Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Seminararbeit
Cloud Data Management
bei Prof. E. Rahm
im Bachelorstudiengang Informatik

# Key Value Stores
# BigTable, Hadoop, CouchDB

**eingereicht von:** Jonas Brekle

**eingereicht am:** 29. Januar 2010

**Betreuer:** Anika Groß

# Inhaltsverzeichnis

# 1 Einführung

In dieser Seminararbeit wird das Konzept von Key Value Stores vorgestellt und am Beispiel von BigTable und der Open Source Implementierung HBase, sowie an CouchDB, das einen ganz anderen Ansatz verfolgt, mögliche Umsetzungen gezeigt.
Die Arbeit ist Teil des Cloud Data Management Seminars des Datenbank Lehrstuhls von Professor Rahm und wird ergänzt von Simon Bins Seminararbeit.
In der Einführung möchte ich einen Einblick in die Entstehung und den theoretischen Hintergrund gewähren. Siehe dazu auch [1].
In den letzten 10 Jahren hat sich die Welt der Informationsverarbeitung entscheidend verändert: große Datenmengen fallen jetzt nicht mehr nur in sehr großen Firmen an, die zum Beispiel Terabyte an Auftragsdaten über Jahre ansammeln oder besondere Berechnungen durchführen müssen. Auch kleinere Unternehmen, die vielleicht erst vor ein paar Monaten gegründet wurden und zum Beispiel Online-Dienstleistungen anbieten, haben ein erhöhtes Bedürfnis mit großen Datenmengen umgehen zu können. Relationale Datenbanken stoßen hier in Hinsicht auf Skalierbarkeit an ihre Grenzen. Die Annahme, dass mit großen Daten auch große Kosten und lange Rechenzeiten verbunden sind gilt nicht immer, wenn man für diesen Zweck optimierte Datenbanken und Datenmodelle verwendet.
Google oder Social-Media-Giganten wie Twitter, Youtube, Facebook, aber auch unzähligen kleinen Anbietern, haben zusammen hunderte Millionen von Nutzern und benötigen die fast unbegrenzte Verfügbarkeit von großen Daten, effizienter Kommunikation und Informationen über soziale Strukturen. Daten werden zur Ware bzw. zum Rohstoff unserer digitalen Welt. Anfragen von beliebigen Orten müssen gleichzeitig bearbeitet, komplexe Suchen und Berechnungen parallel ausgeführt, I/O-intensive Operationen skalierbar gemacht werden.
Beispielsweise ist es für Nutzer von Facebook selbstverständlich, dass die Suche nach einer Person *ohne wahrnehmbare Wartezeit* verläuft, obwohl 300.000.000 Menschen einen Account haben; oder dass eine Suche bei Google über alle indizierten Webseiten des Internets nur den Bruchteil einer Sekunde dauert und auch noch die besten Ergebnisse ganz oben stehen. Im Gegensatz zu professionellen Nutzern (Unternehmen), die für geringe Latenz und hohe Verfügbarkeit bezahlen, erwarten Endanwender oft dass dies ein selbstverständlicher Teil der Dienstleistung ist. Unternehmen müssen dies über effiziente Datenhaltung realisieren, die Datenvolumina unterstützt, die ein vielfaches von gewöhnlichen unternehmensinternen Datenbanken ausmachen.

## 1.1 SQL

Am weitesten verbreitet in der Wirtschaft sind relationale Datenbanken mit SQL als Abfragesprache. Das relationale Modell ist wissenschaftlich fundiert, seit den 1960er Jahren stetig weiterentwickelt und standardisiert wurden und es existieren verschiedene, sehr ausgereifte Implementierungen.
SQL ist eine mächtige Abfragesprache zur Analyse und Extraktion großer Datenmengen aus relationalen Tabellen, die einem festen Schema folgen. Die Konsistenzerhaltung

steht im Mittelpunk und wird bei jeder Transaktion erzwungen. Schemata müssen normalisiert sein; Hierarchische Informationen müssen aufwändig kodiert werden und resultieren in komplexen Schemata.
Es sind umfangreiche eingebaute Operationen wie Outer- und Inner Joins, Unions, Groups und Aggregatfunktionen, sowie benutzerdefinierte Erweiterungen möglich.
SQL erlaubt also hoch dynamische Queries, transaktionsbasierten Zugriff und Integritätsbedingungen zur Konsistenzerhaltung.
Allerdings hat dies einen Nachteil: Es ist sehr formell und bietet schlechte „Abwärtskompatibilität“ mit simplen Datenmodellen. Es ist dort wenig *Tradeoff* zu Geschwindigkeit möglich. Ausserdem sind super-skalierende SQL-Datenbanken ein offenes Problem.
Bei Anwendungen in denen Konsistenz kein kritisches Kriterium ist, arbeiten RDBMS suboptimal: Kunden auf Amazon interessiert es nicht ob eine von 10 Buchempfehlungen nicht zutrifft oder nur 9 angezeigt werden - wenn der Seitenaufbau zu Stoßzeiten jedoch 3 Sekunden dauern würde, wäre das Einkaufserlebnis stark beeinflusst und der Kunde kauft sein Buch womöglich bei einem anderen Anbieter.
Die Mächtigkeit und Korrektheit des relationalen Modells bringt also gewisse Nachteile mit sich und ist auch nicht immer erwünscht. Gerade extrem große Anwendungen mit großen aber einfachen Daten brauchen eine bessere Lösung.

