung der Abhängigkeiten und Eigenschaften des Biber Softwaresystems sollen die Usecases des Systems näher erörtert werden. Das Verständnis der Aktoren, deren Rechte und Abhängigkeiten, sollen uns helfen die Anforderungen an das System besser zu verstehen.

Aktoren-Rechte Im folgenden finden wir das Zusammenspiel der Rechte und Aktoren

- siehe Abbildung 1.2

- siehe Abbildung 1.3

- siehe Abbildung 1.4

ID	Acronym	Bescreibung
1.1	CPA	Anlegen eines Proxy-Administrators
1.2	UPA	Angleichen eines Proxy-Administrators
1.3	DPA	Löschen eines Proxy-Administrators aus dem System
2.1	CTA	Anlegen eines Test-Administrators
2.2	UTA	Angleichen eines Test-Administrators
2.3	DTA	Löschen eines Test-Administrators aus dem System
3.1	CSA	Anlegen eines School-Administrators
3.2	USA	Angleichen eines School-Administrators
3.3	DSA	Löschen eines School-Administrators aus dem System
4.1	CS	Anlegen einer Schule
4.2	US	Angleichen der Daten einer Schule
4.3	DS	Löschen einer Schule aus dem System
5.1	CP	Anlegen eines Schülers
5.2	UP	Angleichen der Daten einer Schülers
5.3	DP	Löschen eines Schülers aus dem System
6.1	CSS	Anlegen von Schulverwaltungsdaten
6.2	USS	Angleichen von Schulverwaltungsdaten
6.3	DSS	Entfernen von Schulverwaltungsdaten aus dem System
7.1	CSS	Anlegen von Schulverwaltungsdaten
7.2	USS	Angleichen von Schulverwaltungsdaten
7.3	DSS	Entfernen von Schulverwaltungsdaten aus dem System
8.1	CE	Anlegen einer Aufgabe
8.2	UE	Angleichen einer Aufgabe
8.3	DE	Entfernen einer Aufgabe aus dem System
9.1	CC	Anlegen einer Testkategorie
9.2	UC	Angleichen einer Testkategorie
9.3	DC	Entfernen einer Testkategorie aus dem System
10.1	CE	Anlegen einer Testevaluierung
10.2	UE	Angleichen einer Testevaluierung
10.3	DE	Entfernen einer Testevaluierung aus dem System
11.1	CT	Anlegen einer Tests
11.2	UT	Angleichen eines Tests
11.3	DT	Entfernen eines Tests aus dem System
12.1	ST	Test ausführen

Figure 1.2: Aktorenrechte

Aktor	Rechte
System-Administrator	**{CPA, UPA, DPA}, {CTA, UTA, DTA}, {CSA, USA, DSA}, {CS, US, DS}, {CP, UP, DP}, {CSS, USS, DSS}, {CE, UE, DE}, {CC, UC, DC}, {CE, UE, DE}, {CT, UT, DT}, ST**
Proxy-Administrator	**{CTA, UTA, DTA}, {CSA, USA, DSA}, {CS, US, DS}, {CP, UP, DP}, {CSS, USS, DSS}, {CE, UE, DE}, {CC, UC, DC}, {CE, UE, DE}, {CT, UT, DT}, ST**
Test-Administrator	**{CSA, USA, DSA}, {CS, US, DS}, {CP, UP, DP}, {CSS, USS, DSS}, {CE, UE, DE}, {CC, UC, DC}, {CE, UE, DE}, {CT, UT, DT}, ST**
School-Administrator	**{CS, US, DS}, {CP, UP, DP}, {CSS, USS, DSS}, {CE, UE, DE}, {CC, UC, DC}, {CE, UE, DE}, {CT, UT, DT}, ST**
Test Participant	**ST**

Figure 1.3: Aktorenliste

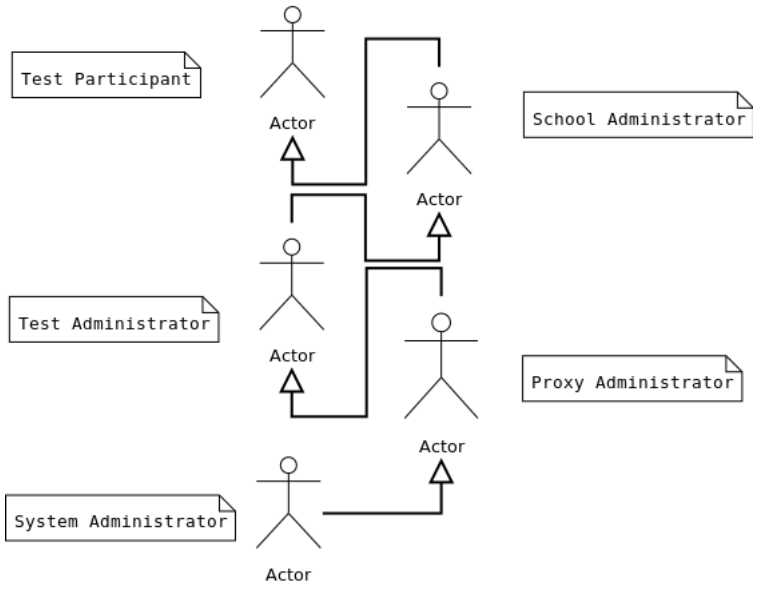


Figure 1.4: Beziehung unter Aktoren

Systembeschreibung - Anforderungen

Die für die Durchführung des Wettbewerbs notwendigen technischen Restriktionen sollen kurz besprochen werden.

Webtechnologie Das System hat den vorgegebenen technischen infrastrukturellen Unzulänglichkeiten der pädagogischen Einrichtungen Rechnung zu tragen. Die Herausforderung besteht darin die Applikation auf einer Zahl von rudimentären Internet-Technologien aufzubauen.

- **HTTP**: **H**yper **T**ext **T**ransfer **P**rotocol (Quelle:[18])
  Ist ein Protokoll zur Übertragung von Daten über ein Netzwerk. Es wird hauptsächlich eingesetzt, um Webseiten aus dem World Wide Web in einen Webbrowser zu laden.

  HTTP gehört der sogenannten Anwendungsschicht etablierter Netzwerkmodelle an. Die Anwendungsschicht wird von den Anwendungsprogrammen angesprochen, im Fall von HTTP ist das meist ein Webbrowser. Im ISO/OSI-Schichtenmodell entspricht die Anwendungsschicht den Schichten 5–7. HTTP ist ein zustandsloses Protokoll.

  Ein zuverlässiges Mitführen von Sitzungsdaten kann erst auf der Anwendungsschicht durch eine Sitzung über eine Session-ID implementiert werden.

- **Client Technologie**: Client-Technologie sind lediglich bezüglich des Einsatzes von Plugins beschränkt.

  Gängige Browser werden durchgehend unterstützt.

  - **Firefox**
  - **Chrome**
  - **Opera**
  - **Internet Explorer**
  - ...

- **HTML 5** (Quelle:[19])
  HTML 5 ist eine textbasierte Auszeichnungssprache zur Strukturierung und semantischen Auszeichnung von Inhalten wie Texten, Bildern und Hyperlinks in Dokumenten.

  HTML 5 ist als Sammlung mehrerer Technologien, die im Tandem arbeiten, zu verstehen.

  - **HTML**
  - **JavaScript**

  - **CSS**

- **Servertechnologie**
  Die Wahl der einzusetzenden Severtechnologie obliegt keinen Restriktionen.

Die geforderten Restriktionen sollen gewährleisten, daß auch Schulen mit veralteter Ausstattung am Wettbewerb teilnehmen können.

CHAPTER 2

Anforderungsanalyse

Klasse von Softwareanwendungen

Die für uns interessante **Klasse von Softwareanwendungen** möchten wird durch die für sie charakteristischen Eigenschaften beschreiben. Die gewünschte Klasse findet sich in der **Durchschnittsmenge** der durch folgende Eigenschaften und Anforderungen beschriebenen Softwaresysteme:

- **verteiltes System**
- **basierend auf dem HTTP Protokoll**
- **unterstützt HTML 5**
- **hoch skalierbar**
- **Datenverarbeitung im Terrabyte Bereich**
- **performantes Laufzeitverhalten** □

Verteilte Systeme

Die Biber Webapplikation ist eine verteilte Softwareanwendung. Sprechen wir von einem verteilten System das Daten verarbeitet, müssen wir einige Eigenschaften fordern: (siehe [10])

1. **Hohe Verfügbarkeit**
2. **Ortstransparenz**
3. **Replikationstransparenz**
4. **verteilte Anfragebearbeitung**
5. **verteilte Transparenzbearbeitung**
6. **Hardwareunabhängigkeit**
7. **Betriebsystemunabhängigkeit**
8. **Leistungstransparenz** □

World Wide Web

Http Protokoll - Client/Server Haben wir es mit einem System zu tun, das im Bereich des Internets zum Einsatz kommen soll, müssen zusätzlich zu den Vorgaben eines verteilten Systems weitere Eigenschaften gefordert werden.

Sehen wir uns mit der Herausforderung konfrontiert, ein Client-Server System für das Internet zu entwickeln, gibt es eine Reihe von technischen Einschränkugen, die wir erfüllen müssen. In erster Linie leiten sich diese Einschränkungen aus den Eigenschaften des **HTTP Protokolls** ab. □

Eigenschaften des HTTP Protokolls

- **Zustandslos**
- **Request-Response Kommunikation**
- **Client-Server Architektur**
- **MIME Datentypen Verständnis** □

Anwendungsanforderungen

Webanwendung Die Biber Webapplikation selbst stellt einige grundlegende Anforderungen an die Gestaltung der Prototypen. Der gleichzeitige Zugriff mehrerer tausend Teilnehmer auf den Dienst der Webapplikation resultiert in der Formulierung kritischer Anforderungen an das System.

- **Webanwendung:** Die Anwendung ist als Client-Server System für das Internet umzusetzen. In diesem Zusammenhang müssen wir alle Restriktionen beachten die bei der Gestaltung des Prototyps im Umfeld des Internets zu befolgen sind.

- **Skalierbarkeit:** Der Spezifikation entsprechend muß das System in der Lage sein, tausende von Anfragen gleichzeitig zu verarbeiten. Von Jahr zu Jahr steigt die Zahl der am Wettbewerb teilnehmenden Schulen. Gleichzeitig finden immer mehr Länder den Weg in den Wettbewerb.

  Die Prototypen müssen in der Lage sein, zu skalieren, um die anfallende Serverlast zu bewältigen.

- **performante Datenverarbeitung:** Die Prototypen müssen in der Lage sein, in kürzester Zeit immense Datenmengen zu bewegen. Wir erkennen, daß die Datenverarbeitung eine kritische Größe für das Verständnis der Anforderungen an die Architektur darstellt. Finden wir eine effiziente Architektur, können wir Salierbarkeit und ein performantes Laufzeitverhalten garantieren.

  Eine ausführliche Diskussion zu diesem Thema soll uns helfen, ein Verständnis für die geeignete Lösung zu etablieren.

- **Replikationsfähigkeit:** Aus der Forderung nach Skalierbarkeit des Systems erwächst die direkte Forderung nach einem Replikationsmechanismus innerhalb des Systems. Replikation ist eine der Technologien, um Skalierbarkeit zu etablieren.

- **performantes Laufzeitverhalten:** Das Laufzeitverhalten einer Applikation ist das direkte Resultat der gewählten **Replikationsstrategie**, des **Konsistenzmodells** und des **Datenschemas**. □

CHAPTER 3

NoSQL: Etablierung eines neuen Datenverarbeitungs Standards

Einführung

Definition: NoSQL Bislang gibt es keine **einheitliche** und **abgrenzende** Definition für **NoSQL Datenbankensysteme**, dafür ist das Technologieparadigma noch zu neu.

Hier jedoch ein Versuch einer Definition wie sie von Edlich [7] vorgenommen wurde:

- **Kein relationales Datenmodell:** Postuliert wird eine Vielzahl neuer Modelle mit dem Gedanken, daß das **relationale Modell** in bestimmten Bereichen an seine Grenzen stößt oder nicht zweckmäßig ist.

- **Eignung für Systeme mit vertikaler und horizontaler Skalierbarkeit:** Schnelle Reaktionszeiten auf **Anfragen** und **Manipulationen** lassen sich nur mit dem **Scale-out** Prinzip erreichen, also mit **horizontaler Skalierung** durch dynamisches Einbinden/Löschen von Rechnerknoten (*Nodes*). Die gängige Praxis für RDBS hingegen ist das **Scale up** Prinzip, bei dem ein Rechner immer weiter technisch aufgerüstet wird.

- **Schemafrei:** Zur Bewältigung der Datenverwaltungsaufgaben mit ständig wachsendem Datenvolumen, in kürzest möglicher Zeit wird die Schemainformation in die Anwendung verlagert weg vom Datenbanksystem.

- **Einfache Datenreplikation zur Unterstützung der verteilten Architektur:** Die Grundidee der Verteilung des Datenbestands auf viele Knoten erfordert eine einfache und schnelle **Replikation** der Daten. Die Daten müßen auch bei Knotenausfällen weiter verfügbar sind. Möglich wird dies durch „abgeschwächte“ Konsistenzanforderung. Wir sehen eine praktische Umsetzung im **BASE-Konsistenzmodell**.

- **Ein angepasstes Konsistenzmodell:** Kein **ACID** als Konsistenzmodell □

Technologieparadigmen

NoSQL Datenbanken bieten eine Fülle von Eigenschaften, die für die Entwicklung verteilter, datenverarbeitender Systeme vorteilhaft sind. Einige der Eigenschaften sind:

- **Gutes Laufzeitverhalten**
- **Skalierbarkeit**
- **effizientes Replikationsmodell** und **Partition Tolerance**

Diese Eigenschaften werden erreicht durch die Etablierung unterschiedlicher **Technologieparadigmen**. Die wichtigsten dieser **Technologieparadigmen** sollen hier näher beschrieben werden. □

Consistent Hashing

Consistent Hashing Funktionen **Consistent Hashing Funktionen** als zentrales Konzept von **NoSQL** Systemen verstehen sich als eine Weiterentwicklung von Hashing Funktionen im Datenverarbeitungsbereich.

