

Masterstudiengang Wirtschaftsinformatik

Big Data Engineering

FH Münster
Master Wirtschaftsinformatik
Wintersemester 2015
Dozent: Lars George



Einheit 5

- Rückblick auf Einheit 4
- NoSQL „Datenbanken“ im Überblick
 - Verschiedene Arten und deren Konzepte
 - Unterschiede zu relationalen Datenbanken
- HBase als Beispiel
- **Hauptziel:** Konzepte der NoSQL Speichersysteme kennenlernen und verstehen
- **Übung 5:**
 - HBase zur Speicherung von Daten verwenden



Einheit 5

- **Rückblick auf Einheit 4**
- NoSQL „Datenbanken“ im Überblick
 - Grundlagen
 - Gründe für NoSQL
 - Zugrundeliegende Annahmen
 - Entwurfsmuster
 - Fragenkatalog
- HBase als Beispiel



Rückblick auf Einheit 4

- Fragen?
 - Hive oder Pig ausprobiert?
 - Verschiedene Ausgabeformate verglichen?
 - Bücher mit Solr indiziert?
- Übung 3+4: TF-IDF?
 - Wie kann man das Endergebnis nutzen?
 - Vergleich mit Solr oben?



Rückblick auf Einheit 4

Hinweis:

Das Code Repository enthält einen **Jetty** basierten Server mit **REST API** und einfacher **JSP** Oberfläche. Dieser kann auch weiter benutzt werden für die Projektarbeit!

Das Maven Projekt hat **einige** interessante **Plugins** integriert, um sowohl den Jetty Server, als auch die MapReduce JAR zu erstellen.



Rückblick auf Einheit 4

Search Portal ≡

🔍

Found 17 documents in 0.0080 seconds.

Alice's Adventures in Wonderland

Author: Lewis Carroll
Release: March, 1994
Score 0.000036 | ID: 11

Leaves of Grass

Author: Walt Whitman
Release: August 24, 2008
Score 0.000029 | ID: 1322

Moby Dick; or The Whale

Author: Herman Melville
Release: January 3, 2009
Score 0.000029 | ID: 2701



Einheit 5

- Rückblick auf Einheit 4
- **NoSQL „Datenbanken“ im Überblick**
 - Grundlagen
 - Gründe für NoSQL
 - Zugrundeliegende Annahmen
 - Entwurfsmuster
 - Fragenkatalog
- HBase als Beispiel



Zuerst einige SQL Grundlagen

Die **Grundlage** für die bekannten, **SQL** basierten Datenbanken wurde 1969 von **Edgar F. Codd** formuliert. Diese **basiert** auf der Spezifikation des **relationalen** Datenmodels und den **Codd's 12 Rules**, welche 13 (null basiert!) Regeln für relationale Datenbanksysteme festlegen.

Es gibt auch noch **andere**, aber **weniger** populäre Modelle, wie Netzwerk-, hierarchische, oder objektorientiert Modelle.



Relationales Datenmodell

In diesem Modell werden Daten als **Tuples** repräsentiert und in **Relations** gruppiert (stammt aus der „Prädikatenlogik erster Stufe“). Eine relationale **Datenbank** implementiert dieses Modell. Es bietet eine **deklarative** Methode Daten zu **spezifizieren** und abzufragen.

Relational model

From Wikipedia, the free encyclopedia

The **relational model** for **database management** is a **database model** based on **first-order predicate logic**, first formulated and proposed in 1969 by Edgar F. Codd.^{[1][2]} In the relational model of a database, all data is represented in terms of **tuples**, grouped into **relations**. A database organized in terms of the relational model is a **relational database**.



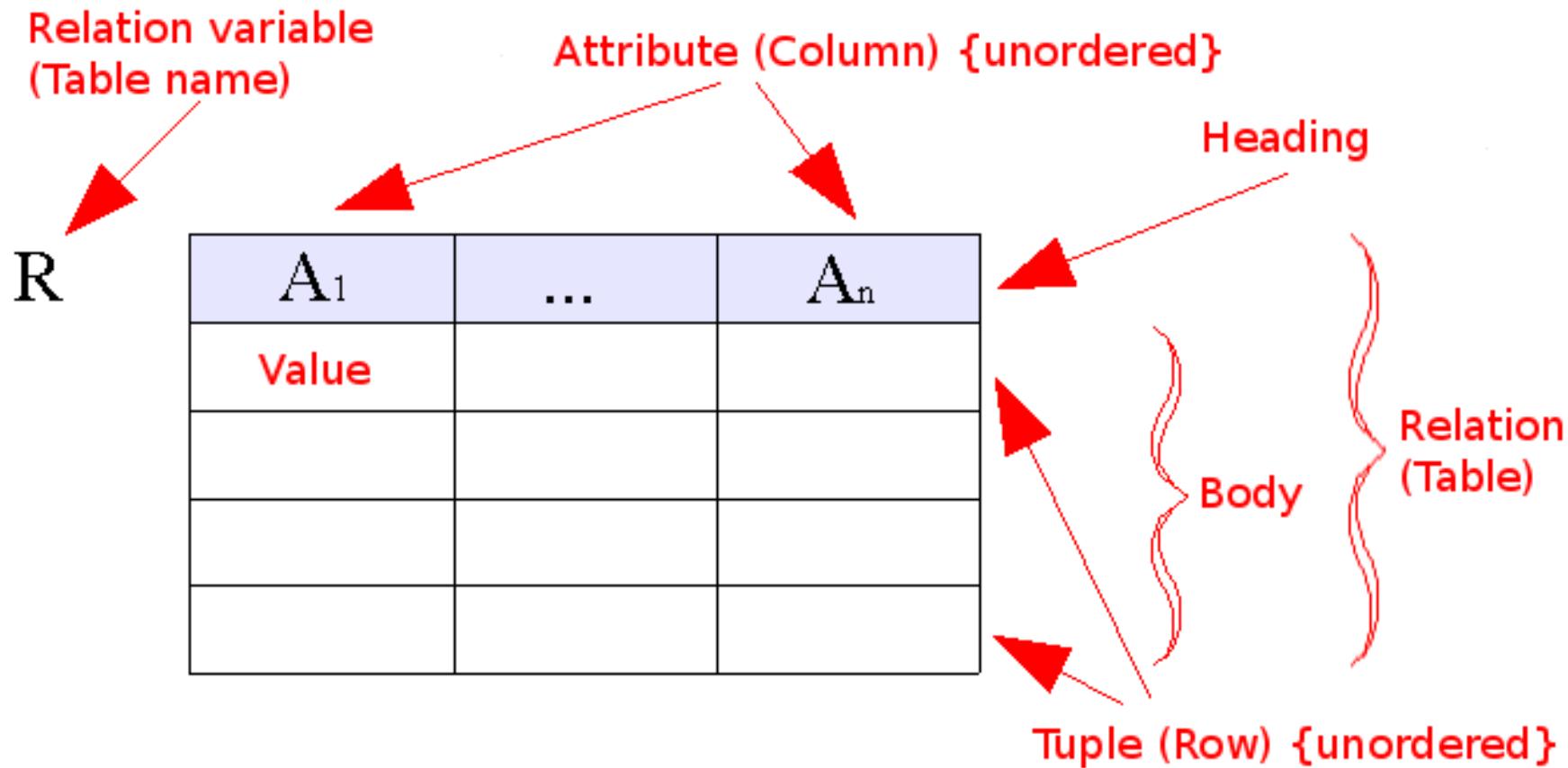
Relationales Datenmodell

Benutzer eines **solchen** Datenmodells kennen **keine** Implementierungsdetails, sondern **beschreiben**, welche **Daten** es gibt und welche sie für von einer **Abfrage** erwarten.