## 1.2 NoSQL

Aus dieser Konsequenz heraus hat sich in den letzten Jahren eine lebhafte und unkonventionelle Community von Entwicklern nicht-relationaler Datenbanken entwickelt[1].
Zum einen wird hier an Key Value Stores gearbeitet, andererseits zählen hier auch objekt-relationale oder NF2 wie BerkleyDB, O2, GemStone, Statice dazu.
Die Mitglieder dieser losen Gruppierung sind meist Mitarbeiter großer Unternehmen, die mit den oben genannten Problemen zu kämpfen haben und hauseigene Lösungen in der Open Source Community weiterentwickeln. Wie zum Beispiel Facebook, die ihre Datenbank *Cassandra*[2] nach einer mehrjährigen initialen internen Entwicklungsphase unter Apache 2 Lizenz veröffentlicht haben. Oder desweiteren *voldemort*[3], dass neben der Open Source Community hauptsächlich von LinkedIn entwickelt wird.

## 1.3 Key Value Stores

Key Value Stores (KVS) bezeichnet ein allgemeines Konzept von Datenbanken, deren Entitäten (Values) über einen eindeutigen Schlüssel (Key) indiziert werden. Damit sind KVS wohl der allgemeinste Fall einer Datenbank. Sie stellen eine simple Abbildung dar:

$$f(K) = V$$

[1] http://nosql-databases.org/
[2] http://incubator.apache.org/cassandra/
[3] http://project-voldemort.com/

Über die Struktur von V wird dabei keine Aussage gemacht, daher spricht man von schemalosen Daten. Struktierte Daten werden jedoch nicht ausschlossen. Realisiert wird die Indizierung meist mit einem B*-Baum. Wichtig ist hier, dass es keine sekundären Indizes, wie zum Beispiel Bitlisten-Indizes über Attribute, gibt und vorallem keine Fremdschlüsselintegrität gewahrt wird. Der Value kann (oft) von beliebigem nichtfestem Datentyp sein (auch Arrays, Dokumente, Objekte, Bytes, ...)
Das Weglassen von Konsistenzerhaltungsmechanismen (bzw. die Delegation an andere Schichten) versucht die Datenbank effizienter für Applikationen mit vielen aber einfachen Daten zu machen.
Da es nur einen Primärindex über den Schlüssel gibt, bietet die Abfragesprache eine viel geringere Komplexität. Das heißt, dass zunächst nur Anfragen nach einzelnen Keys oder Key-Ranges möglich sind. Mit dem MapReduce Konzept ermöglicht komplexere Anfragen z.B. mit Aggregatfunktionen realisiert und die verteilte Auswertung wird begünstigt[4]. Insgesamt fokussieren KVS primär auf Skalierbarkeit, Distribution (bzw. zumindest Synchronisation) und Fehlertoleranz.

## 1.4 Beispiel: Key-Value mit ruby + pstore

Das folgende Beispiel zeigt das allgemeine Konzept:

```
require "pstore"
store = PStore.new("data-file.pstore")
store.transaction do

  store[:item_1337] = {
    "name" => "BigMachine",
    "compatibleSoftware" => ["php", "mysql"]
  }

end

my_var = store[:item_1337]
```

Der verwendete Ruby-spezifische *pstore* ist ein nicht-verteilter Key-Value-Store mit *Active Record*-Funktionalität[5], der dafür allerdings Transaktionen unterstützt. Sprachinterne Konstrukte wie Strings und Arrays können transparent in die Datenbank geschrieben und wieder geladen werden. Der Zugriff erfolgt stets wie bei einem Array, mit einem alphanumerischen Schlüssel.

[4] siehe dazu die Seminararbeit von Thomas Findling und Thomas König. MapReduce wird daher im folgenden nicht weiter vertieft, ist jedoch sehr relevant im Zusammenhang mit KVS

[5] durch Überladen von gewissen Sprachelementen führt das Setzen eines Variablenwertes im Hintergrund zu Datenbankmanipulation.

## 1.5 CAP-Theorem

Das CAP-Theorem [2] ist eine von Dr. Eric A. Brewer aufgestellte Behauptung, die besagt, dass von drei möglichen Eigenschaften eines Datenbanksystems — Konsistenz, Verfügbarkeit und Partitionierbarkeit — stets nur zwei zugleich erreichbar sind. In Abbildung 1 existieren Schnittmengen zwischen 2 Eigenschaften, in der Mitte jedoch gibt es keinen Punkt in dem sich alle Eingeschaften überlappen.