Das Konzept der **Consistent Hashing Funktionen** trägt dem Umstand Rechnung daß eine Datenbank über mehrere tausend Rechner verteilt sein kann. Die Herausforderung besteht in diesem Zusammenhang darin, **Knotenzugänge** wie auch **Knotenausfälle** zu kompensieren, ohne daß der laufende Betrieb beeinträchtigt wird. □

Funktionsweise Implementierung von **Consistent Hashing Funktionen** siehe [14]

- Der Wertebereich der Zielmenge wird nicht als Liste, sondern als Ring verstanden. Es werden beide Enden zusammengefügt
- Die **Server** 1 - n werden gemäß des Hashwertes ihrer **ID's** (**Servernamen**, **IP-Adressen**, ...) auf diesem Adressring angeordnet.
- Die Datensätze werden gemäß ihres Hashwertes auf dem Adressring angeordnet. Die Datensätze werden auf den **Servern** gespeichert, die im Uhrzeigersinn auf dem Adressring nachfolgend angeordnet sind.
- Wird ein neuer **Server** hinzugefügt, so wird er gemäß seines Hashwertes auf dem Ring angeordnet.

  Die Datensätze, deren Adressen nun im Uhrzeigersinn vor dem neuen Server liegen, werden vom ursprünglichen Server auf den neuen Server umgespeichert. Alle anderen Datensätze können an der ihnen zugewiesenen Adresse bleiben.

- Wird ein **Server** herausgenommen, dann werden seine Daten auf den ihm Uhrzeigersinn nachfolgenden Server kopiert. Alle anderen Datensätze sind nicht betroffen von dieser Änderung.
- Verfügt ein Server über mehr Kapazität, so können für ihn entsprechend viele **virtuelle Server** erzeugt werden.

  Diese **Virtuellen Server** unterscheiden sich in der ID und damit werden für jeden dieser **virtuellen Server** andere Hashwerte ermittelt und damit auch unterschiedlich auf dem Adressring plaziert.

  Auf den realen Server werden dann alle Daten kopiert, deren Adressen von den zugehörigen **virtuellen Servern** auf dem Ring angeordnet sind. □

Einsatz **Consistent Hashing** dient als wichtige Grundlage für im **NoSQL** Bereich eingesetzten **Datenmodelle**. **Consistent Hashing** wird eingesetzt um **Partition Tolerance** und **lineare Skalierbarkeit** implementieren zu können. □

MVCC - Multiversion Concurrency Control

Das **Multiversion Concurrency Control Verfahren** stellt sicher, daß konkurrierende Zugriffe auf Datensätze durch die Verwaltung verschiedener, unveränderlicher Versionen der Datensätze kontrolliert werden.

Für jeden datenändernden Zugriff

- **Einfügen**
- **Ändern**
- **Löschen**

wird für den Datensatz eine neue Version erstellt. Die verschiedenen Versionen werden in einer zeitlichen Reihenfolge gereiht als Liste von Versionen. □

Funktionsweise **MVCC** stellt ein wichtiges Konzept für **NoSQL** Systemen dar, das es ermöglicht, konkurrierende Zugriffe auch ohne das Sperren von Datensätzen zu koordinieren.

- **Lesen:** Das Lesen ist vom Schreiben völlig entkoppelt. Zu jedem Zeitpunkt steht eine aktuelle Version zum Lesen bereit.
- **Schreiben:** Jeder Prozess, der die aktuelle Version ändert, erstellt eine neue Version dieses Datensatzes mit einem Verweis auf die gelesene aktuelle Version.

  Beim Transaktionsende wird die Vorgänger-Versionsnummer des aktuell in dieser Transaktion geänderten Datensatzes mit seiner aktuellen Versionsnummer verglichen.

- **Aufräumen:** In periodischen Zeitabständen müssen alte Versionen gelöscht werden. Ein Datensatz wird als alt eingestuft, wenn er von keiner Transaktion mehr verwendet wird. □

Einsatz MVCC als Technologie kommt in den meisten **NoSQL Datenmodellen** zum Einsatz. Gleichzeitig greifen auch einige **RDBMS** Systeme auf diese Technologie zurück. □

Map/Reduce

Map/Reduce ist weniger ein **Technologieparadigma** als ein Algorithmus zur verteilten Datenverarbeitung. Map/Reduce wird in 2 Hauptphasen durchgeführt:

- **Mapping Phase**
- **Reduction Phase**

Algorithmus Map/Reduce ist ein verteilter Algorithmus, der darauf ausgelegt ist, mit der Zahl der verfügbaren Ressourcen linear zu skalieren. Voraussetzung für einen möglichen Einsatz des Verfahrens ist daß Abhängigkeiten innerhalb der Eingabedaten nicht hinderlich sind für eine Aufteilung der Daten in Blöcke beliebiger Größe. Die Zahl der Blöcke skaliert in bijektiver Weise mit der verfügbaren Ressourcen.

In jeder der Phasen werden die Daten einem bestimmten Ablauf folgend verarbeitet.

- **Partitioning:** Die Eingabedaten in Form einer List als Key/Value Paare vorliegen. Die Daten werden nun sinnvoll über die zur Verfügung stehenden Rechner verteilt.
- **Mapping Phase:** In der Mapping Phase leitet der Map/Reduce Algorithmus die Eingabedaten an den Mapper weiter.

  Die Aufgabe des Mappers ist es die Daten zu filtern und in einen sinnvoles Format zu transformieren. Der Vorgang der Transformation muß in sich abgeschlossen sein und darf für jedes Key/Value Paar nur vom Ausgangszustand des gegebenen Paars abhängig sein.

  Die Ausgabedaten der Map Phase dienen als Eingabe für die 2te Phase des Algorithmus. Die Daten liegen weiter in Form einer List als Key/Value Paare vor.
- **Shuffling:** Während des Shufflings der transformierten Daten, werden jene Daten, die denselben Key der Vorgabe der Applikation entsprechend zusammengefügt und anschließend an den Reducer weitergeleitet.

- **Reduce Phase:** Die Aufgabe des Reducers ist es, auf der gesamten Datenmenge die finalen Schritte auszuführen. □

Quellen

- siehe [7]
- siehe [13]
- siehe [14]

NoSQL Datenmodelle

Konkrete Implementierungen von **NoSQL Datenbanken** vereinen in der Regel mehrere mögliche Datenmodelle. Die Datenmodelle und die ihnen zugrundliegenden Eigenschafen sind darauf abgestimmt, als Einheit diverser Technologien zu arbeiten.

Key/Value Datenmodell

Key/Value Datenmodelle bedienen sich eines einfachen **Key/Value** Systems. Ein Key ist immer einem bestimmten Schlüssel zugeordnet, der aus einer **strukturierten** oder **willkürlichen** Zeichenkette bestehen kann.

Eigenschaften Das Key-Value Datenmodell hat eine Reihe erfreulicher Eigenschaften, die es von anderen Datenmodellen abhebt:

- **linear skalierbar**
- **partitionierbar**
- **hoch verfügbar**
- **kurze Antwortzeiten**

Die hohe **Verfügbarkeit** der Daten wird durch ein **Konsistenzmodell** erkauft das nur die **BASE** Kriterien erfüllt. Einschräkend kommt hinzu daß nur einfache Datenstrukturen in einem Key/Value Datenmodell abgebildet werden können. □

Technologien

- **Consistent Hashing**
- **MVCC** □

Spaltenorientiertes Datenmodell

Der Ansatz, Daten nicht in **Zeilen-** (**relationale Datenbanken**) sondern in **Spaltenform** zu speichern, ist ein fundamentales Konzept im **NoSQL** Bereich.

Struktur Ein spaltenorientiertes Design der Datenstruktur ermöglicht, daß zusammengehörende Daten in einem **disk block** abgelegt werden können. Ein Eintrag besteht dabei aus:

- **Name der Spalte**
- **Daten**
- **Zeitstempel.** □

Column Family Spalten mit **ähnlichen** oder **verwandten** Inhalten bilden dabei die sogenannte Column Family. In ihrem Aufbau entspricht die Column Family einer Tabelle im **relationalen Modell**. Im Gegensatz zur Tabelle hat die Column Family keine strikte Struktur. Sie kann aus Tausenden oder sogar Millionen von Spalten bestehen. □

Zelle Die kleinste Datenverarbeitungseinheit ist eine Zelle. Spaltenorientierte Datenmodelle mit Versioning sind in Lage, mehrere Versionen desselben Datensatzes in einer Zelle zu verwalten. □

Eigenschaften Durch die Speicherung der Daten in Spaltenform erkaufen wir uns einige willkommene Eigenschaften, die im **relationalen** Modell nicht gegeben sind:

- Im Zuge des Leseprozesses wird nur die gewünschte Information gelesen
- Der Schreibprozess besteht in der Regel aus einem einzelnen **disk seek**

Einzig im Fall daß eine Vielzahl von Daten zu schreiben ist, die über mehrere Spalten verstreut sind, bemerken wir eine Verlangsamung im Schreibprozess. □

Dokumentorientiertes Datenmodell

Das Dokumentorientierte Datenmodell stellt ein fundamentales Konzept im Bereich der **NoSQL** Datenbanken dar.

Dokument Im Bereich der Dokumentorientierten Datenbanken werden Daten als Dokumente abgelegt. Ein Dokument versteht sich in diesem Sinne als eine strukturierte Zusammenstellung der Daten. Der Begriff des Dokuments sollte in diesem Zusammenhang nicht verwechselt werden mit den von Textverarbeitungsprogrammen erzeugten Dateien. □

Datenstruktur Die Daten in einem Dokument sind in Form von Key/Value-Paaren strukturiert. Das dokumentorientierte Datenmodell genügt damit keiner Normalform, eine spezifische Strukturierung der zu speichernden Daten ist nicht vorgeschrieben. In jedem einzelnen Dokument kann eine andere Struktur zu finden sein, ein allumfassendes, die ganze Datenbank umspannendes **Datenbankschema** existiert nicht.

Das Datenmodell wird als **Schema frei** bezeichnet. Die Verwendung von **Constraints** oder **Triggern** ist im Datenbanksystem nicht möglich. Jedes Dokument stellt für sich gesehen eine geschlossene Einheit mit eigenem Schema dar. Das Datenmodell selbst unterstützt von Haus aus keine Abfragesprache im Sinne von **SQL**. □

Eigenschaften Die Stärke dokumentenbasierter Datenbanken ist es, zusammengehörige Daten als Einheit zu speichern. Im Gegensatz zu **relationalen DBMS**, die mit **SQL** über einen einheitlichen Standard verfügen, unterscheiden sich die Implementierungen dokumentenbasierter Datenbanken mitunter erheblich in ihrem Funktionsumfang. □

NoSQL Implementierungen

Wir wollen einige Vertreter von **NoSQL Datenbanksystemen** näher analysieren, um ihren Wert für einen Einsatz im **NoSQL Prototypen** abzuschätzen:

- **Cassandra**
- **CouchDB**
- **Hadoop-Hive**
- **MongoDB**
- **AppEngine**

Cassandra

Beschreibung Cassandra wurde mit dem Augenmerk entwickelt, große Datenmengen, die über mehrere Netzwerkknoten verteilt sind, zu verwalten.

Die Datenbankarchitketur wurde von vornherein darauf ausgelegt, Knotenausfälle zu kompensieren. Die Daten sind als **Replikate** auf mehreren Knoten vorhanden. Damit wird **Reliability** und **Fault Tolerance** für das System etabliert. □

Cassandra Cluster Eine Cassandra Instanz ist eine Sammlung unabhängiger Knoten, die in Clustern zu einer Einheit gebündelt werden.

Alle **Knoten** in einem Cassandra Cluster sind **Peers**, kein Knoten besitzt mehr **Rechte** oder **Aufgaben**. In einem Cassandra Cluster findet sich damit kein **Single point of failure**. □

Eigenschaften

- Die Daten werden über alle Knoten im Netz verteilt.
- Knoten kommunizieren über ein **Peer to Peer gossip** Protokoll untereinander.
- Cassandra basiert auf einem **spaltenorientierten Datenmodell**. Als Adressierungsmethode wird **Consistent Hashing** eingesetzt.
- Alle Schreibzugriffe auf die Daten werden automatisch im gesamten Cluster repliziert. □

CQL Als Datenmanipulationssprache wird **CQL** eingesetzt. Zur Vereinfachung der Arbeit mit **Cassandra** verwendet **CQL** eine ähnliche Syntax wie **SQL**.

Für jede Anfrage kann in Cassandra der Grad der gewünschten **Konsistenz** angegeben werden. Eine Anfrage, die mehr Datenkonsistenz verlangt, ist auch langsamer in der Ausührung. □

Konsitenzmodell:

- **Stufen der Schreibkonsistenz:**
  - **any:** Gewährleistet, daß die Daten in mindestens einem Knoten gespeichert sind.
  - **one:** Gewährleistet, daß die Daten in den **Commit-Log** von mindestens einem **Replica** gespeichert sind.
  - **quorum:** Gewährleistet, daß die Daten in einem **Quorum** von **Replicas** gespeichert sind.
  - **all:** Die Daten müssen auf allen **Replicas** gespeichert werden.
- **Stufen der Lesekonsistenz:**
  - **any:** Nicht verfügbar.
  - **one:** Liefert die Daten vom naheliegensten Knoten zurück
  - **quorum:** Liefert die neuesten Daten auf Basis der Zeitstempel durch eine Ausführung eines **Quorum Read**
  - **all:** Liefert die neuesten Daten auf Basis der Zeitstempel aus allen **Replicas** zurück. □

Quellen:

- Architecture Overview:
  `http://wiki.apache.org/cassandra/ArchitectureOverview`
- Internal Architecture:
  `http://wiki.apache.org/cassandra/ArchitectureInternals`
- Data Model:
  `http://wiki.apache.org/cassandra/DataModel`
- CQL:
  `http://www.datastax.com/0.8/api/cql.ref`

CouchDB

CouchDB ist ein Akronym und steht für **Cluster of unreliable commodity hardware Database**.