Wie die Daten **organisiert** oder **abgelegt** sind ist die Aufgabe des Datenbank Management Systems (**DBMS**). Dieses legt Speicher- und Abfragestrukturen wie benötigt **intern** fest.

Konzepte

Folgende Konzepte existieren im relationalen Model:



Mengenlehre

Die Mengenlehre definiert die Grundbegriffe, welche im relationalen Model benutzt werden.

Begriff	Bedeutung
Tuple	Repräsentiert die Zeilenwerte
Header	Name aller Spalten
Projektion	Eine Abbildung eines Tupels auf ein anderes
Relation	Ein Tupel mit Header und Body, welches aus den Wertetupeln entsprechend dem Header besteht

SQL – Eine Annäherung



Typisch für relationale Datenbanksysteme (RDBMS) ist die **Benutzung** von **SQL** (Structured Query Language) – welche eine **technische** Annäherung an das **relationale** Model darstellt.

Eine Tabelle (**Table**) ist ein **Prädikatvariable**, dessen **Inhalt** eine Relation (**Relations**). Wir kennen diese Nomenklatur bereits von Pig aus der **letzten** Einheit.

SQL – Eine Annäherung



Key Constraints (Zwangsbedingungen des Schlüssels) und **andere** Bedingungen, sowie die **eigentliche** SQL Abfrage entsprechen den **Prädikaten**.

SQL hat aber **einige** Freiheiten, welche laut dem Model **nicht** erlaubt sind, z. B. Zeilen mit **demselben** Inhalt (Bag vs. Set), Spaltennamen mit **gleichem** Namen, aber auch der bekannte **NULL „Wert“**.

Beispiel #1



Relational Model

Activity Code	Activity Name
23	Patching
24	Overlay
25	Crack Sealing

Key = 24

Activity Code	Date	Route No.
24	01/12/01	I-95
24	02/08/01	I-66

Date	Activity Code	Route No.
01/12/01	24	I-95
01/15/01	23	I-495
02/08/01	24	I-66



Beispiel #2

Dieses Beispiel bildet ein **Bestellsystem** ab. Zu beachten ist, dass es **nicht** vollständig **normalisiert** ist.

Relationsvariable	Attribute
Customer	<u>Customer ID</u> , Tax ID, Name, Address, City, State, Zip, Phone, Email
Order	<u>Order No</u> , <u>Customer ID</u> , <u>Invoice No</u> , Date Placed, Date Promised, Terms, Status
Order Line	<u>Order No</u> , <u>Order Line No</u> , <u>Product Code</u> , Qty
Invoice	<u>Invoice No</u> , <u>Customer ID</u> , <u>Order No</u> , Date, Status
Invoice Line	<u>Invoice No</u> , <u>Invoice Line No</u> , <u>Product Code</u> , Qty Shipped
Product	Product Code , Product Description

Legende: Fett+unterstrichen – Hauptschlüssel; Nur unterstrichen - Fremdschlüssel



Codd's Regeln

Neben dem **relationalen** Model hat Codd auch definiert, welche **Bedingungen** ein DBMS **erfüllen** muss, um als **relational** zu gelten. Dieses sind **13** Regeln (0-12).

Leider sind die Regeln so **abstrakt**, dass in der Praxis so gut wie **keine** Datenbank danach als **relational** bezeichnet werden kann.



Codd's Regeln

Regel 0: Eine relationale Datenbank wird von einem Datenbank Management System verwaltet (DBMS).

Dieses System soll in der Lage sein die Datenbank vollständig über die Relationen der einzelnen Elemente zu verwalten.

Regel 1: Informationen (und wie sie dargestellt werden)

Jede Information die in einer relationalen Datenbank abgespeichert wird, wird als ein WERT in Tabellen dargestellt. Dies betrifft auch die Namen von Spalten und Namen von Tabellen.

Regel 2: Garantierter Zugriff

Auf die Daten in der relationalen Datenbank muss garantiert über eine Kombination aus Primärschlüssel, Spalte und Tabelle zugegriffen werden können.



Codd's Regeln

Regel 3: Null-Werte

Die Null Werte werden systematisch behandelt. Das bedeutet, dass Nullwerte das Gegenteil von „normalen“ Werten darstellen. Befindet sich ein Eintrag in der Tabelle einer relationalen Datenbank, so ist das ein normaler Wert. Wenn allerdings ein Eintrag fehlt, oder unbekannt ist, so ist das ein Null-Wert.

Regel 4: Katalogisierter Relationen-Struktur

Alle Daten, (auch die Datenbank selbst) wird in Tabellen gespeichert. Die Daten, die Datenbank und Tabellen werden alle auf einer Systemkatalog Ebene gespeichert. Dadurch wird die relationale Datenbank über eine Datenbanksprache abgefragt werden können.

Regel 5: Verständliche Sprache und Sprachenregeln

Jede relationale Datenbank wird mit (mindestens einer) Datenbanksprache verwaltet. Diese wird genutzt um in der Datenbank die Daten zu manipulieren, definieren, Regeln setzen und Rechte vergeben.



Codd's Regeln

Regel 6: Aktualisierung und Sichten

Sämtliche Sichten in einer Datenbank können von System aus aktualisiert werden und bedürfen hierfür keine spezielle Programme.

Regel 7: Bearbeitung von Tabellen

Eine relationale Datenbank kann nicht nur abgefragt werden. Das Management System soll auch Operationen wie: Anlegen von Tabellen und Spalten, Löschen und Ändern, ermöglichen.

Regel 8: Physikalische Datenunabhängigkeit

Der Zugriff auf die Daten mit Hilfe von Anwendungen und Programmen muss unabhängig vom physikalischen Zugriff und Strukturen der Speicherung erfolgen.



Codd's Regeln

Regel 9: Logische Datenunabhängigkeit

Wenn die Struktur einer Tabelle in der Datenbank verändert wird, so darf das keinen Einfluss auf die Programme haben, die auf eine relationale Datenbank zugreifen.

Regel 10: Integrität

Die Regeln der Integrität (Integritätsregeln) müssen im Systemkatalog abgespeichert werden. Daher muss man sie in einer Sprache definieren können. Die Regeln sollen so konstruiert sein, dass man sie nicht umgehen kann und zwingen mit ihnen arbeiten muss.

Regel 11: Verteilung

Geht ein Programm, das die Datenbank nutzt, von einer nicht verteilten und verteilte Datenbank über, so darf der logische Zugriff nicht abgeändert werden.



Codd's Regeln

Regel 12: Low Level Regel (Umgehung durch Sub-Sprache)

Die definierte Integritätsregeln dürfen nicht mit Hilfe einer Sub-Sprache umgangen werden. Diese Regeln werden mit Hilfe der Datenbanksprache definiert und nur durch diese Sprache dürfen sie geändert oder aktualisiert werden.



Transaktionen

Eine weitere **wichtige** Komponente eines (R)DBMS sind dessen **Eigenschaften** bei der Verarbeitung von Daten, speziell wenn diese **mehrere** Schritte umfasst.

Diese Eigenschaften werden typischerweise mit **ACID** umschrieben und bedeuten:

- A – **Atomicity**
- C – **Consistency**
- I – **Isolation**
- D – **Durability**



Transaktionen

Atomarität (Abgeschlossenheit)

Eine Sequenz von Datenoperationen wird entweder ganz oder gar nicht als durchgeführt betrachtet.

Konsistenzerhaltung

Wenn die Datenoperationen abgeschlossen sind, befindet sich die Datenbank wieder in einem konsistenten Zustand (wenn es vorher auch schon war).



Transaktionen

Isolation (Abgrenzung)

Nebenläufige (zeitgleiche) Datenoperationen beeinflussen sich nicht. Transaktionale Isolationsgrade bestimmen den Grad.