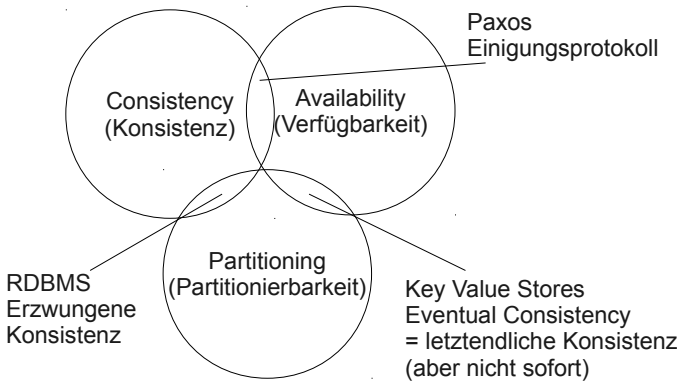


Abbildung 1: CAP-Theorem

Ein Beweis und eine gute Erläuterung ist in [3] zu finden. Die zentrale Konsequenz des Theorems ist folgende: Relationale Datenbanken sind konsistent und partitionierbar — jedoch auf Kosten der Verfügbarkeit. Ein partitioniertes System (also z.B. eine verteilte Datenbank wie HBase) kann jedoch nur noch entweder Konsistenz **oder** Verfügbarkeit erreichen. Wie angedeutet wird bei oben genannten KVS aufgrund des Einsatzgebiets zugunsten der Verfügbarkeit auf Kosten der Konsistenz entschieden. Brewer stellt dem ACID-Konzept von RDBMS ein BASE-Konzept für skalierende verteilte Anwendungen gegenüber. Konsistenz (und auch **I**solation) wird aufgegeben um Verfügbarkeit zu erreichen, wobei Mischformen möglich sind. BASE steht für **B**asically **A**vailable, **S**oft-state, **E**ventual consistency. Letzteres bedeutet, dass Konsistenz keinesfalls ausgeschlossen wird — im Gegenteil: *letztendlich* ist das System konsistent, nämlich dann wenn die Anwendung dies will und dafür sorgt.
Im folgenden möchte ich die beiden Konzepte gegenüberstellen:

**ACID**

- stark konsistent
- Verfügbarkeit wenn möglich
- pessimistisch: Locking
- mächtig
- komplizierte Schemaevolution

**BASE**

- konsistent wenn möglich
- hochverfügbar
- optimistisch: commit-Ablehung
- einfach
- schnell
- leichte Evolution

Als dritter Weg existiert noch Paxos — dies ist jedoch nach Definition nicht Partitionierbar und daher hier nicht relevant.

## 1.6 Viele Andere

In dieser Arbeit werden das BigTable-Konzept und dessen Implementierung HBase sowie das alternative CouchDB näher vorgestellt. Es sei jedoch auch auf die Seminararbeit von Simon Bin verwiesen, der inhaltlich an diese Arbeit anknüpft. Er stellt ergänzend Amazon Dynamo und Facebook Cassandra vor.
Neben diesen gibt es zahlreiche weiterer Systeme, die hier nicht behandelt wurden:

- Tokyo Cabinet (Datenbank für mixi.jp - ein japanischer facebook Klon, unter LGPL Lizenz)
- Redis (BSD-Lizenz)
- memcacheDB (BSD-Lizenz)
- MongoDB (AGPL Lizenz)
- voldemort (LinkedIn, Apache 2.0 Lizenz, Klon von Amazons Dynamo)
- ...

Ein sehr guter Überblick über Lizenzen und technische Merkmale wird in [4] gegeben.

# 2 Google BigTable

Google veröffentlichte 2006 Informationen [5] über die intern verwendete Datenbank, die hinter fast allen Diensten von Google steht.
Beschrieben wurde eine Architektur eines versionierten hoch-skalierenden Key-Value Stores und des Datenmodells. BigTable ist allerdings nicht direkt ein verteiltes System, denn es ist für einen lokalen Cluster konzipiert, in dem dann auch starke Konsistenz garantiert ist. 2009 wurden in [6] ergänzendende Informationen veröffentlicht. BigTable ist proprietär und Einzelheiten der Implementierung sind nur aus den Papern und Vorträgen bekannt. BigTable bezeichnet aber nicht nur das konkrete System, das von Google entwickelt wurde, sondern auch das dahinter stehende Konzept.

## 2.1 Google Requirements

Im Hintergrund von Google arbeiten gleichzeitig verschiedene asynchrone Prozesse (Crawler, Indexer usw.), die kontinuierlich den Datenbestand aktualisieren. Es liegen hohe read/write Raten von Millionen ops/sec vor. Regelmäßig werden Scans über die gesamten Daten bzw. Teile gemacht und Joins (mit MapReduce) müssen möglich sein. Die Veränderungen von Daten müssen im Zeitverlauf analysierbar sein.
Diese Anforderungen haben zur die Entwicklung von Google BigTable geführt. Dieses System ist hoch performant, hoch verfügbar und unterstützt komprimierte Datenhaltung. BigTable skaliert ausgesprochen gut auf Petabyte HDD-Daten und Terabyte RAM-Daten, welche auf tausenden Server zu Clustern zusammengefasst wurden. Diese Cluster benötigen ein intelligentes load-balance Selbstmanagement, das dafür sorgt, dass zum Beispiel häufig angefragte Bereiche mehr im *Cache* gehalten oder Aufgaben von ausgefallenen Servern automatisch verteilt werden.