Datenmodell Die Speicherung der Daten erfolgt im **JSON** Format. CouchDB baut auf einem dokumentorientierten Datenmodell auf. Dokumente werden in CouchDB bei einer Änderung nicht überschrieben, sondern es wird ein neues Dokument mit der selben ID generiert. Dokumente werden zur effizienteren Verarbeitung in **B-Bäumen** gespeichert. □

Kommunikationsmodell Die Kommunikation mit CouchDB erfolgt über **HTTP**. Als Schnittstelle wird **REST** eingesetzt. Da es sich bei **HTTP** um ein zustandsloses Protokoll handelt, müssen alle relevanten Transaktionsdaten in einer Anfrage übergeben werden. Innerhalb einer Anfrage erfüllt CouchDB die **ACID** Kriterien. □

Konsistenzmodell **CouchDB** wurde konzipiert, um eine hohe **Datenverfügbarkeit** aufzuweisen. Das wird dadurch erreicht, daß die Daten auch während **Replikationsvorgängen** weier lesbar bleiben. Um ein **Locking** der Daten zu verhindern setzt CouchDB auf MVCC. Durch diese Form der Konfliktauflösung gehen keine Daten verloren, jedoch kann nicht sichergestellt werden, daß die Daten in der aktuellen Version vorliegen. □

Replikation CouchDB ist als verteilte Datenbank konzipiert. Um die Konsistenz der Daten sicherzustellen, setzt CouchDB auf **Replikation**.

Die **Replikation** der Daten findet als inkrementeller Prozess statt. Die einzelnen Knoten kommunizieren untereinander ebenfalls über die **REST** Schnittstelle. Hinzu kommt, daß das dokumentenweise Übertragen der Daten gegenüber Hardwareausfällen sicher ist. Die Architektur der Datenbank macht sie generell gegenüber Knotenausfällen robust. □

Datenmanipulation Als Abfragesprache wird in CouchDB auf in JavaScript programmierte Map/Reduce Funktionen gesetzt. □

Quellen:

- Apache Projekt:
  `http://couchdb.apache.org/`
- CouchDB Wiki:
  `http://wiki.apache.org/couchdb/FrontPage`

Hadoop-Hive

Hadoop Hadoop ist ein freies Java basiertes quelloffenes Framework, das unter der Apache Software Lizenz entwickelt wird. Hadoop ist eine Implementierung des Map/Reduce Algorithmus. Die Stärke von Hadoop besteht darin große Datenmengen in relativ geringer Zeit verarbeiten zu können.

Im Kern besteht Hadoop aus folgenden Komponenten:

- **Hadoop Distributed File System:** Ist das verteilte Dateisystem von Hadoop. Es zeichnet sich durch hohe Ausfallsicherheit aus. Mit Hilfe von **HDFS** werden die Daten auf die verschiedenen Knoten des **Clusters** erteilt.
- **Map/Reduce Framework**: Wird zur Datenverarbeitung eingesetzt. □

Hive Hive ist eine **Data Warehouse Infrastruktur** die auf Hadoop aufsetzt.

Hive wird eingesetzt, um **Analysen** und **Anfragen** auf großen Datenbeständen auszuführen. Hive besitzt mit **HQL** eine **SQL** ähnliche Abfragesprache zur Interaktion mit dem Anwender.

Hive setzt in erster Line auf

- **Skalierbarkeit:** Hinzufügen von Knoten zum Hadoop Cluster
- **Erweiterbarkeit:** Hinzufügen von eigenen Map/Reduce Skripten
- **Ausfallstoleranz**

Updates, **Transaktionen** und **Indizes** werden von Hive nicht unterstützt. Hive greift zur Datenverarbeitung in erster Line auf Map/Reduce Skripten zurück. Updates werden über das Anlegen neuer Tabellen im Kontext der Ausführung von Skripten verwirklicht.

In traditionellen Datenbanksystemen wird beim Einspielen von Daten ein Schema erzwungen. Passen die Daten nicht zum Schema, werden sie verworfen. Dieser Ansatz wird auch **schema on write** genannt da die Daten beim Schreiben gegen das Schema geprüft werden. Hive hingegen arbeitet nach dem **schema on read** Ansatz. Dieser sieht vor, daß die Daten nicht schon beim Laden, sondern erst bei Anfragen gegen das Schema geprüft werden. □

Features	SQL	Hive
Updates	update insert delete	insert, overwrite, keine updates
Transktionen	ja	nein
Indizes	ja	nein
Latenz	Millisekunden	Minuten
Multi-inserts	nein	ja
select	sql 92	nur 1 Tabelle in der from Klausel
join	sql 92	inner, outer, semi und map joins
Unterabfragen	sql 92	nur in der from klausel

Quellen:

- Apache Projekt:
  `http://hive.apache.org/docs/r0.9.0/index.html`

MongoDB

MongoDB ist ein weiterer bekannter Vertreter für NoSQl Datenbanken. Namensgebend für MongoDB ist ein Wortspiel mit dem englischen **humongous**.

Eigenschaften

- Der MongoDB Server verwaltet einen Netzwerk von mehreren Datenbankinstanzen. Jeder Knoten kann einzeln **behandelt** und **konfiguriert** werden.
- **skalierbar**
- **hochperformant**
- **schemafrei**
- **dokumentorientiert:** BSON
- **integrierte Querylanguage**
- MongoDB skaliert horizontal. Erreicht wird das durch den Einsatz von **Replikation** und **Sharding**.
- Schwächen zeigt die Datenbank in der Sicherheit.
- Ebenfalls sind **Queries** und **Transaktionen** im Vergleich zu RDBMS nur verschlankt einsetzbar.
- Map/Reduce wird eingesetzt um Funktionen auf den Daten umzusetzten. Map/Reduce entspricht dem Group Operator. □

Quellen

- Projekt:
  `http://http://www.mongodb.org`
- BSON Spezification
  `http://www.bsonspec.org`

AppEngine

Beschreibung Die AppEngine setzt auf BigData zur

- **Verwaltung seiner Daten**
- **Verwaltung seiner Struktur**
- **Sicherstellung der Datenkonsistenz** □

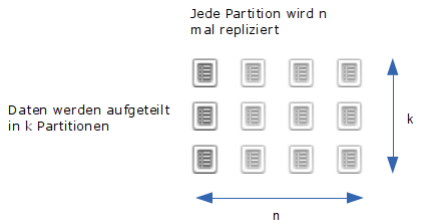
BigTable BigTable ist ein verteiltes Datenverarbeitungssystem das vom Konzept her als eine Tabelle ausgelegt ist, eine Tabelle die Petabytes an Daten speichert und über tausende Netzwerkknoten verteilt sein kann. BigTabel ist in der Lage, tausende von Anfragen in einer Sekunde zu verarbeiten.

BigTable nimmt im **NoSQL** Bereich eine fundamentale Vorreiterrolle ein. Viele der entscheidenden Konzepte im **NoSQL** Bereich wurden für BigTable pioniert. □

Eigenschaften von BigTabel

- **Fundamentally Distributed:** Big Table wurde von Beginn an als verteilte, robuste Architektur entworfen. Die zu speichernden Daten sind verteilt auf auf eine Menge von Netzwerkknoten. Die Architektur ist so ausgelegt daß Knotenausfälle durch **Replikation** kompensiert werden können.

  Data partitioning wird eingesetzt um die Daten ihrem Schlüssel entsprechend auf verschieden Knoten zu verteilen. **Replikation** sorgt auf der anderen Seite dafür daß Kopien von Daten auf mehreren Knoten verteilt vorliegen.



- **Column Oriented:** BigTable implementiert ein spaltenorientiertes Datenmodell.
- **Sequential write:** BigTable ist optimiert für Schreiboperationen.
- **Chubby: lock/file/name service**. Chubby realisiert einen Lockingmechanismus für Bigtable. Chubby wird in Bigtable für mehrere Aufgaben eingesetzt:
  - Sicherstellung daß es zur Ausführungszeit nur einen aktiven Master gibt
  - Speicherung von BigTable Schema Informationen
- **Map/Reduce**
- **MVCC**
- **Consistent Hashing** □

Quellen:

- Projekt:
  `https://developers.google.com/appengine/`
- AppEngine Docs
  `https://developers.google.com/appengine/docs/whatisgoogleappengine`

CHAPTER 4

RDBMS und andere Datenbanksysteme

Relationale Datenbanksysteme

Beschreibung **Relationale Datenbanksysteme** wurden entwickelt um den wachsenden Ansprüchen einer neuen Generation von Applikationen gerecht zu werden.

- **Redundanz und Inkonsistenz** von Daten
- **beschränkte Zugriffsmöglichkeiten**
- **Herausforderungen des Mehrbenutzersbetriebs**
- **Verlust von Daten**
- **Integritätsverletzung**
- **Sicherheitsprobleme** □

Relationale Datenbanken: Prinzipien

Relationales Datenmodell: Anfang der siebziger Jahre wurde das **relationale Datenmodell** konzipiert. Die Besonderheit dieses Datenmodells besteht in der mengenorientierten Verarbeitung der Daten im Gegensatz zu den bis dahin vorherrschenden Datenmodellen.

Das **relationale Datenmodell** ist im Vergleich zu anderen Modellen sehr einfach strukturiert. Es gibt im wesentlichen nur flache Tabellen (**Relationen**), in denen die Zeilen den Datenobjekten entsprechen. In dieser einfachen Struktur liegt vermutlich der Erfolg der relationalen Datenbanken. Die in den Tabellen gespeicherten Daten werden durch entsprechende Operatoren ausschließlich mengenorientiert **verknüpft** und **verarbeitet**.

Das **Entity-Relationship** Modell wird verwendet, um die Struktur der Datenbank und die Relationen der Tabellen untereinander zu beschreiben. Das **EER** besitzt zwei grundlegende Strukturierungskonzepte:

- **Entitytypen:** Entsprechen den zu modellierenden Daten
- **Beziehungstypen:** Werden eingesetzt, um die Beziehungen zwischen den Daten abzubilden.

Der große Vorteil des **Relationalen Datenmodells** liegt in seiner Eigenschaft, komplexe Beziehungen zwischen Daten einfach modellieren zu können. Unter Zuhilfenahme verschiedener **Constraints** kann die logische Konsistenz der Daten stets gewährleistet werden. □

Relationale Anfragesprache: SQL Die **relationale Algebra** und der **Relationenkalkül** bilden die theoretischen Grundlagen für die Anfragesprache **SQL**.

Anfragesprachen wie **SQL** sind im Allgemeinen deklarativ. Die Benutzer geben lediglich an, welche Daten sie interessieren, und nicht, wie die Auswertung der Daten vorgenommen wird. Die oft sehr komplexen, zur Festlegung der Auswertung nötigen Entscheidungen werden vom Anfrageoptimierer des Datenbanksystems übernommen.

Zusätzlich zur Manipulation von Tabellen beinhalten Anfragesprachen auch Möglichkeiten zur Definition von **Integritätsbedienungen** für die Daten, zur Vergabe von **Zugriffsrechten** und zur **Transaktionskontrolle** □

Elemente von SQL

- **Datentypen**: **Relationale Datenbanken** zeichnen die zu speichernden Daten aus, indem sie ihnen Datentypen zuordnen
- **Schemadefinition:** Mit dem Wissen der Datentypen können die Daten in Tabellen abgebildet werden. Die zu einer Datenbank gehörenden Tabellendefinitionen werden als **Schema** bezeichnet.
- **SQL-Anfragen:** Die **SQL Anfragen** stellen eine mengentheoretische Auswertung des Datenbestandes dar. Der Grad der Komplexität der Anfrage wird bestimmt durch die Anfragestellung:
  - Anfragen über mehrere Relationen hinweg

  - Aggregatfunktionen und Gruppierungen
  - Geschachtelte Anfragen
  - Quantifizierte Anfragen
  - Sichten □

Transaktionen Unter einer **Transaktion** versteht man die Bündelung mehrerer Datenbankoperationen, die in einem verteilten System ohne unerwünschte Einflüsse durch andere Transaktionen als Einheit fehlerfrei ausgeführt werden sollen.

Eine Transaktion wird definiert durch folgende Eigenschaften:

- **Atomicity:** Abgeschlossenheit - Jede Operation in einer Transaktion ist vollständig abgeschlossen (oder es gibt gar keine)
- **Consistency:** Konsistenz - Bei Transaktionsstart und -ende ist die Datenbank in einem konsistenten Zustand.
- **Isolation:** Abgrenzung - Die Transaktion verhält sich so, als wäre sie die einzige Operation die auf der Datenbank ausgeführt wird
- **Durability:** Dauerhaftigkeit - Ist eine Transaktion einmal abgeschlossen, wird sie nicht mehr rückgängig gemacht. □

Konsistenzmodell: Eine der herausragenden Eigenschaften von **relationalen Datenbanken** ist die stets gegebene Konsistenz der Daten. Ist eine performantere Verarbeitung der Daten gewünscht, besteht die Möglichkeit, die durch das **Konsistenzmodell** gegebenen Einschränkungen herabzustufen. Eine Schlüsseltechnologie hier sind die Transaktionen. □

verteilte Datenbanken: Grundprinzip verteilter Datenbanken ist die **Verteilungstransparenz**. Ein verteiltes Datenbanksystem muß sich dem Anwender gegenüber genauso verhalten wie ein nicht verteiltes.