Dauerhaftigkeit

Nach Abschluss der Operationen sind deren Änderungen permanent in der Datenbank gespeichert. Dies wird meistens mit einem Transaktionslog abgebildet.



Konsistenzerhaltung

Im **speziellen** ist die Konsistenzerhaltung für die **Normalisierung** des Datenbestandes verantwortlich, denn sie erlaubt **Redundanzen** gering zu halten.

Beziehungen werden durch **Fremdschlüssel** und **Eigenschaften** abgebildet. Die Konsistenz bezieht sich deshalb auf **Inhalt** und **referentielle Integrität**.

Dies **erlaubt** Änderungen über **mehrere** Tabellen hinweg.



Konsistenzerhaltung

In verteilten Datenbanken kommt es zu Problemen, wenn alle ACID-Eigenschaften erfüllt werden sollen. Diese Probleme wurden in dem CAP-Theorem von Brewer formuliert. Im Umfeld der NoSQL-Datenbanken wird daher häufig das BASE-Prinzip (Basically Available, Soft state, Eventual consistency) verfolgt.



Datenspeicherung

Eine typische Datenstruktur unterhalb des DBMS sind die B-Tree's. Davon gibt es einige Varianten, welche Vor- und Nachteile haben.

Des Weiteren kann eine Datenbank die Werte in unterschiedlichen Orientierungen ablegen, also z. B. spalten- oder zeilenweise.



Skalierung

Kehren wir zum **Big Data** Engineering zurück und **fragen**: wie kann ein (R)DBMS **skaliert** werden?

Es gibt zwei grundlegende Ansätze:

1. **Scale up** - oder -
2. **Scale out**

Also entweder den **einzelnen** Datenbankserver **aufrüsten** oder aber die **Datenbank** auf mehrere Maschinen zu **verteilen**.



Einheit 5

- Rückblick auf Einheit 4
- **NoSQL „Datenbanken“ im Überblick**
 - Grundlagen
 - Gründe für NoSQL
 - Zugrundeliegende Annahmen
 - Entwurfsmuster
 - Fragenkatalog
- HBase als Beispiel



Warum NoSQL?

Nachdem wir uns die **Merkmale** der relationalen Datenbanken angeschaut haben, **fragen** wir, warum diese (R)DBM Systeme nicht **alles** abhandeln können, was **Datenspeicherung** und **Datenabfrage** betrifft.

Was sind die **Merkmale** dieser Systeme und wo bestehen **Probleme** in unserer Big Data Welt?

Merkmale von Speichersystemen

- **Datenmodell**
 - Zeilen, Dokumente oder Graphen
 - Relational, Graph, Objektorientiert
- **Orientierung der Datenspeicherung**
 - Nach Zeilen oder Spalten (oder Spaltengruppen)
- **Transaktionen**
 - Generell ACID Merkmale
- **Abfrageschnittstelle**
 - SQL oder API
- **Skalierbarkeit**
- **Performanz und Verfügbarkeit**
 - Durchsatz und Latenz, MTTR



Warum NoSQL?

NoSQL Systeme sind aus **mehreren** Gründen entstanden. Diese gehen auf die genannten **Merkmale** zurück und versuchen, für **spezielle** Fälle eine „**passendere**“ Lösung zu finden (d. h. zu entwickeln).

Wie auch bei Hadoop kommt hier dazu, dass es heutzutage **einfacher** ist, verteilte Systeme zu **entwickeln**. Hardware ist **preiswert** und es gibt **genügend** technische Hilfsmittel.



Warum NoSQL?

Man muss aber auch erwähnen, dass dies **kein** komplett **neues** Thema ist. Es gab schon immer **spezielle** Systeme für **bestimmte** Aufgaben, welche RDBMSe **nicht** gut genug abdecken konnten (z. B. kdb, GT.M, oder auch Berkeley DB).

NoSQL steht **allgemein** für **nicht-relational**, oder **nicht SQL** basiert, oder **nicht merkmalskonform**.



Gründe für NoSQL

Einer der Gründe ist das Thema **Skalierbarkeit**, denn eins der V's in der Big Data Definition ist **Volume**. Aber auch **Velocity** ist hier ein Thema: Daten kommen im Internet zu **schnell**, zu **umfangreich** und dann auch noch in einem **anderen Schema (Variety)**. Also sind alle drei V's als Grund zu nennen.



Gründe für NoSQL

Viele der **bestehenden** Datenbanktechnologien sind **Jahrzehnte** alt. Dies **spürt** man unter der Last der drei V's an **vielen** Ecken und Enden. Auch wenn **manche** solcher Systeme bis zu einem **gewissen** Grad mithalten, so sind alleine die **Kosten** ein vierter, sehr wichtiger Grund: **Scale up** kostet viel **Geld** durch Lizenzen und spezieller Hardware. **Scale out** liegt den DBMSen eher **weniger**.



Scale out vs. Scale up

Wie erwähnt, ist Scale up verbunden mit **exponentiell steigenden Kosten**.

Für Scale out bietet sich das **Sharding** an, also das **Aufteilen** der Daten in physikalisch **getrennte** Bereiche. Dabei **verliert** aber die Datenbank meistens an **Funktionalität**. Aber noch viel **schlimmer** ist die operative **Komplexität**, welche **nicht** zu unterschätzen ist.

Deswegen **bieten** sich verteilte **NRDBMSe** an.



Nicht-quelloffene NRDBMSes

Hier gibt es einige bekannte Systeme:

- Google BigTable
 - Wird in Google für viele Anwendungen genutzt,
z. B. Earth, Reader, Maps, Blogger, WebCrawl
- Amazon Dynamo
 - Für den Einkaufskorb benutzt, auch als SaaS
verfügbar (AWS)
- Yahoo! PNUTS
 - Nur interne Nutzung, Details sind unbekannt



Datenschnittstellen

Der Zugriff auf die Daten ist meistens eine API mit einfachen Funktionen:

- Jede Zeile hat eine eindeutigen Schlüssel (PK)
- Datenpaar get() und put() Funktion
- multiget() und multiput()
- Bereichsabfrage? Mit Prädikat Weiterleitung?
- MapReduce Anbindung?
- SQL?



Einheit 5

- Rückblick auf Einheit 4
- **NoSQL „Datenbanken“ im Überblick**
 - Grundlagen
 - Gründe für NoSQL
 - **Zugrundeliegende Annahmen**
 - Entwurfsmuster
 - Fragenkatalog
- HBase als Beispiel

Annahme: Datengröße



Folgende **Annahmen** werden gemacht:

- Die Daten passen **nicht** auf einen **einzelnen** Knoten (Rechner)
- Die Daten passen möglicherweise **nicht** mehr in einen **Rechnerschrank**
- SAN Lösungen sind zu **teuer**

Ergebnis:

- Das System **muss** die Daten **partitioniert** über **viele** Knoten hinweg ablegen



Annahme: Ausfallsicherheit

Folgende **Annahmen** werden gemacht:

- Das System **muss** hochverfügbar sein, um Web (und andere) Anwendungen zu **bedienen**
- Weil das System auf **vielen** Rechnern läuft, sind Ausfälle von Knoten **normal** während des normalen Betriebs
- Daten müssen **sicher** sein, auch wenn Festplatten oder ganze Rechner **ausfallen**

Ergebnis:

- Das System **muss** die Daten auf **mehrere** Knoten replizieren und **muss** trotz dem Ausfall bestimmter Systemkomponenten **verfügbar** sein.



Annahme: Leistungsfähigkeit

Folgende **Annahmen** werden gemacht:

- Alle besprochenen Systeme müssen den drei V's gerecht werden und sind für den nicht-Batch Einsatz gedacht (real time)
- 95. oder 99. Perzentil ist wichtiger als durchschnittliche Latenzzeit
- Gebräuchliche Hardware und langsame Platten

Ergebnis:

- Das System **muss** auf Standard-Hardware gut funktionieren und auch während der Wiederherstellung von Systemkomponenten eine niedrige Latenzzeit aufrechterhalten.