## 2.2 Datenmodell

Das verwendete Datenmodell ist kein voll relationales Modell, obwohl es sich in gewissen Aspekten so verhält. Die Entwickler selbst bezeichnen es als „a sparse distributed multi-dimensional sorted map“. *Sparse* bedeutet in diesem Zusammenhang, dass verwendete Spalten (Columns) oft nur bei wenigen Datensätzen (Rows) genutzt werden und auch große Lücken zwischen den Keys existieren können. Das heißt, dass viele Nullwerte existieren. Dieses Problem wird über eine vertikale Partitionierung gelöst[6].
Die *multi-dimensional map* ist eine Abbildung

$$(row : string, column : string, time : int64) \rightarrow string$$

Oft wird *string* auch mit *byte[]* angegeben. Die Abbildung zeigt auch, dass alle Einträge versioniert sind. Die Form der Datenhaltung (ob möglichst im RAM oder nur HDD) ist Teil des Schemas, das somit Verfügbarkeitseigenschaften konfigurierbar macht.
Abbildung 2 zeigt die konzeptionelle Sicht auf eine Tabelle: Hier zu sehen ist ein Bei-

[6] Näheres dazu wird im Abschnitt 3.1 über HBase erklärt.

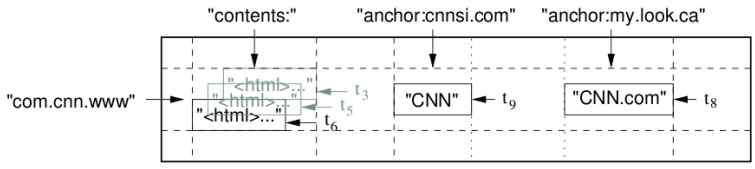


Abbildung 2: Eine Tabelle in BigTable

spiel, dass aus dem PageRank-Kontext stammt. Es werden in den Zeilen Datensätze zu Websites gespeichert. Als Spalten wird der HTML-Inhalt sowie über welche andere Seiten diese Websites verlinkt wurde gespeichert. Es liegt nahe, dass die Tabelle *sparse* seien wird, denn eine Seite wird gewöhnlich nur von wenigen anderen Seiten referenziert, sodass viele NULL-Werte (in der konzeptionellen Sicht) vorhanden sein werden.

## 2.3 Implementierung

Google setzt BigTable bei fast allen Services, die mit großen Daten umgehen müssen, ein. Als Beispiel ist da die Google-Suche, YouTube sowie Google Books zu nennen.
Da BigTable externen Nutzern ansonsten nur indirekt über die Google App Engine zugänglich ist, gibt es Open-Source Implementierungen, die genau dieses Konzept umsetzen, dabei aber teilweise andere Wege der konkreten Umsetzung gehen. Zum Beispiel:

- Cassandra
- HBase
- Hypertable

HBase ist ein sehr naher Klon, der nahezu alle bekannten Architekturmerkmale von BigTable umsetzt und auf das selbe Konzept aufbaut. Alle in HBase genutzten Techniken treffen ebenso für BigTable zu, besitzen dort aber oft andere Bezeichnungen. Daher werde ich nach diesem Einblick in das Konzept BigTable nicht näher auf die Implementierung von BigTable eingehen, da ich dies indirekt im folgenden Abschnitt über HBase tun werde.

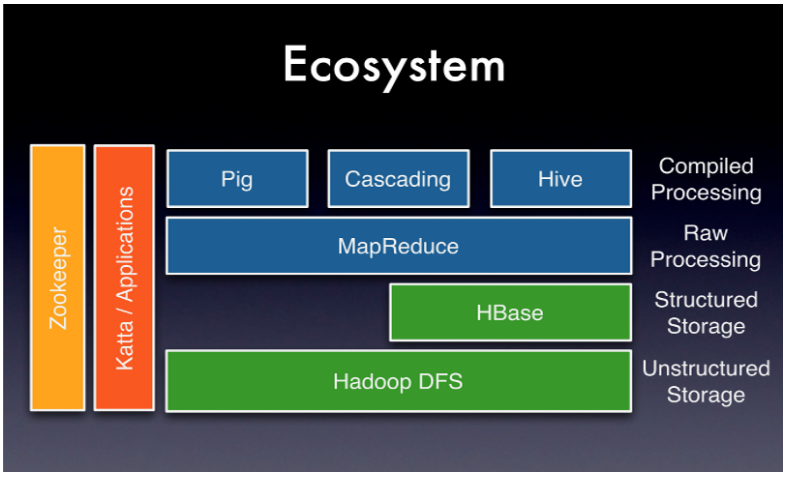


Abbildung 3: Das Hadoop Ecosystem

# 3 HBase

HBase ist die Datenbank von Hadoop[7]. HBase ist dabei fest integriert in das sogenannte „Ecosystem“, das aus den Hadoop-Komponenten besteht (siehe Abbildung 3). HBase ist eine in Java geschriebene Open-Source Implementierung des BigTable-Konzepts mit dem Ziel, Tabellen mit Milliarden Rows, X Tausend Columns und X tausend Versionen performant zu unterstützen. Dieses Ziel wurde bisher noch nicht vollständig erreicht. Das System wird zur Zeit primär und sehr aktiv von den Firmen StumbleUpon, WorldLingo und Powerset entwickelt.

## 3.1 Datenmodell – Column Families

Analog zu BigTable werden die Tabellen intern in sogenannte Column Families zerlegt: zusammengehörige Spalten werden isoliert verwaltet. Verschiedene Anwendungen greifen jeweils nur auf eine Teilmenge, der Columns zu. Diese Anwendungen können unterschiedliche Nutzungsprofile haben und so lassen sich Konfigurationen, wie Versionierung oder Kompression für jede Column Family einzeln definieren. Weitere konfigurierbare Eigenschaften sind Bloomfilter, Art der bevorzugten Speicherung (HDD oder RAM) und Länge der Felder. Diese vertikale Partitionierung hat auch den Vorteil, dass ungebundene Attribute einer Entität nicht explizit gespeichert werden müssen; die Zeile wird dann in der Column Family weggelassen.
Die Sortierung der Einträge erfolgt, alphanumerisch nach dem Key, dies ermöglicht Range-Queries.