Eine verteilte Datenbank ist ein logisch zusammengehörender Datenbestand, der physisch auf mehrere Knoten verteilt ist und durch das verteilte Datenbanksystem administriert wird. □

Relationale Datenbanken: Implementierungen

MySQL Features:

- **Thread Pooling**
- **Multicore Scalability**

- MySQL Cluster
- Semisynchronous Replication
- Partitioning □

11 G Features:

- Database Replay
- Partitioning
- Schema Management
- Data Warehousing and OLAP
- PL/SQL: Efficient Coding
- Security
- Oracle Exadata Simulator
- SQL Plan Management
- Manageability
- SQL Access Advisor
- SQL Operations: Pivot and Unpivot
- Backup and Recovery
- Resiliency
- Automatic Storage Management
- Compression
- Caching and Pooling
- Data Guard

CHAPTER 5

Implementierung der Prototypen

Einführung

Übersicht Im Zuge der Arbeit haben wir einen Katalog von Anforderungen erarbeitet, der die Rahmenbedienungen zur Gestaltung der **Architektur** und technischen **Umsetzung** einer bestimmten Klasse von Applikationen definiert. Wir haben die unterschiedlichen **Konzepte** und **Paradigmen** kennengelernt, die für uns relevant sind, um das Problem zu verstehen und adäquat zu interpretieren.

Zur Etablierung unserer Ergebnisse werden 2 Protypen erstellt und geprüft. Die Prototypen werden unterschiedlichen Paradigmen folgend entwickelt. Zum Vergleich werden die Prototypen identischen Lastentests unterzogen. Im Laufe dieses Kapitels wollen wir die unterschiedlichen Technologien vergleichen, die in die Gestaltung der Prototypen eingeflossen sind. □

RDBMS vs NoSQL Beide Prototypen implementieren das MVC Modell. Die Herausforderung in der Gestaltung beider Programme ist es, sowohl das **View** als auch den **Controller** für beide identisch zu halten. Lediglich in der Model Schicht streben wir eine grundlegende Differenzierung im Sinne der Implementierung an.

Die Prototypen weisen eine grundlegende Differenzierung in ihrem **Daten-**, **Konsistenz-** und **Replikationsmodell** auf. Ein Prototyp wird den Prinzipien der NoSQL Ansätze folgend gestaltet, der andere nutzt die Theorie der verteilten relationalen Datenbanken für die Datenverarbeitung. □

Programmiersprache

Verteilte Systeme Die **grundlegende Herausforderung**, aber auch **herausragende Möglichkeit** für heutige Softwareentwickler besteht darin verteilte Systeme zu **verstehen** und zu **entwickeln**. Zeitgerechte Anwendungen, im kleinen aber auch im industriellen Bereich, werden mit hoher Wahrscheinlichkeit in einer Art von Netzwerk zur Anwendung kommen. Moderne **Programmiersprachen** werden im Licht dieser Anforderungen entwickelt. □

Programmiersprachen - Funktionalität Viele Programmiersprachen integrieren bereits auf Sprachebene grundlegende Prinzipien zur Unterstüzung der Entwicklung **verteilter** und **paralleler** Software. Zusätzliche Funktionalität wird den Entwicklern mit Frameworks zur Verfügung gestellt.

Der Fortschritt und die Vielfalt webbasierter **Client-Server** Frameworks erlauben uns aus einem Pool von Technologien zu wählen. Der Großteil dieser Technologien etabliert **HTML Clients** mit Unterstützung für diverse Scriptsprachen, während am Server die zentrale Datenverarbeitung stattfindet.

Zur Realisierung unserer Prtotypen wollen wir uns auf die Wahl einer Programmiersprache einschränken. Die Programmiersprache wird eingesetzt, um die erarbeitete Architektur umzusetzen.

Im Zuge dieses Abschnitts werden wir einige gängige Technologien **vorstellen** und **vergleichen**. □

Java

Java ist eine objektorientierte Programmiersprache. Die Programmiersprache ist ein Bestandteil der Java Technologie. Die Java Technologie ist als Komposition zu verstehen aus einer **virtuellen Maschine**, **Programmiersprache**, **Werkzeugen** und **Bibliotheken**.

Java Technologie (Quelle: [21]) Java setzt sich zusammen aus:

- **JDK:** Java-Entwicklungswerkzeuge zum Erstellen von Java Programmen
- **JRE:** Java-Laufzeitumgebung zu deren Ausführung.
- **JVM:** Laufzeitumgebung besteht selbst aus der **virtuellen Maschine** sowie den mitgelieferten **Bibliotheken** der **Java Laufzeitumgebung**. □

Programmiersprache Java Die Programmiersprache Java dient innerhalb der Java Technologie vor allem zum Formulieren von Programmen. Java Programme werden in 2 Schritten erstellt. Zunächst wird der **Quellcode** erstellt und im Anschluß daran der **Programmcode** generiert.

- **Quellcode:** Dieser liegt zunächst als reiner, menschenverständlicher Text vor, als sogenannter **Quellcode**. Der Quellcode ist nicht direkt ausführbar
- **Bytecode:** der Java-Compiler übersetzt den Quellcode in den **Java-Bytecode**. Die JVM Maschine führt diesen Code aus.

Zweck dieser **Virtualisierung** ist **Plattformunabhängigkeit**. Das Programm soll ohne weitere Änderung auf jeder Rechnerarchitektur laufen können, wenn dort eine passende Laufzeitumgebung installiert ist. □

Eigenschaften von Java

- **Einfachheit**
- **Objektorientierung**
- **Verteiltheit**
- **Robustheit**
- **Sicherheit**
- **Architekturneutralität**
- **Portabilität**
- **Parallelisierbarkeit**
- **Dynamisch** □

Verteilte Systeme Java Java wurde mit der Prämisse entwickelt eine Sprache zur Programmierung verteilter Systeme zu schaffen. Viele grundlegende Konzepte zur Programmierung verteilter Systeme sind direkt in die Sprache eingebetet:

- **Multithreading**
- **Sockets, ServerSockets, URL**
- Unterstützung unterschiedlicher **Kommunikationsprotokolle (TCP ,UDP)**
- **Sicherheitsfeatures** □

Frameworks Für Java steht eine immense Zahl an verschiedenen Frameworks zur Verfügung.

- **JSP**
- **JSF**
- **Grails**
- **EJB**
- **Apache Tapestry**
- **GWT** □

Einsatzbereich Das Einsatzgebiet der Java Plattform findet sich in erster Linie im Enterprisebereich. Java erfährt eine kontinuierliche **Weiterentwicklung** und **Evolution**. Die meisten namhaften Unternehmen setzen Java für die Entwicklung ihre Anwendungen ein.

- **Twitter**
- **Fotolog**
- **Amazon**
- **eBay**
- **Flicker**
- **Amazon Dynamo**
- **FeedBurner**
- **Googlespeak**
- **Mailinator** □

PHP

Beschreibung PHP ist eine Skriptsprache mit einer an **C** und **Perl** angelehnten Syntax, die hauptsächlich zur Erstellung dynamischer Webseiten oder Webanwendungen eingesetzt wird. PHP wird als freie Software unter der PHP-Lizenz verbreitet.

PHP zeichnet sich durch die breite **Datenbankunterstützung** und **Internet-Protokolleinbindung** sowie die Verfügbarkeit zahlreicher Funktionsbibliotheken aus. PHP wird auf etwa 244 Millionen Webseiten eingesetzt (Stand: Januar 2013).

PHP ist ein System, das PHP-Code serverseitig verarbeitet. Das bedeutet, dass der Quelltext nicht an den Webbrowser übermittelt wird, sondern an einen Interpreter auf dem Webserver. Erst die Ausgabe des PHP-Interpreters wird an den Browser geschickt. In den meisten Fällen ist das ein HTML-Dokument, wobei es mit PHP aber auch möglich ist, andere Dateitypen wie Bilder oder PDF-Dateien, zu generieren. □

Webframeworks

- **Cappuccino**
- **Seagull**
- **Symfony**
- **CakePHP** □

ASP.NET

Das von Microsoft entwickelte .NET Framework ist ein bekannter JAVA Klone. Die für Java etablierten Ergebnisse sind in bijektiver Weise auch für .NET gültig.

Wahl der Programmiersprache: Java

Begründung Java als Werkzeug ist weder besser noch schlechter geeignet zur Umsetzung skalierender Architekturen für **verteilte Applikationen im Datenverarbeitungsbereich** als andere Programmiersprachen. Java skaliert genauso wie jede andere Programmiersprache.

Was Java auszeichnet ist:

- **Java als Platform**: Ein Ecosystem bestehend aus **JVM**, **Bibliotheken** und **Werkzeugen**.
- **Community**: Die Evolution von **Java** wird von einer zahlreichen und vielfältigen **Community** vorangetrieben.
- **Cloud Computing Standard:** **Oracle** arbeitet als eines der ersten Unternehemen an der Etablierung eines **Standards** für eine einheitliche **Cloud Computing API**. Mit Java 8 soll diese teil des JDK werden. □

skalierende Infrastruktur Die Java Programmiersprache hat keinen Einfluß, ob ein Programm skaliert oder nicht. Java als Plattform dient zur Entwicklung neuer **skalierender Technologien**. Der Architekturansatz, den Großteil der Arbeitslast auf einen einzelnen **monoliten Server** auszulagern, wird zu Gunsten anderer komplexerer Technologien wie **jGroups**, **ehcache** und **terracotta** aufgegeben. Java als Ecosystem versuchte seit seinem Betstehen die Entwicklung dieser Art von Software zu unterstützen. □

Wahl des Frameworks: GWT-GAE

Anforderungen Die Biber Webapplikation ist als **Client-Server** System konzipiert. Wir werden versuchen, den gesammelten Anforderungen der Spezifikation mit der Wahl einer entsprechenden Technologie zu entsprechen. Die auszuwählende Technologie muß die Kriterien für ein **verteiltes System** erfüllen:

- **Hohe Verfügbarkeit**
- **Ortstransparenz**
- **Replikationstransparenz**
- **verteilte Anfragebearbeitung**
- **verteilte Transparenzbearbeitung**
- **Hardwareunabhängigkeit**
- **Betriebssystemunabhängigkeit**
- **Leistungstransparenz**

Gleichzeitig muß die Applikation als Anwendung für das World Wide Web konzipiert sein. Alle Webframeworks scheinen in gleicher Weise für den Einsatz in unserem Prototypen geeignet zu sein. Als Technologien, die auf **HTTP** aufsetzen und eine **Client-Server** Architektur etablieren, werden die geforderten Eigenschaften erfüllt. □

Begründung GWT-GAE ist ein von Google entwickeltes Framework. GWT-GAE ist für Google Cloud optimiert und stellt ein Webframework zur Entwicklung von **Client Server** Architekturen bereit. Alle geforderten Anforderungen können im Gegensatz zu den anderen Frameworks mit diesem Werkzeug umgesetzt werden. Im Vergleich zu anderen Frameworks ist GWT-GAE **ausgereift** und **erprobt**. Zusätzlich stellt uns das **Framework** die Werkzeuge zur Verfügung sowohl **NoSQL** als auch **RDBMS** Systeme für die Datenverarbeitung zu integrieren. □

Vergleich

- **Biber WebAnwendung:** Die primäre Anforderung an die Applikation ist es eine **performante**, **linear skalierende** Architektur aufzubauen. GWT - GAE ist Teil der Google Cloud. Damit ist das Framework geeignet eine solche Architektur zu implementieren.
- **RDBMS vs NoSQL**: GWT - GAE stellt sowohl eine Implementierung von Bigtable zur Verfügung, als auch eine Anbindung an CloudSQL mit MySQL.
- **HTTP - Client/Server:** GWT - GAE baut auf dem **HTTP Protokoll** auf. □

NoSQL: Implementierung des Models

NoSQL Systeme sind als eine Sammlung unterschiedlicher Konzepte und Technologien zu verstehen. Trotz diverser Technologieparadigmen gibt es einheitliche Ziele die jedes dieser Systeme umzusetzen versucht.

- **Scale-out:** lineare Skalierung
- **Elasticity:** das System kann dynamisch zur Laufzeit erweitert werden

Paradigmenvergleich

Um die unterschiedlichen **NoSQL** Systeme vergleichen zu können erarbeiten wir uns einen Katalog von Kriterien um die Systeme im Anschluß entsprechend bewerten zu können:

- **Read/Write optimized:** Das dem System zugrundeliegende Technologieparadigma resultiert oft in einem unterschiedlichen Verhalten für Schreib- und Lesezugriffe
- **Sync/async Replication:** **Replikation** wird eingesetzt um in einem verteilten System Datenverlust zu vermeiden, die Geschwindigkeit zu verbessern und für Availability zu sorgen.

  Wir unterscheiden zwei Formen der Replikation:

  - **synchrone Replikation:** synchrone Replikationen stellt sicher daß die Daten in allen Knoten kosistent sind. Bei diesem Replikationsansatz nimmt man gleichzeitig hohe Latenzzeiten bei Datenupdates in Kauf.
  - **asynchrone Replikation:** Asynchrone Replikation hat keinen negativen Auswirkungen auf die Latenz des Systems, es kann aber vorkommen, daß in manchen Knoten alte Daten gespeichert werden.

Systemvergleich

Das **CAP Theorem** (Kapitel 1, Seite.11) hilft uns eine **Klassifizierung** für **verteilte Systeme** im **Datenverarbeitungsbereich** zu etablieren. Im **NoSQL** Bereich beschränkt sich unsere Diskussion damit auf die CP- und AP Systeme. Die folgende **Systeme** sollen für einen Vergleich näher betrachtet werden:

- **MongoDB**
- **HBase**
- **BigTable**

MongoDB vs BigTable: Vergleich

- **Replikation:** Beide unterstützen **synchrone Replikation**
- **Read/Write Optimized:** Beide Systeme sind **Schreib/Lese** optimiert
- **Datenchema:** Beide DB Systeme setzen auf ein **schema-less** Datenschema
- **Abfragen:** Bei beiden Systemen muß bei der Gestaltung von Queries auf ihre Komplexität achtgegeben werden.
- **Skalierbarkeit:** Beide Systeme **skalieren linear**
- **Transaktionen:**
  - **BigTable** unterstützt Transaktionen solange die entsprechenden Daten in Gruppen strukturiert wurden.
  - **MongoDB** unterstützt keine Transaktionen als solche, es gibt aber **atomic Operations** die helfen Transaktionen zu simulieren.