Einheit 5

- Rückblick auf Einheit 4
- **NoSQL „Datenbanken“ im Überblick**
 - Grundlagen
 - Gründe für NoSQL
 - Zugrundeliegende Annahmen
 - **Entwurfsmuster**
 - Fragenkatalog
- HBase als Beispiel



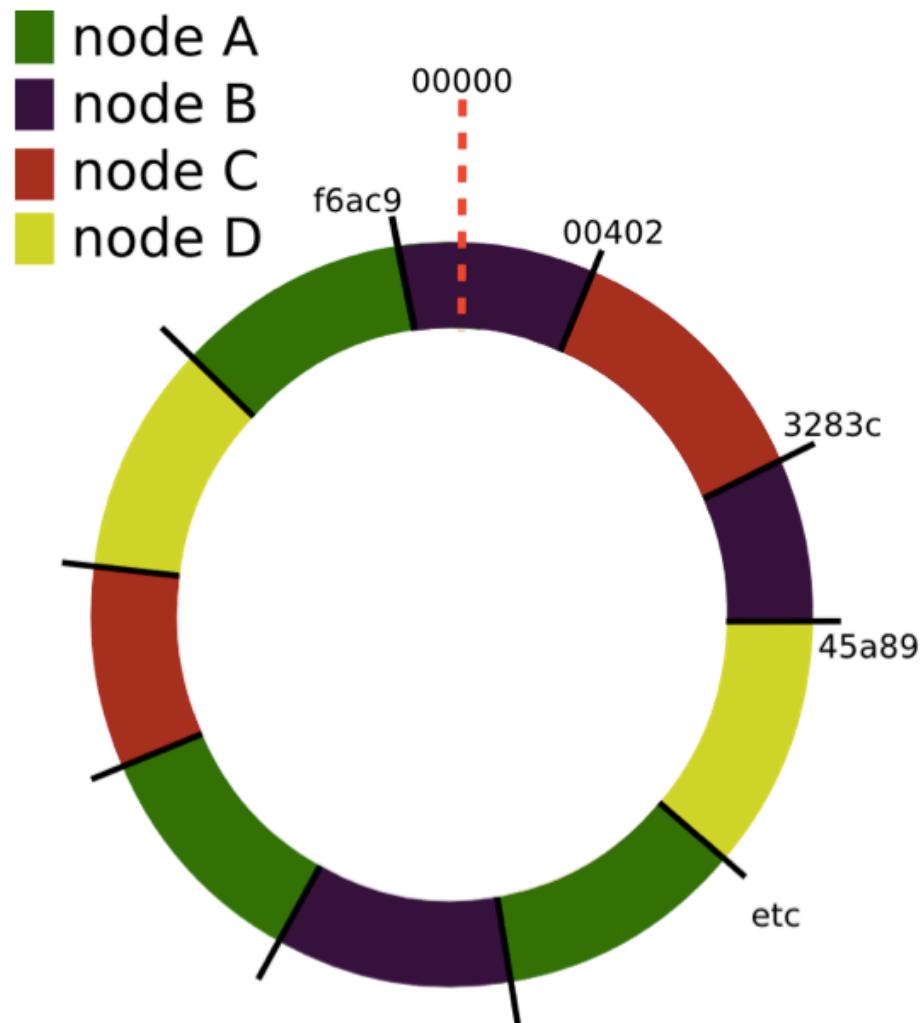
Partitionierung

Für eine Partitionierung der Daten sind folgende Merkmale **wichtig**:

- Für einen **gegebenen** Schlüssel muss man **herausfinden**, zu welchem Knoten er gehört
- Wenn der Knoten **nicht** verfügbar ist, dann muss eine Kopie auf einem **anderen** gefunden werden
- Schwierigkeiten
 - **Unbegrenzte** Anzahl an Schlüssel
 - **Dynamische** Cluster Mitgliedschaft
 - Knoten **Ausfall**

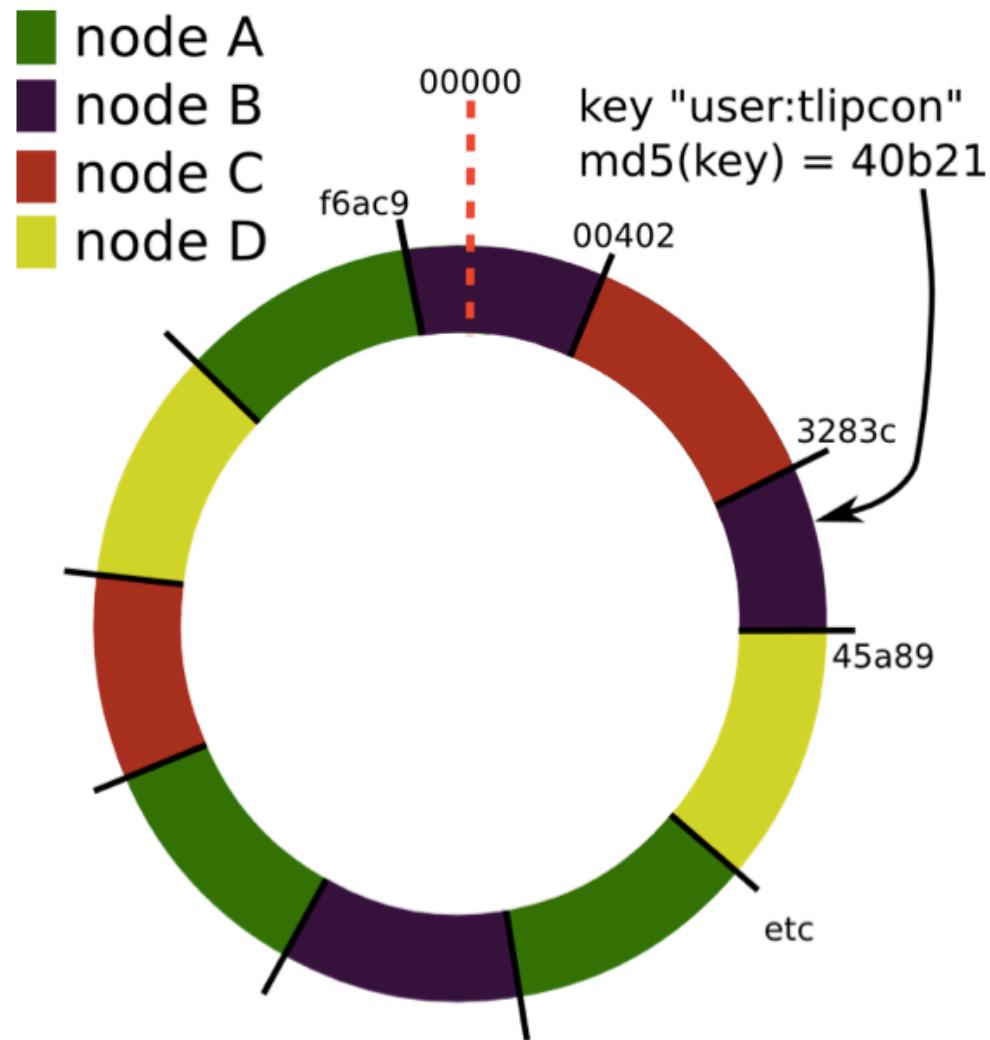
Konsistentes Hashing

Um einen Schlüssel Hashwert in einem dynamischen Cluster erhalten zu können, werden die möglichen Hashwerte in Bereiche aufgeteilt und Knoten zugeordnet.



Konsistentes Hashing

Aus einem Schüsselwert wird ein Hash erzeugt und dann über die Zuordnung zu dem gegebenen Bereich der Knoten gefunden.