In Abbildung 4 ist die vertikale Partitionierung nach Column Families dargestellt.

[7] ein „Framework für hochverfügbare, skalierende, verteilten Berechnungen“ (siehe http://hadoop.apache.org/)

Konzeptionell

| Row Key | Time Stamp | Column "contents:" | Column "anchor:" | | Column "mime:" |
|---|---|---|---|---|---|
| "com.cnn.www" | t9 | "<html>abc..." | "anchor: cnnsi.com" | "CNN" | |
| | t8 | "<html>def..." | "anchor: my.look.ca" | "CNN.com" | |
| | t6 | "<html>ghi..." | | | "text/html" |

Intern

| Row Key | Time Stamp | Column "contents:" |
|---|---|---|
| "com.cnn.www" | t9 | "<html>abc..." |
| | t8 | "<html>def..." |
| | t6 | "<html>ghi..." |

| Row Key | Time Stamp | Column "anchor:" |
|---|---|---|
| "com.cnn.www" | t9 | "anchor: cnnsi.com" |
| | t8 | "anchor: my.look.ca" |

...

Abbildung 4: konzeptionelle vs. interne Sicht

Diese Partitionen werden dann einzeln gespeichert, denn auf mehrere Column Families gleichzeitig wird (nach Definition) nicht in einer Anfrage zugegriffen.

## 3.2 Zugriff

Es gibt eine RESTful-API, die den Zugriff über das HTTP-Protokoll ermöglicht[8], die RPC-API Thrift, die Bibliotheken für verschiedenste Programmiersprachen enthält, sowie eine Shell-Konsole, die lokale Konfiguration ermöglicht.
Die Benutzung über die Shell

```
${HBASE_HOME}/bin/start-hbase.sh
${HBASE_HOME}/bin/hbase shell
hbase> create "mylittletable", "mylittlecolumnfamily"
hbase> describe "mylittletable"
hbase> put "mylittletable", "x"
hbase> get "mylittletable", "x"
hbase> # get "mylittletable", "x", {COLUMN => 'c1', TIMESTAMP => ts1}
hbase> scan "mylittletable"
```

## 3.3 Architecture Design

Das Design ist sehr gut im eigenen Wiki[9] sowie in [7] erklärt und realisiert verteilte aber zentral verwaltete Datenhaltung.

[8] siehe dazu auch Abschnitt 4.2
[9] http://wiki.apache.org/hadoop/Hbase/DesignOverview

Die Entwickler bezeichnen HBase als Multiple Client – Multiple Server Architektur. Dies wird durch einen Cluster realisiert, der aus *einem* Master und n RegionServern besteht sowie beliebig vielen Clients, die Anfragen stellen. Der Master-Server enthält die Metadaten, die nötig sind um Datensätz zu finden, nicht direkt (um ihn nicht mit hohen Durchsatzraten zu belasten), allerdings ist er dennoch ein *Single Point of Failure*: Wenn er ausfällt ist (bis auf caches) unbekannt *wo* die Metadaten liegen. Es gibt jedoch aktuelle Entwicklungen, die eine Multi-Master Erweiterung anstreben.
Wie bei BigTable erwähnt unterstützt HBase mit dieser zentralistischen Architektur eine starke Konsistenz, erst bei der Replikation über mehrere entfernte Cluster greift das CAP-Theorem.

### 3.3.1 Regions (Row Ranges)

Konzeptionell ist eine Tabelle eine Liste von Tupeln (Row), sortiert nach Row Key aufsteigend, Column Name aufsteigend und Timestamp absteigend. Intern ist eine Tabelle in Column Families geteilt (s.o.). Diese werden wiederum in Row Ranges, sogenannte Regions, geteilt. Jeder Region enthält Rows vom start-key ($\in$) bis end-key ($\notin$), bildet also eine horizontale Partitionierung. Regions sind (per Default) 256MB groß und werden redundant gespeichert. Regions können sequentiell mit einem Iterator (Scanner) durchlaufen werden.

### 3.3.2 RegionServer

Diese Regions werden von sogenannten RegionServern verwaltet. Ein RegionServer enthält mehrere Regions und nimmt Anfragen an diese Regions entgegen. Bei Schreibzugriffen schreibt er zuerst in einen write-ahead Log, der als Buffer dient. Bei Lesezugriffen ließt er zunächst im write-ahead Log und fungiert bei einem Treffer als Cache. Die physische Speicherung läuft über sogenannte StoreFiles ab und somit über HDFS, das verteilte Dateisystem von Hadoop. Bei Compactions (Komprimierungen) wird zwischen Minor und Major Compactions unterschieden. Ersteres bezeichnet die Konsolidierung der x zuletzt genutzten StoreFiles. Diese werden zusammengefasst und komprimiert. Major Compactions werden je nach Konfiguration seltener vorgenommen, da sie sehr I/O intensiv sind. Alle StoreFiles werden dabei zusammengefasst. Die RegionServer unterstützen außerdem gesondertes Caching für häufig nachgefragte Regions.