MongoDB vs Bigtable: Resultat Beide DB Systeme bieten ähnliche Features, es muß aber erwähnt werden daß **Googles BigTable** das Ältere der beiden Systeme ist und mit der **Implementierung** von **MongoDB** nicht mithalten kann. MongoDB bleibt unerreicht solange ganze Familien von **Objektgraphen** gespeichert oder geladen werden sollen.

Für den Einsatz von **Googles Appengine** spricht in letzter Instanz jedoch nur Googles unerreichte Harware Konfiguration auf der die Appengine aufsetzt. □

HBase vs BigTabel: Vergleich **Apache HBase** ist eine open-source **NoSQL** Datenverarbeitungssystem modelliert nach **Googles BigTable**. Im Gegensatz zum BigTable, das auf Googles File System aufsetzt, verwendet **HBase Hadoop** als Kern.

Vergleich:

- **Replikation:** HBase unterstützen **synchrone Replikation**
- **Read/Write Optimized:** Der Zugriff auf Daten ist **streng konsistent** jedoch nicht immer sehr performant unter HBase
- **Datenchema:** Beide DB Systeme setzen auf ein **schema-less** Datenschema

- **Abfragen:** Das System ist in der Lage große Datenmengen in kürzester Zeit zu verarbeiten. HBase setzt dazu **Map/Reduce** ein.
- **Skalierbarkeit:** linear skalierbar

HBase vs BigTabel: Resultat Wieder fällt eine große Ähnlichkeit zwischen den beiden System auf. Wieder muß sich **BigTable** gefallen lassen das ältere von beiden Systemen zu sein. **HBase** wird jedoch aufgrund seines Kerns in erster Line zum Verarbeiten und Analysieren von Daten eingesetzt. Im Gegensatz dazu ist **BigTable** optimiert für einfahe Schreib-/Lesezugriffe.

Erneut fällt die Wahl zugunsten von **BigTable** aus da **HBase** in erster Linie zur Datenauswertung und -Verarbeitung eingesetzt wird. □

RDBMS - Implementierung des Models

Beschreibung Verteilte relationale Datenbanksysteme werden seit Jahren eingesetzt und weiterentwickelt. RDBMS Systeme finden Verwendung für kleine Anwendungen, genauso wie in Banken und anderen datenintensiven Umgebungen. □

Resultat Die den **RDBMS** Systemen zugrundeliegende Technologie erfährt nun seit Jahren eine Entwicklung und Verfeinerung. Die Produkte der verschiedenen RDBMS System Anbieter weisen zwar im Detail Unterschiede auf, diese sind jedoch marginaler Natur und fallen kaum ins Gewicht. Für die Implementierung des **RDBMS** Models wählen wir MySQL weil es sich um eine sehr gute open source Datenbank handelt. □

CHAPTER 6

Architekturbeschreibung

Eine Patternsprache

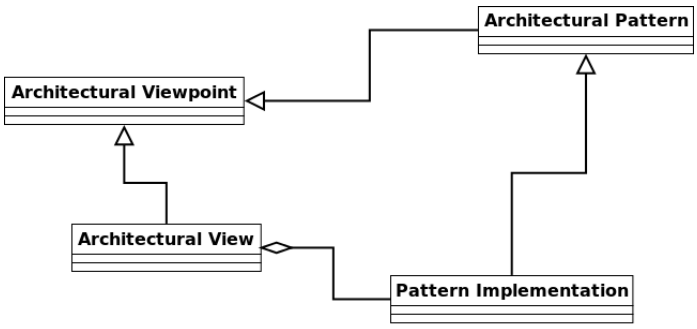
Architectural Patterns Architectural Patterns sind ein grundlegendes Werkzeug im Bereich der Softwareentwicklung. Sie stehen stellvertretend für einen Katalog diverser Alternativen zur Gestaltung umfangreicher Softwareanwendungen. Der Einsatz von Architectural Patterns hilft uns die Zahl der möglichen Architekturansätze auf ein überschaubares Maß einzuschränken. □

Patternsprache Wir wollen uns des Konzeptes der Patternsprache bedienen um eine strukturierte Klassifizierung der Designmöglichkeiten zu schaffen. Eine Patternsprache versteht sich als Supermenge existierender Sammlungen von Architecturalpatterns. Die Klassifizierung der Pattern baut auf dem Konzept der Architectural Views auf.

Beschreibung der Elemente des Konzepts: (siehe [22])

- **Architectural View:** Ist eine Repräsentation des Systems aus der Perspektive unterschiedlicher verwandter Belange. (Ein Belang in einem verteilten Softwaresystem beschreibt die Abbildung der Softwarekomponenten auf Netzwerkknoten)
- **Architectural Viewpoint:** Die Typen der Elemente und deren Beziehungen werden durch die Viewpoints beschrieben. Eine View ist eine Instanz eines Viewpoints für ein bestimmtes System.

Grundlegend für Patternsprachen ist das Konzept der Views. Views geben uns die Möglichkeit Pattern in Perspektive bestimmter Schwerpunkte zu betrachten. □



Viewliste

- **Layered View:** Das System wird dargestellt als ein komplexes System heterogener Komponenten, die sich als autonome interagierende Teile verstehen.
- **Data Flow View:** Das System versteht sich als Folge von Transformationen auf einem Stream von Eingabedaten.
- **Data Centered View:** Das System wird gesehen als ein persistenter, gemeinsam genutzer Datenport, genutzt und verändert von diversen Komponeneten.
- **Adaption View:** Das System wird als Summe unveränderlicher und adaptationsbereiter Teile verstanden.
- **User Interaction View:** Das System wird primär aus der Perspektive des Anwenders dargestellt. Wir unterscheiden zwischen GUI Teilen und Komponenten, die die Applikationslogik implementieren.
- **Component Interaction View:** Das System wird als eine Zahl unabhängiger Komponenten die miteinader zusammenarbeiten verstanden.
- **Distribution View:** Das System wird gesehen als Summe von Komponenten die auf ein Netzwerk verteilt werden. □

Top Level Architektur

Layers Pattern

Architektur Zur Beschreibung der obersten Ebene unserer Architektur wählen wir das **Layers Pattern**. Um eine Vergleichbarkeit der Prototypen zu gewährleisten, soll die Architektur beider Programme dem Prinzip des **Schichtenmodells** folgend gestaltet werden. Gelingt es uns einen Großteil der Schichten in beiden Prototypen identisch zu halten, kann ein legitimer Vergleich der Programme durchgeführt werden.

Das **MVC Modell** ist eine Implementierung des **Schichtenmodells**, eingesetzt zur Strukturierung einer bestimmten Klasse von Anwendungen.

Die einzelnen Schichten des Modells sind

- **Model:** Das Model als Schicht ist für die Datenverarbeitung der Anwendung verantwortlich
- **View:** Das View dient in jeder Weise als Anwenderschnittstelle. Diese Schicht erlaubt den Zugriff des Anwenders auf die Funktionalität des Programmes.
- **Controller:** Der Controller dient als Mediator zwischen **Model** und **View**. □

Differenzierung Beide Prototypen implementieren das **MVC Modell**.

Die Prototypen weisen eine grundlegende Differenzierung in ihrem

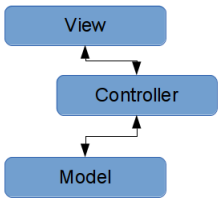
- **Datenmodell**
- **Konsistenzmodell**
- **Replikationsmodell** auf.

Der ein Prototyp wird den Prinzipien der **NoSQL** Ansätze folgend gestaltet, der andere nutzt die Theorie der verteilten **relationalen Datenbanken** für die Datenverarbeitung.

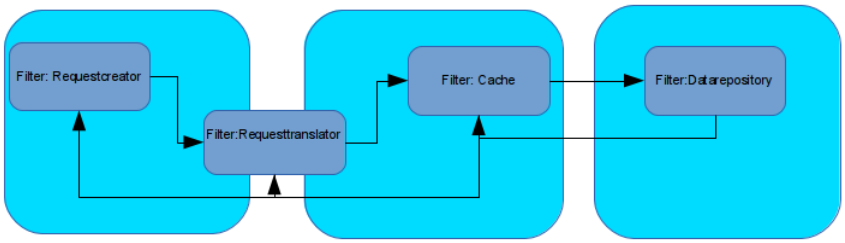
Das **MVC Modell** ist ein etabliertes Pattern im Bereich der Webanwendungen. In diesem Zusammenhang scheint unsere Entscheidung für das **MVC Pattern** nachvollzierbar und logisch. Das grundlegende Konzept des World Wide Webs als Client-Server System zwingt uns eine entsprechende Modularisierung unserer Anwendung auf. Die Layerstruktur ist eingebetet in eine einschließende Client-Server Architektur. Zur Entlastung des Servers setzten wir auf Smart Clients und serverseitg auf Replikation. □

Diagramme

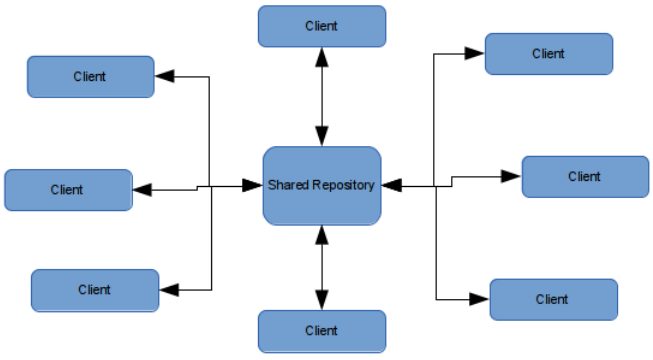
Layered View Darstellung der obersten Architekturschicht in einem Layered View



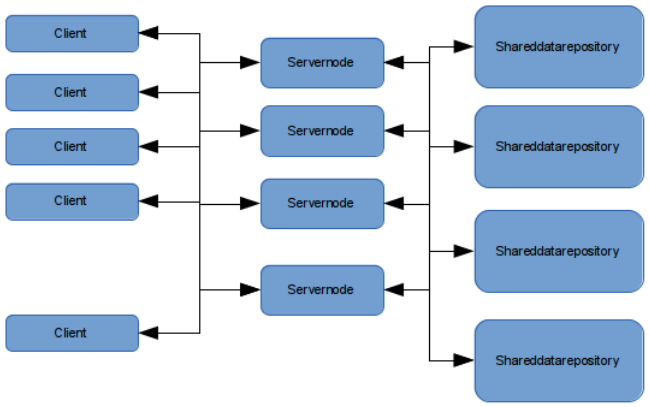
Dataflow View Darstellung der obersten Architekturschicht in einem Dataflow View



Datacentered View Darstellung der obersten Architekturschicht in einem Datacentered View



Componentinteraction View Darstellung der obersten Architekturschicht in einem Componentinteraction View



Alternative Designmöglichkeiten

Im Lichte eines wissenschaftlichen Diskurses möchten wir andere Architectural Patterns auf ihre Tauglichkeit für einen Einsatz in unserer Anwendung prüfen.

Folgender Katalog an möglichen Patterns soll näher diskutiert werden.

- **SOA**
- **ETL**
- **EAI/ESB**

Data Extraction Transformation and Loading (ETL) Das ETL Pattern findet in erster Line Einsatz in Enterprise Data Warehouse Anwendungen. Das Pattern ist primär darauf ausgerichtet **Datenanalyse** und **Reporting** transparent und performant zu gestalten. Wir bemerken einen grundlegenden Schwerpunkt im Datenverarbeitungsbereich.

Aufgaben eines ETL Systems:

- Das Pattern ist primär darauf ausgerichtet **Datenanalyse** und **Reporting** transparent und performant zu gestalten
- Ein ETL System besteht aus unabhängigen Komponenten, die nach Belieben vernetzt werden
- ETL unterstützt parallele Datenverarbeitung immenser Datenmengen

Das primäre Augenmerk in der Gestaltung der Architektur des Prototypen dient dem Laufzeitverhalten der Applikation und der linearen Skalierbarkeit der Anwendung, beides Kriterien, die im Zusammenhang des ETL Pattern zweitrangig sind

oder keine Bedeutung finden. ETL Systeme sind kaum optimiert in Hinsicht auf ihr Laufzeitverhalten. Globales Transaktionshandling ist kritisch. □

EAI/ESB Ein Enterprise Service Bus ist ein **Architektur Modell**, das eingesetzt wird um die **Interaktion** und **Kommunikation** von untereinander unabhängigen Softwareanwendungen zu gestalten. Als Softwarearchitektur für verteilte Systeme ist ESB eine Abstrahierung der **Client/Server** Architektur, die in erster Linie auf **Tranzparenz** und **Flexibilität** im Zusammenhang der Kommunikation und Interaktion setzt. ESB wird primär eingesetzt in Enterprise Application Integration (EAI).

Die Aufgaben eines **ESB** sind:

- **Monitoring** des Nachrichtenaustausches unter Applikationen
- **Verwaltung** des Versioning von Applikationen
- **Nachrichten** und **Ereignis Queueing**
- **Sicherheitsmanagement**
- **Ausnahmemanagement**

ESB wird eingesetzt, um **Interaktion** und **Kommunikation** von Services zu steuern. Im Zusammenhang unserer Anwendung besteht jedoch keine Notwendigkeit, mehrere Services untereinander zu vernetzen. Einige von einem ESB zur Verfügung gestellten Features finden auch in unserer Anwendung Eingang, unsere primären Anforderungen liegen jedoch in anderen Bereichen. □

Service-oriented architecture (SOA) SOA ist ein Softwaredesignansatz, der komplexe Softwareanwendungen als Komposition mehrerer unabhängiger Services versteht. Jedes Service stellt eine klar definierten Katalog an Funktionalität zur Verfügung. Ein Service wird unabhängig von den anderen Services erstellt.

SOA wird eingesetzt, um die Services untereinander zu vernetzen und eine homogene Funktionalität für den User zu erzeugen. Die Services können in unterschiedlichen Programmiersprachen erstellt werden. SOA integriert die unterschiedlichen Services zu einem Ganzen.