Konsistenzmodelle



Ein Konsistenzmodell **bestimmt** die Regeln für die **Sichtbarkeit** und augenscheinliche **Reihenfolge** der Änderungen.

Beispiel:

- Zeile X ist repliziert auf Knoten M und N
- Client A schreibt Zeile X auf Knoten N
- Eine Zeit t vergeht
- Client B liest Zeile X von Knoten M
- Sieht Client B die Änderungen von Client A?

Konsistenz ist ein **Kontinuum mit Kompromissen!**



Strikte Konsistenz

Alle Leseoperationen müssen die Daten der letzten kompletten Schreiboperation liefern, egal auf welchem Replikat die Operation lief.

Dies erfordert eine von diesen Bedingungen:

- Alle Operationen für eine bestimmte Zeile gehen an denselben Knoten -oder-
- Die Knoten benutzen eine Art Protokoll für verteilte Transaktionen (z. B. 2-Phase Commit oder Paxos)

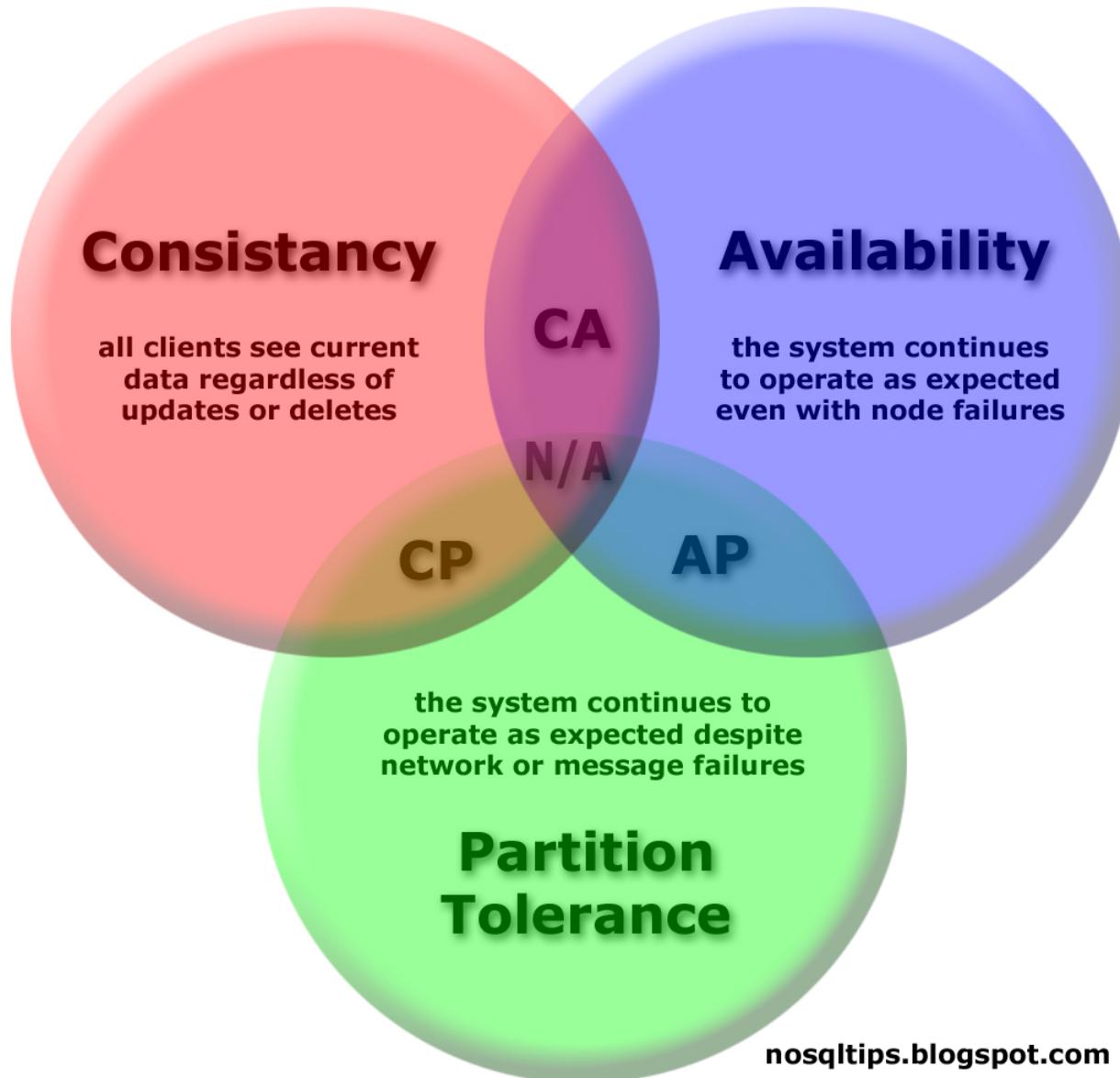


CAP-Theorem

Hier kommt der CAP Satz von Brewer zum Tragen:

„Das CAP-Theorem oder Brewer's Theorem besagt, dass es in einem verteilten System unmöglich ist, gleichzeitig die drei Eigenschaften Konsistenz, Verfügbarkeit und Partitionstoleranz zu garantieren.“

CAP Theorem



Visual Guide to NoSQL Systems

Availability:
Each client can
always read
and write.

Data Models

Relational (comparison)
Key-Value
Column-Oriented/Tabular
Document-Oriented

A

CA

RDBMSs
(MySQL,
Postgres,
etc)

Aster Data
Greenplum
Vertica

AP

Dynamo
Voldemort
Tokyo Cabinet
KAI

Cassandra
SimpleDB
CouchDB
Riak

Pick Two

C

Consistency:
All clients always
have the same view
of the data.

CP

BigTable
Hypertable
Hbase

MongoDB
Terrastore
Scalarmis

Berkeley DB
MemcacheDB
Redis

P

Partition Tolerance:
The system works
well despite physical
network partitions.



CAP-Theorem

Beispiele:

- **AP (Nicht-strikte Konsistenz)**
 - Domain Name Service
 - Cloud Dienste
 - BASE Systeme
- **CA (Monolithisches System)**
 - Relationale DBMSes
- **CP (Entfernte Anbindung)**
 - Banking Anwendungen



Schlussendliche Konsistenz

Im Englischen wird diese als **Eventual Consistency** bezeichnet. Dies kann man so **nicht** direkt übersetzen. Es geht darum, dass **irgendwann** ein konsistenter Zustand erreicht wird, also eine **zeitabhängig** besteht. Diese kann **unbemerkt** sein, aber auch **Stunden** oder **Tage** bedeuten. Wichtig ist, das **letztendlich** der konsistente Zustand **erreicht** wird, deswegen bezeichnen wir hier dieses Merkmal als **schlussendliche Konsistenz**.



Schlussendliche Konsistenz

- Während $t \rightarrow \infty$ sehen alle Leser die Änderungen
- In einem stabilen Zustand gibt das System garantiert schlussendlich den zuletzt geschriebenen Wert zurück
 - Beispiel: DNS oder MySQL Slave Replikation (d. h. binlog kopieren)
- Spezielle Fälle der Schlussendlichen Konsistenz
 - Lesen der eigenen Veränderungen (read-your-own-writes [RYOW], z. B. „Gesendete E-Mails“ Ordner)
 - Kausale Konsistenz: Wenn Y geschrieben wird nachdem X gelesen wurde, sieht jeder der Y liest auch X
 - Gmail hat RYOW aber nicht kausale Konsistenz!

Zeitstempel und Vektoruhren



Schlussendliche Konsistenz **verlässt** sich auf das Entscheiden, welchen Wert einer Zeile **schlussendlich** annehmen wird.