### 3.3.3 Master, ROOT und META

Der Master ist erster Anlaufpunkt bei der Suche und daher besonders leichtgewichtig indem seine Funktion nur das Verteilen von Regions auf die RegionServer und das Verwalten des Schemas ist. Ausserdem registriert er Ausfälle von RegionServern wenn sogenannte *Heartbeat-Messages* ausbleiben und kompensiert dies durch Wiederherstellung der benötigten Anzahl an Replikaten deren Regions.
Der Zugriff auf Regions verläuft hierarchisch: Auf welchem RegionServer eine Region liegt wird ebenfalls mit einer HBase Tabelle verwaltet: der sogenannten META Table.

Diese kann sehr groß werden und wird wiederum in Regions zerlegt, welche auf RegionServern verwaltet werden. Um die Teile der META Table zu finden gibt es die ROOT Region, die die Adressen der RegionServer der META Regions enthält. Die Root Region wird wie eine reguläre Region von einem RegionServer verwaltet. Die Addresse dieses Servers ist allerdings direkt im Master gespeichtert. Wenn ein Eintrag gesucht wird, wird zunächst die ROOT Region nach dem RegionServer der entsprechenden META Region befragt. In dieser Region findet sich dann die Adresse des RegionServer, der die eigentlich gesuchte Region hält.
Der Zusammenhang zwischen Master, ROOT, META und den RegionServern wird in Abbildung 5 schematisch dargestellt.

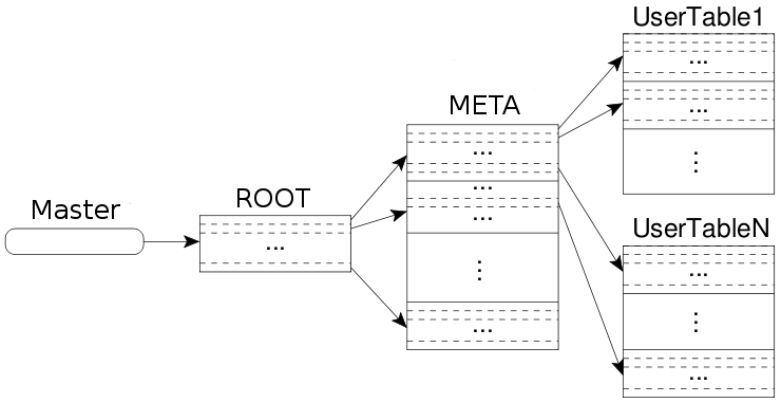


Abbildung 5: Die HBase Architektur

### 3.3.4 Client

Das Vorgehen eines HBase-Clients, um einen Eintrag zu finden, gestaltet sich wie folgt: er fragt den Master nach der ROOT Region, scannt die ROOT Region nach dem Ort der META Region, scannt die META Region nach dem RegionServer für die gesuchte Region, kontaktiert den RegionServer und scannt die gesuchte Region. Dabei cacht er möglichst die Suchgebnisse für zukünftige Anfragen.

### 3.3.5 Region Splits

Wenn eine Region durch Einfügeoperationen eine konfigurierte Größe überschreitet, muss sie geteilt werden:

$$parent \rightarrow \{child1[start, mitte], child2[mitte, ende]\}$$

Dazu wird folgende Prozedur verwendet:

1. setze parent offline
2. registriere children in Meta Table ohne RegionServer

3. informiere Master. Dieser weist RegionServer zu
4. komprimiere children
5. setze children online
6. parent in Garbage Collector

## 3.4 Versioning

Die Versionierung ist sehr vorteilhaft realisiert: Den genauen Timestamp eines Eintrags muss man nicht kennen um ihn auszuwählen. Man gibt den Zeitpunkt an, von dem man den Zustand der Entität sucht und dann ermittelt das System den passenden Timestamp (indem es den *größten kleineren* Timestamp sucht — denn dieser ist der aktuellste zum gesuchten Zeitpunkt). Ausserdem ist ein Versions-Fenster konfigurierbar: Wenn Versionen zu alt werden oder zu viele existieren, werden sie automatisch gelöscht.

## 3.5 Einschränkungen HBase

Da HBase keine relationale Datenbank ist, ergeben sich einige Einschränkungen, die oft eine neue Herangehensweise an Probleme verlangen:

- keine Joins
- keine hochentwickelte Query Engine
- keine Datentypen für Columns
- keine eingebaute Warehouse-Funktionalität (dafür Hive)
- (noch) keine Transaktionen („atomic per row“ist implementiert - weiteres ist in Entwicklung)

Damit ist HBase kein 1:1 Ersatz für RDBS: das heißt es ist für viele Zwecke unter Umständen nicht geeignet: wie oben erwähnt wird Konsistenz gegen Verfügbarkeit getauscht und es gibt selbstverständlich viele Anwendungsfälle, bei denen es nicht nötig ist Hoch-Performant zu sein. Wenn Daten sehr homogen und einfach strukturiert sind ist das Datenmodell BigTable nicht angebracht. Auch muss bei einer Neueinführung eines nicht-relationalen Systems erheblicher Modellierungsaufwand betrieben werden und Anwendungen aktiv Konsistenz herstellen, denn dies ist nicht mehr im DBMS garantiert.
Das System befindet sich noch in der Entwicklungsphase:Zum Zeitpunkt dieser Arbeit liegt Version 0.20.2 vor, diese ist allerdings nicht als Beta gekennzeichnet. Der Code an sich ist großteils stabil und wird bereits bei verschiedenen Projekten und Unternehmen produktiv eingesetzt. Es gibt allerdings noch API-Changes und kompatibilitätsbrechende Änderungen, die eine Migration aller Daten nach sich ziehen und die Versions-History entfernen, sind nicht ausgeschlossen.