Eigenschaften von SOA:

- **Standardized service contract**
- **Service loose coupling**
- **Service abstraction**
- **Service reusability**

- **Service autonomy**
- **Service statelessness**
- **Service discoverability**
- **Service composability**

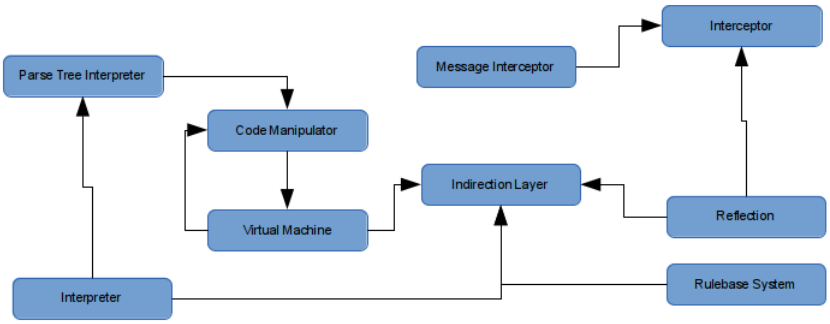
SOA wird eingesetzt, um Interaktion und Kommunikation von Services zu steuern. Im Zusammenhang unserer Anwendung besteht jedoch keine Notwendigkeit, mehrere Services untereinander zu vernetzen. Einige von SOA zur Verfügung gestellten Features finden auch in unserer Anwendung Eingang, unsere primären Anforderungen liegen jedoch in anderen Bereichen. □

View

Das View dient dem Anwender als Schnittstelle zur Anwendung. Die Architektur der View Schicht wird vorgegeben durch die von der Anwendung vorgegebenen Anforderungen.

Webbrowser Webbrowser stellen die **primäre clientseitige Technologiekomponente** im World Wide Web dar. Die Architektur eines Browsers entspricht der eines Interpreters. Die Aufgabe des Browsers ist es HTML Dokumente zu interpretieren und infolge zur Interaktion mit dem Anwender in einer grafischen Schnittstelle zur Verfügung zu stellen. In diesem Sinne wird uns die **technologische** als auch **architekturelle** Entscheidung auf oberster Ebene genommen. □

Architektur



Controller

Der Controller als Schicht dient als Mediator zwischen der **View-** und der **Modelschicht**. Der Controller wird sowohl auf der **Client Komponente** als auch auf der **Server Komponente** deployed.

Architektur

Um die Kommunikation zu den 2 anderen Schichten zu etablieren, wollen wir den Controller in 4 Teile zerlegen:

- **Model**
- **View**
- **Presenter**
- **AppController**

Model Dieser Teil des Controllers fungiert als Interface zum Model. Wir finden folgende Prinzipien:

- **Entityklassen:** Kodierung der unterschiedlichen Entitäten
- **EntityDetail:** Eine abgespeckte Version der Entityklassen. Die Detailklassen werden verwendet, um Daten über das Netzwerk zu schicken. □

View Dieser Teil des Controllers fungiert als Interface zur Viewschicht. Im View befinden sich alle **UI Komponenten**, aus denen unsere Anwendung besteht. Die Views sind ebenfalls verantwortlich für das Layout der UI Komponenten. Die Views als solche sind nicht gebunden an die Entitäten. Die Information, die verwaltet werden soll, bezieht sich ausschließlich auf UI Komponenten.

Für das Wechseln zwischen verschiedenen Views setzten wir auf History Management □

Presenter Der Presenter implementiert die Logik der Controllerschicht. Er beinhaltet eine Brücke zum

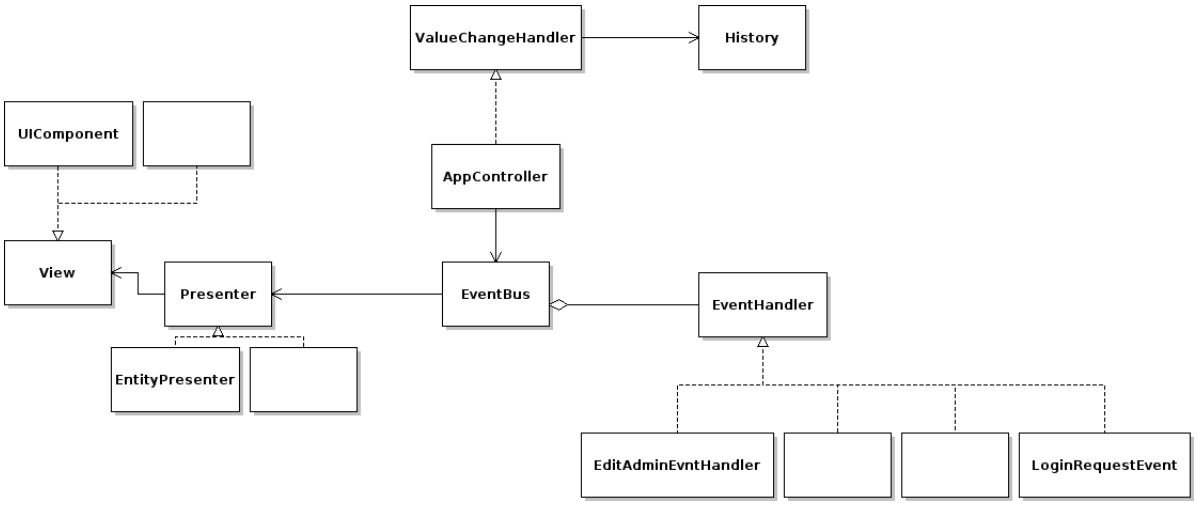
- **Historymanagement**
- **Viewtransition**
- **asynchrones RPC**

Als Faustregel gibt es für jede View einen Presenter der verantwortlich für die Verwaltung der View und der unterschiedlichen Events der UI Komponenten ist. □

AppController Um Logik zu implementieren die nicht spezifisch ist für die Presenter, setzten wir auf die AppController Komponente. Das **Historymanagement** und die **Viewtransition** Logik werden ebenfalls hier umgesetzt.

Der AppController ist zusätzlich verantwortlich für die **Initialisierung** und **Verwaltung** des Eventbuses und der Events. □

Eventbus Haben die Presenter ersteinmal einen Event der UI Komponenten abgesetzt, müßen wir in der Lage sein, auf jene zu reagieren. Wir setzten auf einen EventBus, um die Events zu verwalten und an registrierte Komponenten weiterzuschicken. □

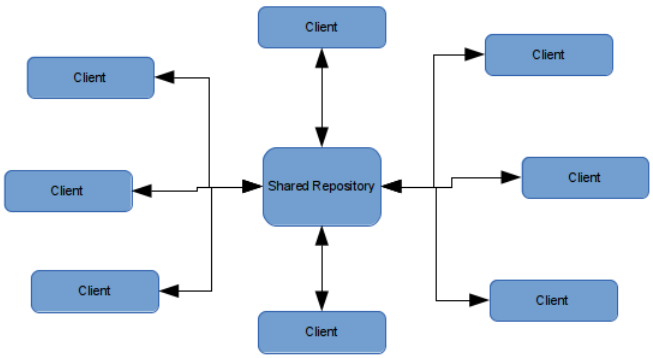


Model

Um eine Vergleichbarkeit der beiden Prototypen etablieren zu können, versuchen wir die beiden Anwendungen so ähnlich, wie möglich zu halten. Von der Architektur her entsprechen sich die Modelschichten beider Prototypen, hinter den **abstrakten Konzepten** stehen jedoch unterschiedliche Implementierungen.

Architektur

Datacentered View Darstelleung des Models im Datacentered View



Componentinteraction View Die grundlegenden 2 Komponenten der Modelschicht sind:

- **EntityManager:** Der EntityManager wird eingesetzt als **Fassade**. Hinter der Implementierung des EntityManagers stehen die unterschiedlichen Datenverarbeitungsbezogenen Technologien: **RDBMS** und **NoSQL**.

  Die Fassade dient uns als Interface zu den unterschiedlichen Technologien.

- **EntityKlassen**

CHAPTER 7

Durchführung der Lastentests

Einführung

Lastentests werden eingesetzt, um das **Laufzeitverhalten** und die **Skalierbarkeit** der Prototypen zu bewerten und zu verstehen. Das Laufzeitverhalten ist in diesem Sinne als die Summe der unterschiedlichen Paradigmen für **Datenkonsistenz**, **Replikation** und des **Datenmodells** zu beurteilen.

Beide Prototypen werden denselben Lastentests unterzogen. Der Vergleich der Ergebnisse der Lastentests gibt Auskunft über die wahre Beschaffenheit der unterschiedlichen Konzepte. Die gefundenen Ergebnisse helfen uns, die gewünschten Eigenschaften zu prüfen und eine finale Architekturentscheidung zu treffen. □

Test Framework

Zur Durchführung der **Lastentests** wird ein Testclient erstellt. Der Testclient wird eingesetzt um **HTTP Requests** an den Server zu schicken. Am Server werden als Antwort darauf mehrer Operationen ausgelöst um die Datenverabeitungkomponente zu testen. □

Testgenerator Architektur

Der **TestClient** ist ein Java Programm zur Ausführung von Testcases und zur Evaluierung der Prototypen. Die primären Komponenten des TestClients sind:

- **Workexecutor:** Der Workexecutor führt Workloads aus. Jedem Workload werden mehrer Threads zugewiesen um die anstehenden Operationen auszuführen. Diese Komponente steuert ebenfalls den Workload und Throughput der Testcases.

  Der Workexecutor ist im allgemeinenen zuständig für die Generierung von Daten und dem Ausführen von Operationen.

- **Threadpool:** Der Threadpool ist eine ressourceeffiziente Struktur zur Verwaltung der Threads.

- **Interface Layer:** Der Interface Layer stellt die Schnittstellt des Programms zu den Prototypen dar. □

Workload

Ein Workload bestimmt die Zusammensetzung von Schreib-/ Leseoperationen, Datengrösse und Requestverteilung eines Testcases. Als Schreiboperation verstehen wir alle **CRUD Operationen** ohne den Leseoperationen.

Wir sind in der Lage unterchiedliche Arten von Testdaten zu generieren:

- **read-heavy Workload:** Der Workload beinhaltet 5% update und 95% read Operationen.

- **write-heavy Workload:** Der Workload beinhaltet 95% update und 5% read Operationen.

- **balanced Workload:** Der Workload beinhaltet 50% update und 50% read Operationen. □

Testkriterien

Die Tests sind ausgelegt um mehrere unterschiedliche Kriterien im System zu evaluieren. Wir versuchen die folgenden Größen zu testen:

- **Performance**
- **Skalierbarkeit**
- **elastische Verhalten**
- **Schreib-/Leseverhalten**

Performance Die Performance bezogenen Testcases werden durchgeführt um die durch datenbezogene Operationen hervorgerufenen Latenzzeiten im System zu evaluieren.

Durch kontinuierliche Steigerung der Systemlast soll, bei gleichzeitiger Messung der Latenzzeiten jener Punkt gefunden werden, an dem das System an seine Grenzen stößt. Im Rahmen dieser Tests wird bewußt auf Skalierung verzichtet. □

Skalierung Die Skalierung bezogenen Testcases werden eingesetzt um die Geschwindigkeit des Systems zu messen im Bezug auf die Eigenschaft des System zu skalieren. In diesem Zusammenhang evaluieren wir 2 Größen:

- **Scale up Fragestellung:** Wie verhält sich das System während die Zahl der Knoten erhöht wird?

  Um die Tests durchzuführen, evaluieren wir die Performance des Systems in einem bestimmten Ausgangszustand. Anschließend werden die Daten gelöscht, die Zahl der Knoten erhöht und die Tests erneut durchgeführt. Skaliert das System sollte die Performance gleich bleiben.

- **Elastic speedup Fragestellung:** Wie verhält sich das System während die Zahl der Knoten zur Laufzeit erhöht wird?

  Die Tests werden durchgeführt und ausgewertet während schrittweise die Zahl der Knoten erhöht wird. □

Test-Setup

Zum Durchführen der **Tests** werden die **Datenverwaltungssysteme** zunächst mit Testdaten versorgt. Das **Testframework** ist darauf ausgelegt die entsprechenden Testdaten automatisch zu generieren und den entsprechenden **Workloads** zuzuordnen. Im folgenden soll die Beschaffenheit der Testdaten, des Workloads und der Hardwarekonfiguration des Systems beschrieben werden.

Testdaten Jeder **Eintrag** hat eine Größe von 500 Bytes. Ein **Eintrag** setzt sich in diesem Sinne zusammen aus

- **Schlüssel:** Typ: varchar
- **Felder:** Typ: varchar □

Workload Jeder Workload ist so angelegt das 200 000 000 Einträge verarbeitet werden. Zusammenstellung des Workloads:

- 200 000 000 **Einträge**
- Größe des Eintrags: 500 Byte
- 5 Felder zu 100 Bytes pro **Eintrag**
- Parallele Verarbeitung der Tests (**ThreadPool**) □

Google Cloud Beide **Prototypen** werden als Teil der **Google Cloud** deployed. Das System stellt uns sowohl eine Anbindung an **Googles BigTable**, als auch an ein **RDBMS**(**MySQL**) System bereit. Die Test werden in der Google Cloud ausgeführt. (Stand der Technologie **Mai 2013**).

Die Google Cloud kommt automatisch mit folgenden Featuren:

- **lineare Skalierung**
- **Fast Data Access**
- **Unlimited Storage**
- **BigQuery**

Je nach Kontrakt werden dem Kunden unterschiedliche Hardware Konfigurationen zur Verfügung gestellt.