Wenn aber zwei Schreiboperationen **gleichzeitig** stattfinden, dann ist das **schwierig**. Eine Lösung sind **Zeitstempel**, wofür aber die Zeit **synchronisiert** sein muss. Außerdem wird die Kausalität **nicht** erfasst.

Vektoruhren (Vector Clocks) sind eine alternative Methode der Erfassung von Reihenfolge in **verteilten** Systemen.



Vektoruhr

Definition:

- Eine Vektoruhr ist ein Tupel $\{ t_1, t_2, \dots, t_n \}$ von Zeitwerten (Uhrzeit) von jedem Knoten
- $v_1 < v_2$ wenn
 - Für alle i , $v_{1i} \leq v_{2i}$
 - Für wenigstens ein i , $v_{1i} < v_{2i}$
- $v_1 < v_2$ bedeutet globale Zeitordnung der Events

Wenn Daten vom Knoten i geschrieben wird, dann setzt es t_i auf dessen Zeitwert. Dies erlaubt der schlussendlichen Konsistenz das Auflösen der Konsistenz zwischen Schreiboperationen auf mehreren Replikaten.



Datenmodelle

Was ist in einer Zeile?

- Primärschlüssel → Wert
- Werte können sein:
 - Blob
 - Strukturiert (Menge an Spalten)
 - Semistrukturiert (Menge an Spaltenfamilien mit beliebigen Spalten, z. B. linkto:<url> in webtable)
 - Jede dieser hat Vor- und Nachteile
- Sekundäre Indizes? Tabellen/Namensraum?



Multiversionierte Speicherung

Eine **dritte** Dimension mit Zeitstempel:

- Jede Tabellenzelle hat einen **Zeitstempel**. Diese müssen **nicht** unbedingt der realen Welt entsprechen.
- **Mehrere** Versionen (und Grabmale) können für eine **gegebene** Zeile existieren.
- Leseoperationen können den aktuellsten, aktuellsten vor Zeitpunkt T usw. Wert zurückgeben (vgl. Snapshots)
- Systeme können optimistische Concurrency Kontrollen mit compare-and-swap auf Basis der Zeitstempel anbieten



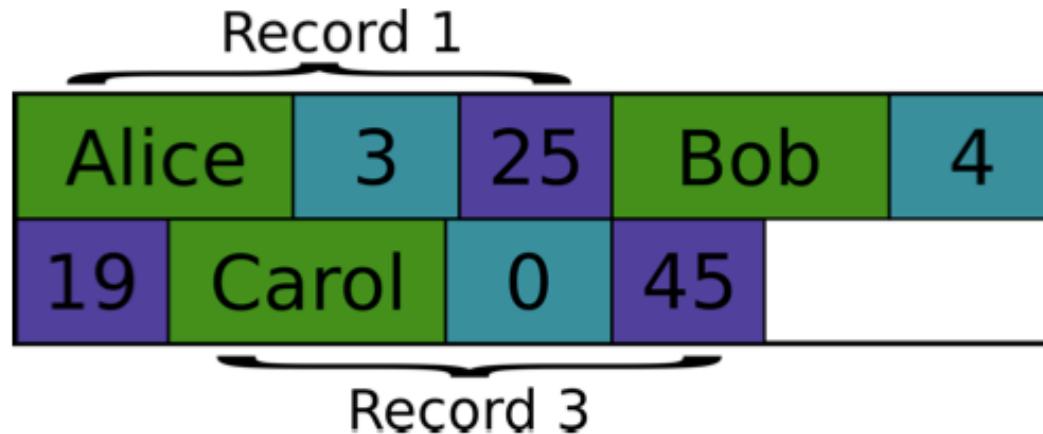
Speicherungslayout

Wie schon in der **letzten** Einheit besprochen, kann jedes System **sich entscheiden**, **wie** es die Daten in Dateien speichert. Dies kann je nach Zugriffsmuster einen **entscheidenden** Vorteil bringen. Das Speicherlayout **überträgt** sich **direkt** auf den Festplattenzugriff.

Fragen: Was ist wichtig? Schnelles Schreiben/
Lesen/Durchlaufen (scan)? Wird die ganze Zeile
oder nur Untermengen an Spalten benötigt?



Zeilenbasierte Speicherung



Vorteile

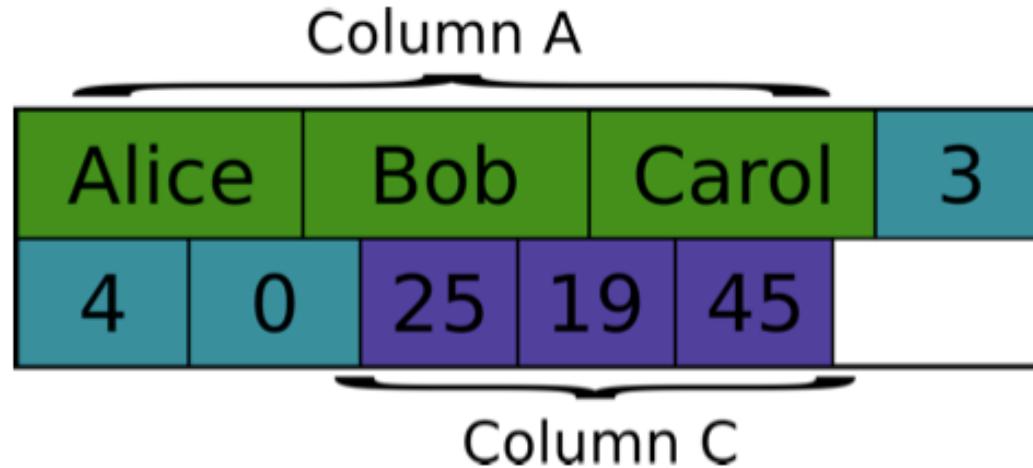
- Gute Lokalität des Zugriffs (auf der Platte und im Cache) auf verschiedene Spalten
- Lesen/Schreiben einer einzelnen Zeile ist eine einzige I/O Operation

Nachteile

- Wenn nur eine Spalten benötigt wird muss trotzdem die ganze Zeile gelesen werden



Spaltenbasierte Speicherung



Vorteile

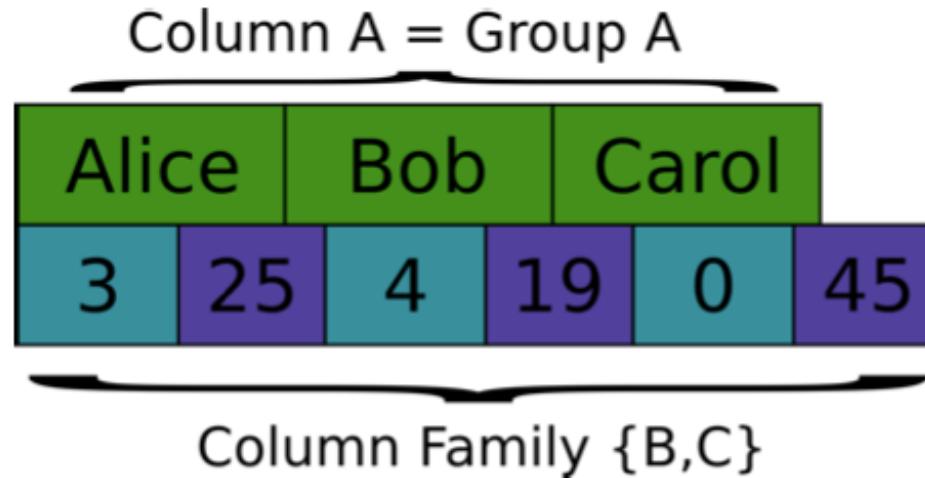
- Daten einer gegebenen Spalte sind **sequentiell** gespeichert
- Durchlaufen **einer** Spalte (z. B. für eine aggregierende Abfrage) ist schnell