# 4 CouchDB

## 4.1 Einführung

CouchDB zielt in Gegensatz zu HBase nicht auf riesige Datenmengen ab. Vielmehr wird hier ein Ansatz verfolgt, der es ermöglicht kleine Datenmengen effizient zu verwalten und eine einfache Integration mit simplen Applikationen zu erreichen.

> „Apache CouchDB is a document-oriented database that can be queried and indexed in a MapReduce fashion using JavaScript. CouchDB also offers incremental replication with bi-directional conflict detection and resolution.“ [8]

In CouchDB sind Dokumente JSON-Objekte[10], das heißt eine Folge von Name/Wert Paaren mit beliebig genesteter Struktur. Die möglichen Datentypen sind JavaScript Primitive wie string, int, float, array und so weiter. Ein Dokument kann binary file „attachments“ enthalten. Die Versionsnummer ist Metainformation jedes Dokuments.

Ein Beispiel für ein CouchDB Dokument (Datensatz) in JSON Notation:

```
{
    "_id": "2DA92AAA628CB4156134F36927CF4876",
    "_rev": "75AA3DA9",
    "type": "contact",
    "firstname": "Smith",
    "lastname": "John",
    "picture": "http://example.com/john.jpg",
    "current_cart": [
        {
            "aid": 45456,
            "amount": 2
        },
        {
            "aid": 66345,
            "amount": "1"
        }
    ]
}
```

Dabei sind die mit _ beginnenden Felder zu beachten: Diese sind reservierte Felder, die systemintern genutzt werden. Das Feld `_id` enthält eine eindeutige Identifikationsnummer. Im Beispiel ist sie systemgeneriert und hexadezimal — es ist allerdings auch möglich beim Anlegen des Dokuments einen beliebigen String anzugeben. Die Revisionsnummer `_rev` wird vom System verwaltet und identifiziert den Änderungsstand eines Dokuments. Wenn man ein Dokument anfordert ist dieses Feld optional. Wird die Revision nicht explizit angegeben erhält der Nutzer die aktuelle Revision.

[10] http://json.org

## 4.2 Web-basierter Zugriff

Der Zugriff erfolgt über eine HTTP-basierte API, die das RESTful-Konzept unterstützt (siehe dazu [9] und [10]). Das bedeutet, dass der Nutzer auf Dokumente in der Datenbank über eine URL zugreifen kann. Das folgende Beispiel zeigt, wie der Nutzer (z.B. über AJAX-Requests in einer Web-Applikation) neue Dokumente (also Datensätze) anlegen kann:

```
PUT /somedatabase/1234 HTTP/1.0
Content-Length: 161
Content-Type: application/json

{
 "content":"xyz"
}

HTTP/1.1 201 Created
Date: Thu, 7 Jan 2010 05:39:28 +0000GMT
Content-Type: application/json
Connection: close

{
 "ok": true,
 "id": "1234",
 "rev": "946B7D1C"
}
```

Auf ähnliche Weise lassen sich die Daten wieder über einen GET-Request auf `/db/1234` abrufen.

```
GET /db/1234 HTTP/1.1

HTTP/1.1 200 OK
Date: Thu, 17 Aug 2009 15:39:28 +0000GMT
Content-Type: application/json
Connection: close

{
 "_id":"1234",
 "_rev":"946B7D1C",
 "content":"xyz",
}
```

Diese Zugriffsweise verschafft CouchDB einige interessante Eigenschaften: Da es sich wie ein normaler Webserver verhält, können hier auch gängige Techniken wie HTTP

Authentifikation, Caching oder load-balancing über Proxies genutzt werden, was den Entwicklungsaufwand verringert.