Die Test wurden auf Serverclustern mit folgender Hardware ausgeführt:

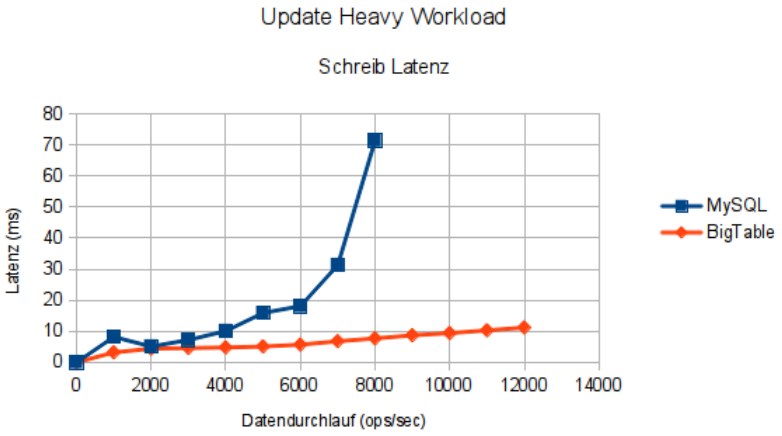
- **Prozessor:** 64 Bit quad core CPU

- **RAM:** 3.75 GB
- **HD:** nach Vereinbarung

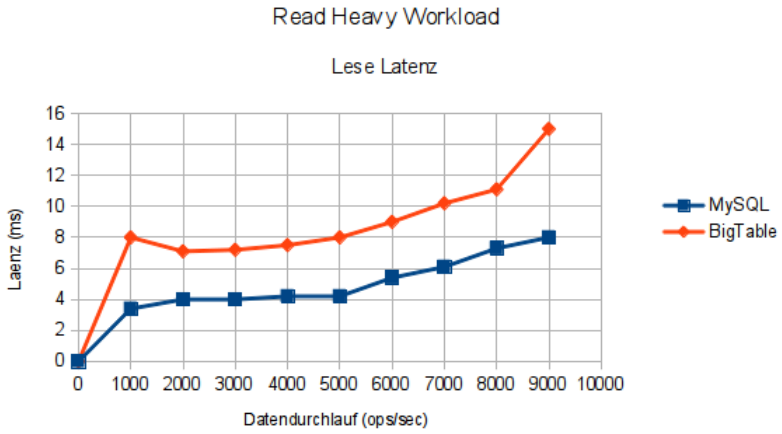
Testdurchführung

Performance-Test

Schreiboperationen Um die Beschaffenheit des **Schreibverhaltens** der beiden **Prototypen** zu testen wird das **Update-Heavy Workload** geladen und bei kontinuierlicher Steigerung der Last die **Latenz** der Systeme gemessen.

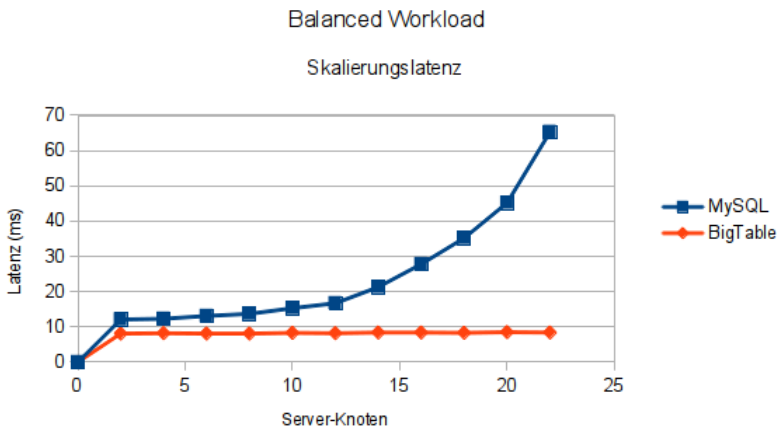


Leseoperationen Um die Beschaffenheit des **Leseverhaltens** der beiden **Prototypen** zu testen wird das **Read-Heavy Workload** geladen und bei kontinuierlicher Steigerung der Last die **Latenz** der Systeme gemessen.



Skalierbarkeits-Test

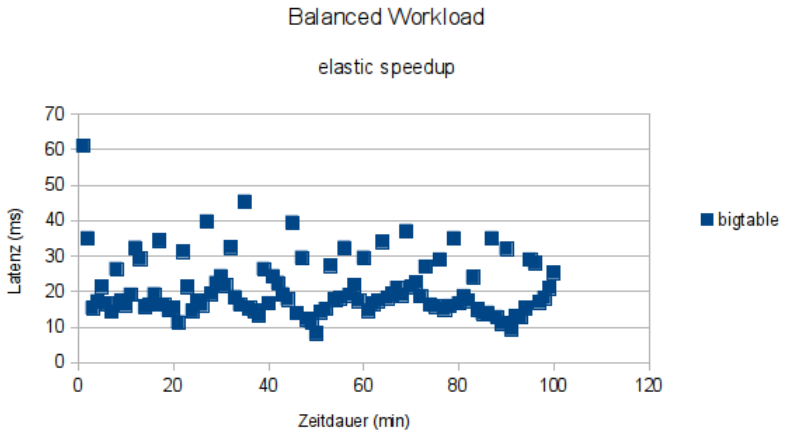
Um die Skalierungsfähigkeit der beiden Prototypen zu messen wird das **Balanced Workload** geladen. Im Gegensatz zu den Performancetests, bei denen die Zahl der Serverkoten während des Tests **konstant** gehalten wurde, wird nun die Zahl der Server **proportinal** mit der Zahl der Operationen erhöht.



elastic Speedup

Ein verteiltes System das in der Lage ist **Server Knoten**, ohne erwähnenswerte **Latenz**, dem System hinzuzufügen besitzt die Eigenschaft des **Elastic Speedup**. Wir laden das **Balanced Workload**. Anschließend werden **Server Knoten** dem System hinzugefügt. In diesem Fall messen wir die Zeit die das System braucht um den neuen **Server Knoten** zur Laufzeit in das System zu integrieren. Ein Server nach dem anderen wird dem System einzeln hinzugefügt. Anschließend wird gewartet bis sich das System wieder stabilisiert hat und keine zusätzliche **Latenz** mehr erzeugt.

MySQL für sich erfüllt in keiner annehmbaren Zeitdauer die Kriterien für **elastiches Speedup**. Die Messungen stabilisieren sich erst nach einer geraumen Zeit.



Resultat

Performance-Tests

Die **Performance Tests** sollen helfen die grundlegenden Eigenschaften **(CRUD Operationen)** der beiden Prototypen zu messen. Die **Performance-Tests** werden ohne **Replikation** durchgeführt. □

Readheavy Workload Um die Fähigkeit der Systeme zu messen, große Mengen von Daten schnell lesen zu können, wird ein **Workload** geladen der sich zu 95% aus **Leseoperationen** zusammensetzt. Bei der Durchführung des Test wird die Zahl der Operationen kontinuierlich erhöht und die **Latenz** der Systeme gemessen. Beide Prototypen erzielen bei diesem **Test** sehr gute Ergebnisse. Die gemessene **Latenz** bleibt mit steigender **Systemlast** über die längste Zeit hinweg nahezu **linear**. Erst ab einem gewissen Wert für den **Datendurchlauf** erkennen wir einen signifikanten Anstieg in der Latenz.

Daß **MySQL** in dieser Testreihe leicht zu fovorisieren ist, ist auf den Umstand zurück zuführen, daß **RDBMS** Systeme im Allgemeinen **lese optimiert** sind. RDBMS Systeme erlauben es nicht nur der Anwendung schnell, große Mengen von Daten lesen zu können, es wird zusätzlich Funktionalität zur Verfügung gestellt um Daten schnell abfragen zu können. Dazu werden die Daten im System indiziert. Was für das **Leseverhalten** in diesem Fall günstig ist, erweist sich für das **Schreibverhalten** als nachteilhaft. Werden neue Daten geschrieben muß der **Index** jedesmal aufs neue konfiguriert werden.

Besteht der gegebene **Workload** eines Datenverarbeitungssystems in erster Linie aus **Leseoperationen** oder grundlegenden **Aggregatfunktionen** (z.B.: SUM(), AVG()) so ist in diesem Fall den **RDBMS** Systemen ein Vorzug gegenüber den **NoSQL** Systemen zu geben. □

Updateheavy Workload Um die Fähigkeit der Prototypen zu messen große Mengen von Daten verändern zu können, wird ein **Workload** geladen der zu 95% aus Schreiboperaionen besteht.

BigTable weist während des Tests das weit bessere **Latenzverhalten** auf. Die Veränderung der **Latenzzeit** entwickelt sich generell **linear**. Wir wissen daß **RDBMS** Systeme leseoptimiert sind. Im Gegensatz dazu können **NoSQL** Systeme als **schreiborientiert** bezeichnet werden. Diese Systeme wurden darauf ausgelegt große Mengen von Daten schnell schreiben zu können.

Mehrere Aspekte von **NoSQL** Systemen zeichnen sich im Allgemeinen für das vorteilhafte **Schreibverhalten** von **Bigtable** im Test aus:

- **Datenschema:** **NoSQL Datenschemas** sind darauf konzipiert, große Mengen von Daten schnell schreiben zu können.

  NoSQL Datenschemas sind in der Regel schemaless. Müssen große Blöcke von komplexen, zusammenhängenden Daten geschrieben werden, so erfolgt das im Normalfall in einer einzelenen **Schreiboperation**. Die Daten werden dabei in einer einzelnen **Zelle** oder in einem **Dokument** abgelegt. Die Aufgabe des Programmieres ist es dabei die Daten so zu strukturieren, daß Abhängigkeiten zwischen den Daten hierarchisch modelliert werden können.

  Um die **Konsistenz** der Daten sicherzustellen, setzten **RDBMS** Systeme auf **Transaktionen**. In **NoSQL Systemen** kennen wir das Konzept der Transaktion in der konservativen Interpretation nicht. In **NoSQL** Systemen gibt es kein schema on demand. Abhängigkeiten zwischen Daten werden auf der **Entwurfsebene** und nicht auf der **Systemebene** behandelt. □

- **Konsistenzmodell:** Das **BASE** Konsistenzmodell von **BigTable** begünstigt das vorteilhafte Schreibverhalten des Systems.

  - **Basic Avaliability**
  - **Soft State**
  - **Eventual Consistency**

  Soft State als Konsistenzeigenschaft bedeutet, daß die Daten im System nicht zu jedem Zeitpunkt konsistent sei müssen. **Eventual Consistency** besagt daß die Daten zu irgendeinem Zeitpunkt konsistent sein werden. Dieses epidemische **Konsistenzmodell** ermöglicht ein wesentlich effizienteres **Schreibverhalten**. □

Analysieren wir das **Schreibverhalten** des **MySQL Systems** so bemerkt man daß nach einer gegebenen linearen Entwicklung der **Latenzzeit** bei steigender **Systemlast** die Funktion ein exponentielles Wachstum erfährt. Das System ist ab einem bestimmten Zeitpunkt nicht mehr in der Lage die **Systemlast** in einem vorgegebenen Rahmen zu verarbeiten.

Mehrer Aspekte von **RDBMS** Systemen sind für dieses Verhalten verantwortlich:

- **ACID Konsistenzmodell**
- **Indizierung der Daten**
- **relationale Struktur der Daten**
- **Master/Slave Systemstruktur**

Skalierbarkeits-Tests

Als grundlegende nicht funktionale Anforderungen an unser System haben wir ein performantes Laufzeitverhalten und die Skalierbarkeit des Systems gefordert. Ist ein System **skalierbar**, ist es in der Lage trotz wachsender **Systemlast** den Systemservice adäquat aufrechtzuerhalten.

Durchführung der Tests:

- **Workload:** Zur Durchführung unsere Tests wurde der **balanced Workload** geladen.
- **Test:** Um die **Skalierbarkeit** des Systems messen zu können wird über eine bestimmte Zeit hinweg die Zahl der **Serverknoten** kontinuierlich erhöht. Gleichzeitig wird proportional dazu die **Serverlast** gesteigert.
- **Messung:** Im Anschluß daran messen wir die entstehenden **Latenzzeiten**.

MySQL Resultat Die Messungen für den **MySQL Prototypen** zeigen einen kleinen Anstieg in den **Latenzzeiten**, der jedoch immer größer ausfällt und sich zunehmend exponentiell entwickelt.

RDBMS Systeme setzen im Gegensatz zu NoSQL Systemen auf den Betrieb mehrerer Servern durch **Clustering**. Dieser Ansatz ist jedoch nur für eine beschränkte Zahl von Servern sinnvoll, da ab einem gewissen Punkt das Systen nicht mehr skaliert. Ein Grund für dieses Verhalten ist die Tatsache, daß **RDBMS Systeme** eine Master/Slave Architektur benötigen um die Konsistenz der Daten gewährleisten zu können. Die Notwendigleit dieses Konsistenzverhaltens leitet sich wiederum aus den ACID Eigenschaften des Transaktionsmodells relationaler Datenbanken her.

In diesem Umfeld erweist sich der relationale Transaktionsansatz mit dem Sperren von Datensätzen als Haupthindernis. Zum einen ist der Zeit- und Kommunikationsaufwand für eine Übereinkunft über das **Setzen** und **Aufheben** einer Sperre groß, zum anderen kann auf einen gesperrten Datensatz nicht lesend zugegriffen werden.

Bigtable Resultat Betrachten wir die Entwicklung der **Messfunktion** für den Bigtable Prototypen so bemerken wir ein schwaches lineares Wachstum der Funktion. Der Prototyp zeigt in diesem Zusammenhang ein beinahe ideales Verhalten.

NoSQL als Technologie ist mit dem Anspruch angetreten **skalierende Syteme** im Datenverarbeitungsbereich zur Verfügung zu stellen. Der gesamte **Technologiestack** wurde unter dem Augenmerk entwickelt linear zu skalieren. Die grundlegende Idee des **NoSQL** Ansatzes ist es die **Softwarekomponente** des Systems an die **Hardwarekomponenete** anzupasen. Dieser Ansatz ist es der lineare Skalierbarkeit ermöglicht.