Nachteile

- Lesen einer **Zeile** könnte pro Spalte eine **Seek** Operation darstellen



Spaltenbasierte Speicherung



- Spalten sind in **Familien** organisiert, die **Lokalitätsgruppen**
- Vorteil der zeilenbasierten Speicherung **innerhalb** der Gruppe
- Vorteil der spaltenbasierten Speicherung durch **ignorieren** ganzer Gruppen

Log Structured Merge Trees



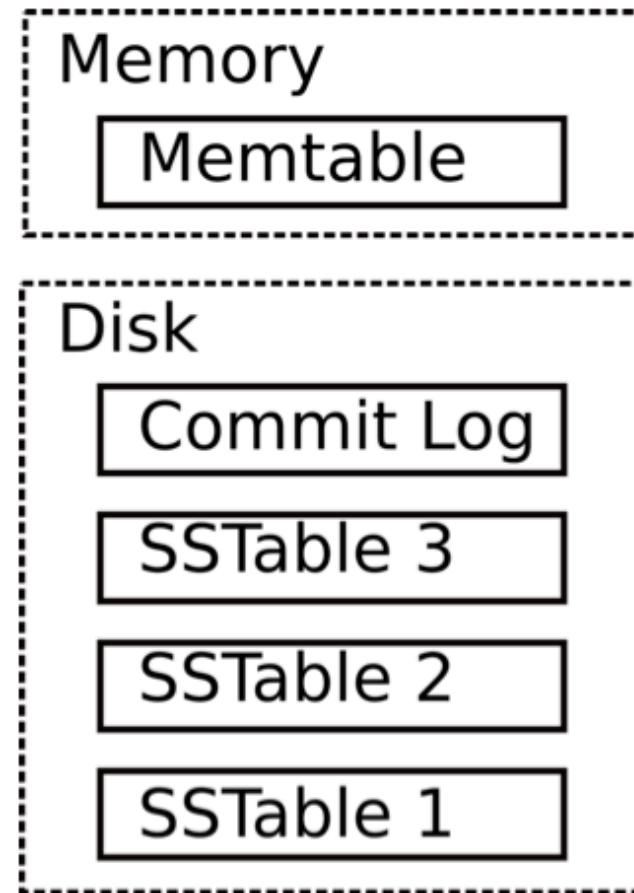
Google's BigTable ist darauf aufgebaut:

- Wahlfreie I/O beim Schreiben ist schlecht (und manchmal sogar unmöglich in einigen DFSen)
- LSM Trees wandeln wahlfreies Schreiben in sequentielles Schreiben um
- Änderungen gehen in ein Commit-Log und den Hauptspeicher (Memtable)
- Der Memtable wird ab und an auf die Festplatte geschrieben (SSTable)
- Die auf den Platten gespeicherten SSTables werden regelmäßig verdichtet

Siehe: P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil: The log-structured merge-tree (LSM-tree), Acta Informatica, 1996