## 4.3 Konflikte

Im Gegensatz zum cluster-orientierten HBase ist CouchDB eine einzelne lokale Instanz. Um Verteiltheit zu erreichen wird hier ein ganz anderes Konzept genutzt: Replikation — also die Synchronisation zwischen zwei Instanzen.
Eine CouchDB-Instanz ist dabei *offline by default*. Dies bedeutet dass CouchDB prinzipiell kein verteiltes System ist, denn eine einzelne Instanz hat nur lokalen Speicher. Es ist allerdings vorgesehen mehrere Instanzen zu einem Quasi-Cluster verknüpfen, in dem Instanzen nur paarweise und nur in eine Richtung synchronisieren. Diese einzelne Verbindung kann als Baustein für komplexere Replikationsschemata dienen. Desweiteren wird nicht von einer permanenten Verbindung zwischen den Instanzen ausgegangen. Beispielsweise eine Kontaktdatenbank, die auf mobile Endgeräte synchronisiert wird. Das Replikationsschema könnte rein unidirektional oder komplett vernetzt sein — je nach Anwendungsfall konfigurierbar. Die Geräte synchronisieren sich sobald sie Netzzugang haben. In einem solchen verteilten System existieren stets mehrere Realitäten gleichzeitig. Konflikte bei Synchronisation dieser Realitäten werden bei CouchDB als „gewöhnlich“ angesehen — sie stellen keine Ausnahme dar und sind legaler Zustand des Systems. In CouchDB existiert dafür eine besondere Markierung (`"_conflicts": true`). Entstandene Konflikte werden nicht vom System aufgelöst, sondern es wird deterministisch eine gewinnende Revision gewählt. Die verlierende Revision bleibt aber (mit Markierung, in allen Replikas) erhalten und wird im regulären Betrieb nicht mehr angezeigt. Die Auflösung des Konflikts ist der Anwendung überlassen (bzw. dem User). Konflikte treten nur bei der Synchronisierung zwischen CouchDB-Nodes in einem Cluster auf.
Bei lokalen Änderungen werden Inkonsistenzen verhindert; allerdings nicht durch *locking* sondern mit sogenannten *optimistic commits*. Dies bedeutet, dass bei commits zunächst davon ausgegangen wird, dass alles gutgehen wird und Datensätze werden nicht gesperrt. Um trotzdem die Konsistenz zu sichern übermittelt man beim commit die Revisionsnummer vom letzten Lesen. Wenn diese veraltet ist, wird der commit abgelehnt. Dieses Verfahren wird beispielsweise auch bei SVN eingesetzt. Wiederholtes commiten kann zwar nötig sein, dafür muss aber nichts gesperrt werden, was die Verfügbarkeit erhöht. Dieser gesamte Zusammenhang wird mit Multiversion Concurrency Control bezeichnet.

## 4.4 Einsatz

CouchDB hat ein vergleichsweise einfaches Konzept und ist damit für „kleine“ Anwendungen besonders gut geeignet. Zum Beispiel Websites (bbc.co.uk und meebo.com) nutzen CouchDB Cluster als Backend. Auch bei Desktop-Anwendungen wird CouchDB eingesetzt: desktopcouch synchronisiert Notizen, Kontakte, Kalender und Bookmarks zwischen mehreren Rechnern eines Users. UbuntuOne bietet Cloud-Speicher auf Couch-

DB-Basis. Ein zukünftiges Einsatzgebiet sollen auch Handys sein. Hier soll vorallem das „offline by default“-Konzept und das leichtgewichtige Design Vorteile bringen. CouchDB ist „build of the web“, denn es nutzt bereits bestehende Webtechnologien wie HTTP, JSON und JavaScript (für Map-Reduce-Funktionen) und ist damit sehr gut gerüstet für zukünftige Entwicklungen.

# 5 Zusammenfassung

Das Key-Value-Konzept erweist sich als flexible Grundlage für verschiedenste Systeme, die alternative Lösungen zum relationalen Modell in Hinsicht auf gute Skalierbarkeit und Performanz bieten. Die vorgestellten Projekte sind sehr unterschiedlicher Natur:
BigTable ist propritär und hier vorallem als theoretische Grundlage für große Cluster-Lösungen für Datenbanken mit heterogen und sehr umfangreichen Datenbeständen betrachtet.
HBase ist eine freie Implementierung des Konzepts und schon sehr leistungsfähig und entwicklerfreundlich, sodass die Zukunft sehr vielversprechend ist.
CouchDB baut auf ausgereifte und fortschrittliche Webtechnologien auf und versucht für kleine und mittlere Projekte eine einfache und schnelle Dokumentendatenbank zu sein, die vielfältige Synchronisationsschemata ermöglicht.
Key-Value-Stores bieten somit Alternativen zu RDBMS für einerseits sehr große sowie für eher kleine Daten, die nicht dem relationalen Modell genügen (müssen) und verschafft sich durch neue Architekturen entscheidende Vorteile, die von zukünftigen Anwendungen dringend benötigt werden um den neuen Anforderungen gerecht zu werden.

# A Literaturverzeichnis

## Literatur

[1] Mark Seeger. Key Value Stores – a practical overview. http://blog.marc-seeger.de/assets/papers/Ultra_Large_Sites_SS09-Seeger_Key_Value_Stores.pdf zuletzt überprüft: 07.01.2010, 2009.

[2] Eric A. Brewer. Towards Robust Distributed Systems. www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf zuletzt überprüft: 07.01.2010. Invited Talk at PODC 2000.

[3] Julian Browne. Brewer's CAP Theorem. http://www.julianbrowne.com/article/viewer/brewers-cap-theorem, Januar 2009.

[4] Richard Jones. Anti-RDBMS: A list of distributed key-value stores
. http://www.metabrew.com/article/anti-rdbms-a-list-of-distributed-key-valu
zuletzt überprüft: 07.01.2010, 2009.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, 2006.

[6] Jeffrey Dean. Designs, Lessons and Advice from Building Large Distributed Systems. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*, 2009.

[7] Ankur Khetrapal and Vinay Ganesh. HBase and Hypertable for large scale distributed storage systems. Technical report, Dept. of Computer Science, Purdue University, 2009.

[8] *CouchDB: The Definitive Guide, 1st Edition*. J. Chris Anderson and Jan Lehnardt and Noah Slater, 2009.

[9] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.

[10] Alex Rodriguez. RESTful Web services: The basics. http://www.ibm.com/developerworks/webservices/library/ws-restful/ zuletzt überprüft: 07.01.2010, 2008.