Technologiekonzepte zur Umsetzung **linearer Skalierbarkeit** des Prototypen:

- **Consistent Hashing**
- **BASE Konsistenzmodell**
- **MVCC**

Elastic Speedup

Als **elastic speedup** wird die Eigenschaft eines **verteilten Systems** bezeichnet, zur Laufzeit **Server Knoten** in das System zu integrieren ohne eine spürbare Erhöhung der **Latenzzeit** zur Laufzeit zu bemerken. Ergibt sich eine erhöhte **Latenzzeit** so ist das darauf zurückzuführen daß das System die neuen Komponeneten erst adäquat initialisieren und integrieren muß. **Skalierende Systeme** sollten so **elastisch** wie möglich sein. □

Bigtabel Test Das System verhält sich in der Regel **elastisch**. Betrachten wir die **Messungen** für den Prototypen so merken wir ein gewisses **Rauschen** in den Latenzzeiten, jedesmal wenn neue **Serverknoten** in das System integriert werden. Das **Rauschen** wird verursacht durch das **Datenmodell** des Systems. Jedesmal wenn Server Knoten in das System **integriert** oder aus dem System **gelöscht** werden, beginnt der **Consistent Hashing Algorithmus** damit Daten entsprechend umzukopieren und **Replikate** zu erstellen. □

MySQL Test **MySQL** wurde nicht darauf konzipiert **elastisch** zu skalieren. □

CHAPTER 8

Etablierung der Ergebnisse

Forschungsfrage

Wir sind mit der Frage angetreten ob es für eine bestimmte **Klasse** von verteilten, datenverarbeitenden Systemen möglich ist ein **performantes Laufzeitverhalten** und **Skalierbarkeit** zu fordern, bei der gleichzeitigen Verwendung eines **RDBMS Systems** als Kern dieses Systems.

Klasse der in Berücksichtigung zu ziehenden Softwaresysteme Die für uns interessante **Klasse von Softwareanwendungen** möchten wird durch die für sie charakteristischen Eigenschaften beschreiben. Die gewünschte Klasse findet sich in der **Durchschnittsmenge** der durch folgende Eigenschaften und Anforderungen beschriebenen Softwaresysteme:

- **verteiltes System**
- **basierend auf dem HTTP Protokoll**
- **unterstützt HTML 5**
- **hoch skalierbar**
- **Datenverarbeitung im Terrabyte Bereich**
- **performantes Laufzeitverhalten** □

Forschungsfrage Die grundlegende Frage in diesem Zusammenhang ist damit:

Soll zur Etablierung eines verteilten webbasierten Client-Server Systems mit Millionen von gleichzeitigen Anwenderzugriffen und der Notwendigkeit Daten im Peta Byte Bereich zu bearbeiten bei gleichzeitig performantem Laufzeitverhalten, ein vertikal oder ein horizontal skalierendes System eingesetzt werden? □

Methodisches Vorgehen

Um die **Forschungsfrage** beantworten zu können wurden die folgenden **Arbeitsschritte** durchlaufen:

- **Technologievergleich**
- **Entwurf der Prototypen**
- **Ausführung von Lastentests**
- **Auswertung**

Technologievergleich

CAP Theorem Das **CAP Theorem** etablierte für uns eine **Klassifizierung** von verteilten datenverarbeitenden Systemen. Aus der Diskussion des Theorems extrahierten wir 2 Klassen von Systemen die für einen Vergleich herangezogen werden sollten. **(CA Systeme vs CP Systeme)** □

CP- und AP Systeme - NoSQL: **NoSQL** versteht sich als Sammlung diverser unterschiedlicher Technologien und Paradigmen, die mit dem Anspruch angetreten sind, skalierbar und hoch performant zu sein. Eine **NoSQL** Datenbanksystem ist als dezentrale Struktur aufgebaut. Durch intelligente **horizontale Skalierung** wird das Bild von nahezu uneingeschränkt verfügbarer Ressourcen geformt.

Die folgenden Technologien werden in **NoSQL systemen** eingesetzt:

- **Key-Value Datenmodell**
- **Consistent Hashing**
- **MVCC**
- **Spaltenorientiertes Datenmodell**
- **Dokumentorientiertes Datenmodell**
- **Map/Reduce** □

CA Systeme - RDBMS: **Relationale Datenbanksysteme** wurden entwickelt um den wachsenden Ansprüchen einer neuen Generation von Applikationen gerecht zu werden. **RDBMS Systeme** wurden entwickelt um folgende Probleme im Datenverarbeitungsbereich zu lösen:

- **Redundanz und Inkonsistenz** von Daten
- **beschränkte Zugriffsmöglichkeiten**
- **Herausforderungen des Mehrbenutzersbetriebs**
- **Verlust von Daten**
- **Integritätsverletzung**
- **Sicherheitsprobleme** □

NoSQL vs RDBMS Systeme **RDBMS Systeme** setzen im Gegensatz zu **NoSQL** Systemen auf den Betrieb mehrerer Servern durch **Clustering**. Dieser Ansatz ist jedoch nur für eine beschränkte Zahl von Servern sinnvoll, da ab einem gewissen Punkt das Systen nicht mehr skaliert. Ein Grund für dieses Verhalten ist die Tatsache, daß **RDBMS Systeme** eine **Master/Slave Architektur** benötigen um die **Konsistenz** der Daten gewährleisten zu können. Die Notwendigleit dieses Konsistenzverhaltens leitet sich wiederum aus den **ACID** Eigenschaften des Transaktionsmodells **relationaler Datenbanken** her. Um die geforderten Antwortzeiten erreichen zu können, mit mehreren Hunderttausend oder auch Millionen von Benutzern, bei gleichzeitiger **Verarbeitung** von Datenmengen im Tera- und Peta-Byte-Bereich, muß das **relationale Konsistenzmodell** hinterfragt werden. In diesem Umfeld erweist sich der **relationale Transaktionsansatz** mit dem Sperren von Datensätzen als Haupthindernis. Zum einen ist der Zeit- und Kommunikationsaufwand für eine Übereinkunft über das Setzen und Aufheben einer Sperre groß, zum anderen kann auf einen gesperrten Datensatz nicht lesend zugegriffen werden.

NoSQL versteht sich als Sammlung diverser unterschiedlicher Technologien und Paradigmen, die entwickelt wurden, um die Restriktionen des **ACID** Konsistenzmodells zu umgehen, ohne auf **Datenkonsistenz** verzichten zu müssen.

Wir isolierten das **Laufzeitverhalten** und die **Skalierbarkeit** als die entscheidenden nichtfunktionalen Anforderungen in unserer Klasse von Softwareystemen. Ein performantes **Laufzeitverhalten** in Verbindung mit der Notwendigkeit, **Millionen von Benutzerrequests** zur selben Zeit zu bearbeiten, ist die Quadratur des Kreises, die es zu meistern gilt.

Eine Vielzahl von Requests gleichzeitig zu bearbeiten, birgt die Notwendigkeit, die Anwendung auf einen Cluster von Rechnern zu verteilen. Zur nichtfunktionalen Anforderung des Laufzeitverhaltens gesellt sich die Forderung nach einem skalierenden System. Das **Laufzeitverhalten** und die **Skalierbarkeit** des Systems sind Größen, die in engem Zusammenhang mit anderen Grössen und Konzepten stehen. □

RDBMS vs NoSQL: Übersicht

	CA Systeme	CP Systeme
Skalierbarkeit:	horizontale Skalierbarkeit	vertikale Skalierbarkeit
Konsistenzmodell:	ACID	BASE
Datenschema:	schemaless	Schema
Datastore:	RDBMS	NoSQL

Skalierbarkeit **Skalierbarkeit** wird eingesetzt, um die Verfügbarkeit von Anwendungen im industriellen Umfeld gewährleisten zu können. Ein Sofwaresystem wird als **skalierbar** bezeichnet, wenn das System in der Lage ist gleichzeitig zur wachsenden **Systemlast** den vereinbarten Dienst zu gewährleisten.

Generell wird unterschieden zwischen

- **horizontale/lineare Skalierung**
- **vertikaler Skalierung** □

Klassifizierung von Konsistenzmodellen

- **Strict Consistency**
- **Eventual Consistency**
- **Functional Partitioning** □

Vergleich ACID vs BASE

ACID	BASE
strenge Konsistenz	schwache Konsistenz
Isolation	Verfügbarkeit
Fokus auf commit	Inetwa richtige Antworten
verschachtelte Transaktionen	schnell
konservativ/pessimistisch	aggressiv/optimistisch
komplizierte Evolution	einfache Evolution

Entwurf der Prototypen

Im Zuge der Arbeit haben wir einen Katalog von Anforderungen erarbeitet, der die Rahmenbedienungen zur Gestaltung der **Architektur** und technischen **Umsetzung** einer bestimmten Klasse von Applikationen definiert. Wir haben die unterschiedlichen **Konzepte** und **Paradigmen** kennengelernt, die für uns relevant sind, um das Problem zu verstehen und adäquat zu interpretieren.

Zur Etablierung unserer Ergebnisse werden 2 Protypen erstellt und geprüft. Die Prototypen werden unterschiedlichen Paradigmen folgend entwickelt. Zum Vergleich werden die Prototypen identischen Lastentests unterzogen.

RDBMS vs NoSQL Beide Prototypen implementieren das MVC Modell. Die Herausforderung in der Gestaltung beider Programme ist es, sowohl das **View** als auch den **Controller** für beide identisch zu halten. Lediglich in der Model Schicht streben wir eine grundlegende Differenzierung im Sinne der Implementierung an.

Die Prototypen weisen eine grundlegende Differenzierung in ihrem **Daten-**, **Konsistenz-** und **Replikationsmodell** auf. Ein Prototyp wird den Prinzipien der NoSQL Ansätze folgend gestaltet, der andere nutzt die Theorie der verteilten relationalen Datenbanken für die Datenverarbeitung.

Ausführung der Lastentests

Lastentests werden eingesetzt, um das Laufzeitverhalten und die Skalierbarkeit der Prototypen zu bewerten und zu verstehen. Das Laufzeitverhalten ist in diesem Sinne als die Summe der unterschiedlichen Paradigmen für **Datenkonsistenz**, **Replikation** und des **Datenmodells** zu beurteilen.

Beide Prototypen werden denselben Lastentests unterzogen. Der Vergleich der Ergebnisse der Lastentests gibt Auskunft über die wahre Beschaffenheit der unterschiedlichen Konzepte. Die gefundenen Ergebnisse helfen uns, die gewünschten Eigenschaften zu prüfen und finale Architekturentscheidung zu trefen. □

Auswertung

Wir haben ein performantes Laufzeitverhalten und die Skalierbarkeit des Systems als primäre **nicht funktionale Anorderung** an das System gestellt. Die zentrale Frage ist ob es möglich ist ein solches System unter der Verwendung eines **RDBMS** Systems als Kern umzusetzten.

Betrachten wir die Ergebnisse für die **Skalierbarkeit** des MySQL Prototypen bemerken wir daß das System nur horizontal skaliert. Unser Anspruch besteht jedoch darin ein System zur Verfügung zu stellen, daß linear skaliert. Diesen Anspruch erfüllt aber nur der **NoSQL Prototyp**. Wir müssen feststellen daß eine **RDBMS** Lösung für unser Problem nicht in Frage kommt. □

Bibliography

[1] Tannenbaum, Steen, 2003. *Verteilte Systeme - Grundlagen und Paradigmen*, Pearson Studium.

[2] Graham, Knuth, Patashnik, 1989. *Concrete Mathematics: A Foundation for Computer Science*, Addison-Wesley.

[3] Heuer, 1997. *Objektdatenbanken - Konzepte, Sprachen, Architektur*, International Thomson Publishing.

[4] Heuer, 2008. *Datenbanken - Konzepte und Sprachen*, mitp Verlag.

[5] Saake, 2005. *Datenbanken: Implementierungstechniken*, mitp Verlag/Bonn.

[6] R. A. Elmasri 2009. *Grundlagen von Datenbanksystemen*, Addison-Wesley.

[7] Edlich, Friedland, Hampe 2010. *NoSQL- Einstieg in die Welt der nichtrelationalen Web 2.0 Anwendungen*, Hanser Verlag.

[8] White, 2011. *Hadoop, The Definite Guide, 2ndEdition*, O'Reilly Media Inc.

[9] Lam, 2010. *Hadoop in Action*, Manning Publications.

[10] Date, 1994. *Mehrrechner-Datenbanksysteme*, Addison-Wesley.

[11] Aviziens, Laprie, Randell, 2001. *Concepts of Dependability*, Research Report No 1145, LAAS-CNRS.

[12] Brewer, 2000. *Towards Robust Distributed Systems*, Folien zur Keynote des 19 ACM SIGACT-SIGOPS Symposium

[13] DeCandia, Hastorun, Jampani 2011. *Dynamo, Amazon's highly available Key-Value Store*, SIGOPS Operating Systems Rev. New York, *pages 205-220*

[14] Karger, Lehman, Leighton 2011. *Consistent Hashing and Random Trees: Distributed Protocols for Relieving Hot Spots on the WWW*, STOC'97, Proceedings of the 29th anual ACM, Symposium on Theory of Computing, El Paso, Texas, USA, 1997, *pages 654-663*

[15] Chang, Dean, Ghemawat, Burrows, Chandra 2006. *Bigtable: A Distributed Storage System for Structured Data*, Google inc., *pages 205 - 218*

[16] Gilbert, Lynch, 2002. *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services*, http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf

[17] http://www.ocg.at/de/biber

[18] Wikipedia Beitrag, Pittimann, 2013. *HTTP*, http://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol

[19] Wikipedia Beitrag, Lippold, 2013. *HTML 5*, http://de.wikipedia.org/wiki/HTML5

[20] SETI@home, https://en.wikipedia.org/wiki/SETI@home

[21] Java Wiki http://de.wikipedia.org/wiki/Java

[22] Avgeriou, Zdun. *Architectural Patterns Revisited – A Pattern Language*, http://hydra.infosys.tuwien.ac.at/staff/zdun/publications/designSpacePatterns.pdf

[23] White Paper by Datastax Corporation 2013. http://www.datastax.com/wp-content/uploads/2013/02/WP-Benchmarking-Top-NoSQL-Databases.pdf

Index