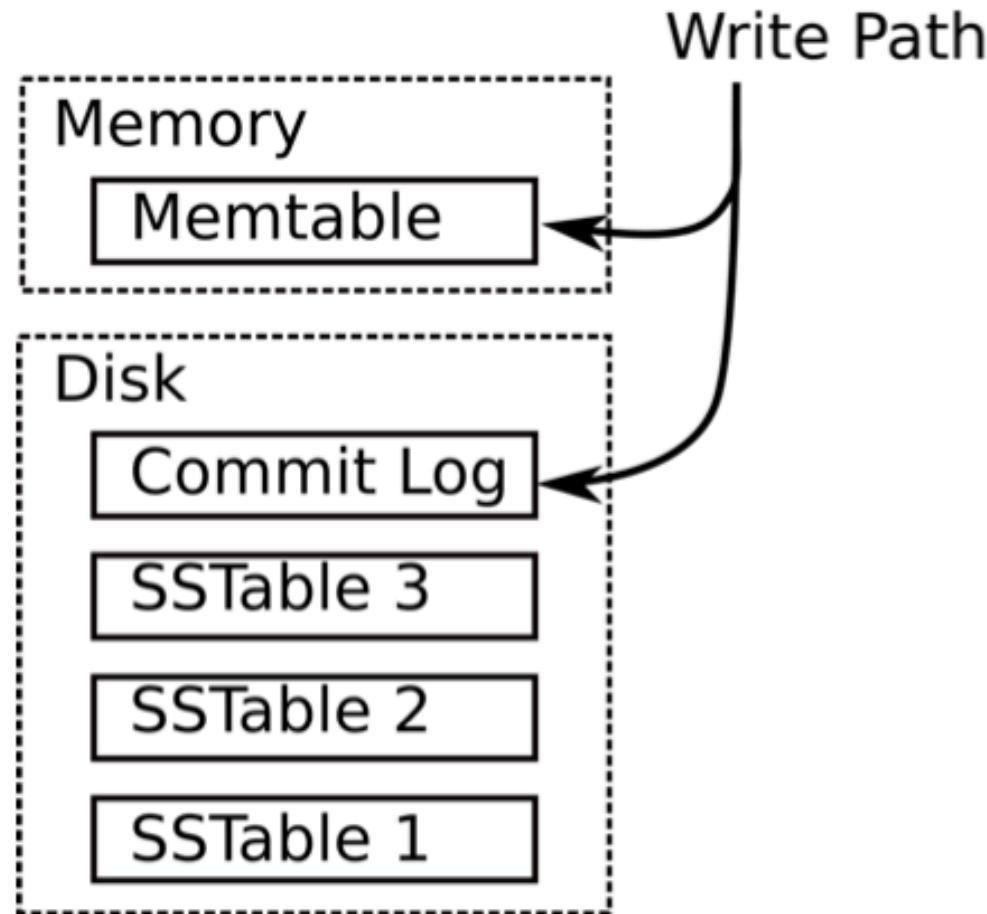


LSM Datenlayout



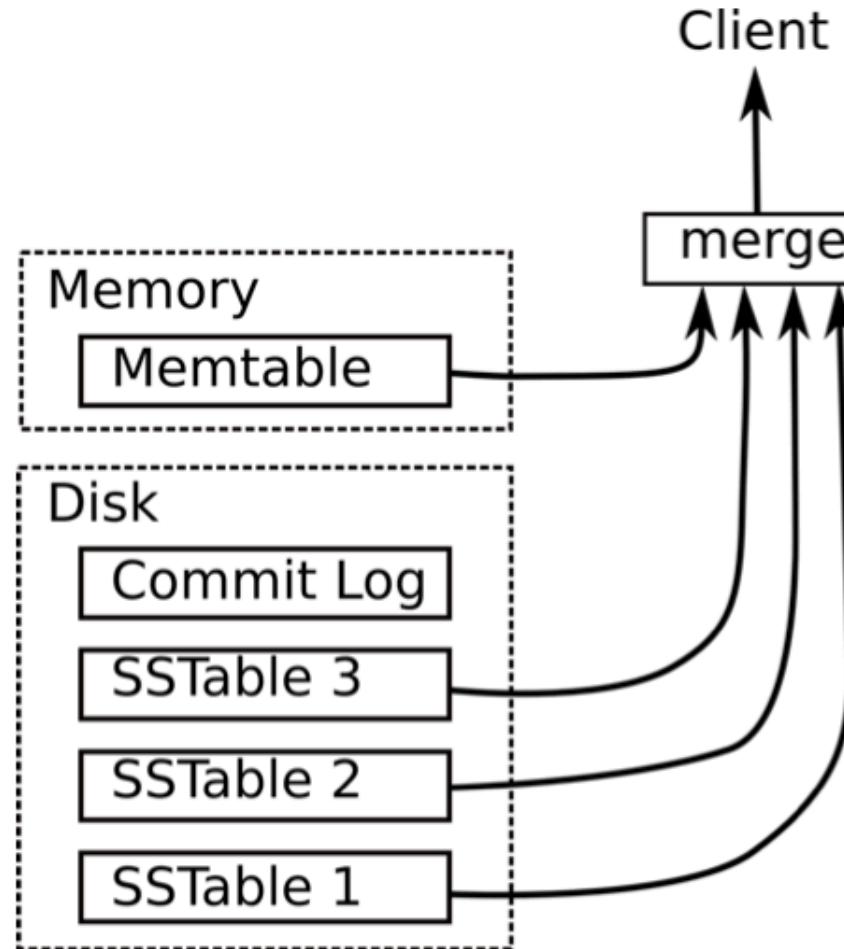


LSM Schreibpfad





LSM Lesepfad

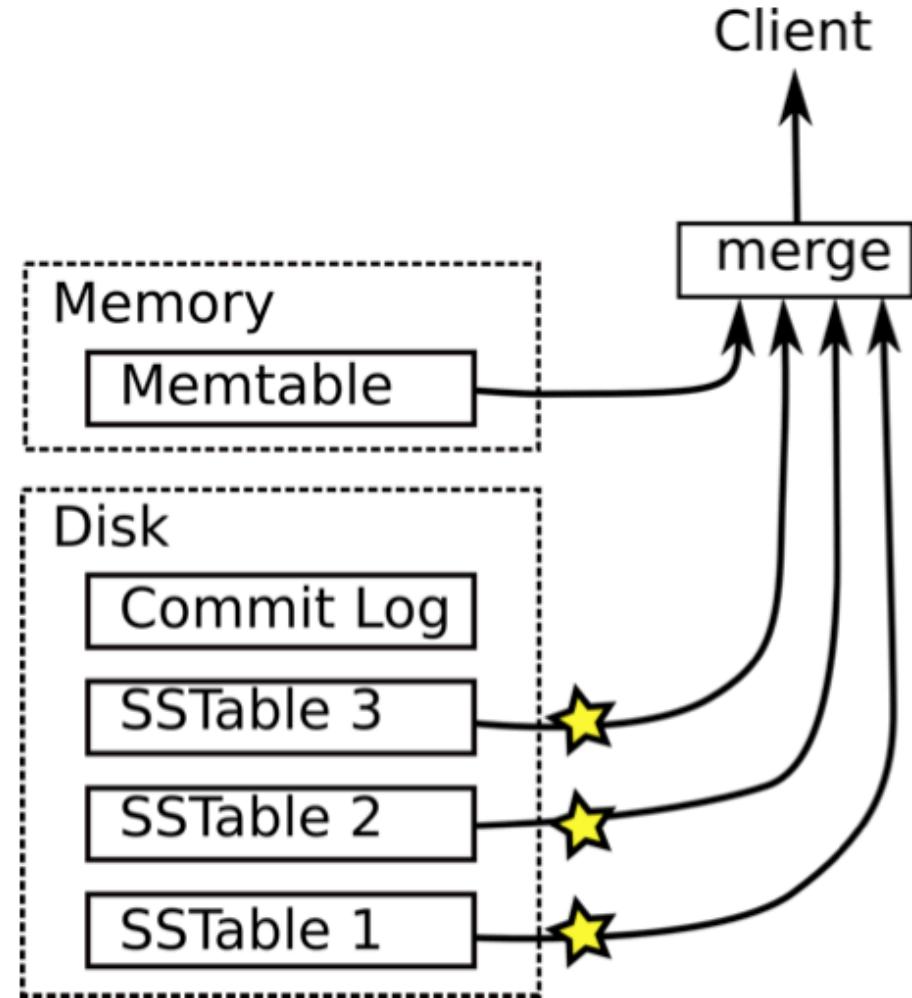


LSM Lesen mit Bloom Filter



Bloom Filter helfen die I/O Operationen zu vermindern:

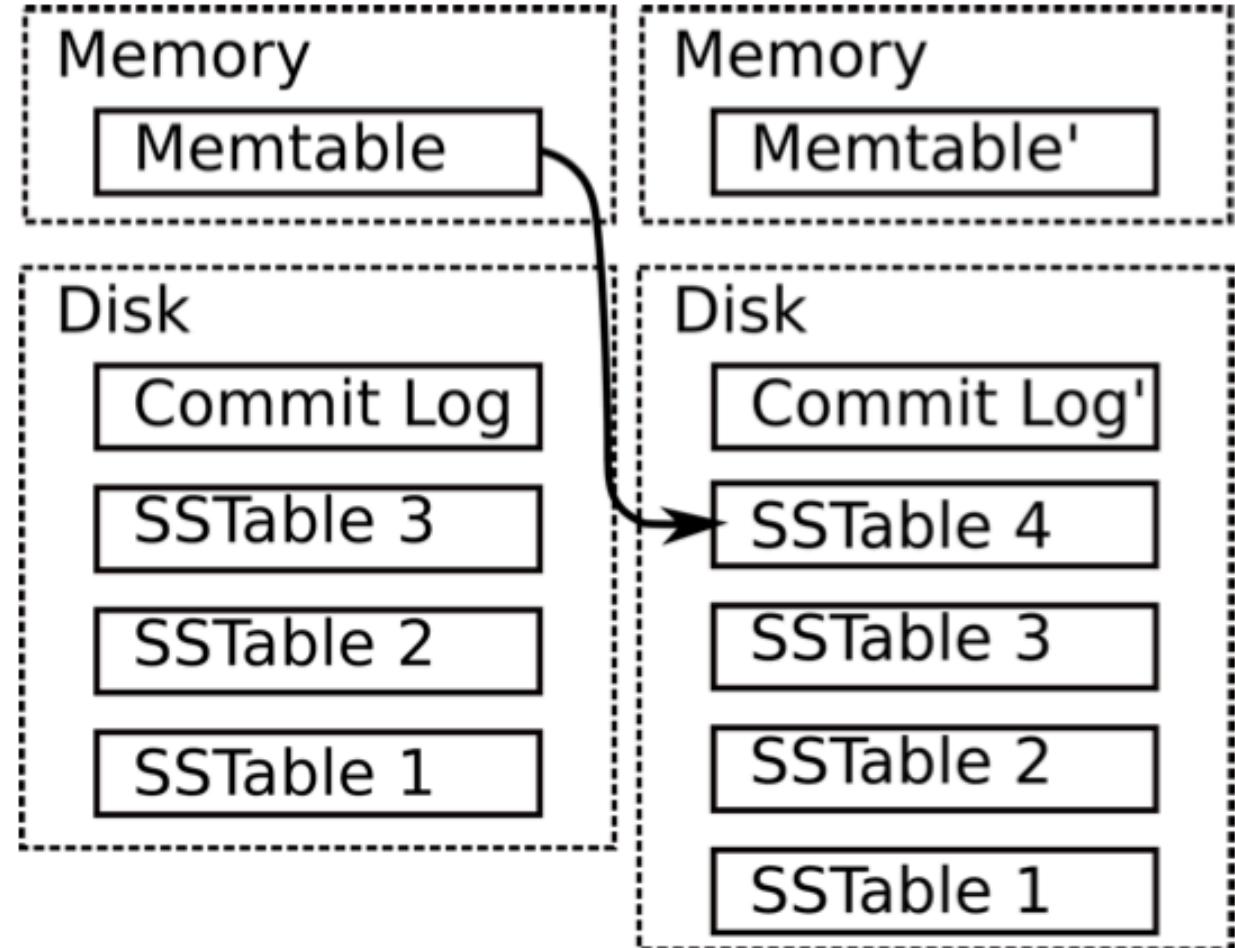
Anstatt immer Daten aus den SSTables zu lesen findet hier zuerst eine Prüfung auf Existenz statt.





LSM Memtable Speicherung

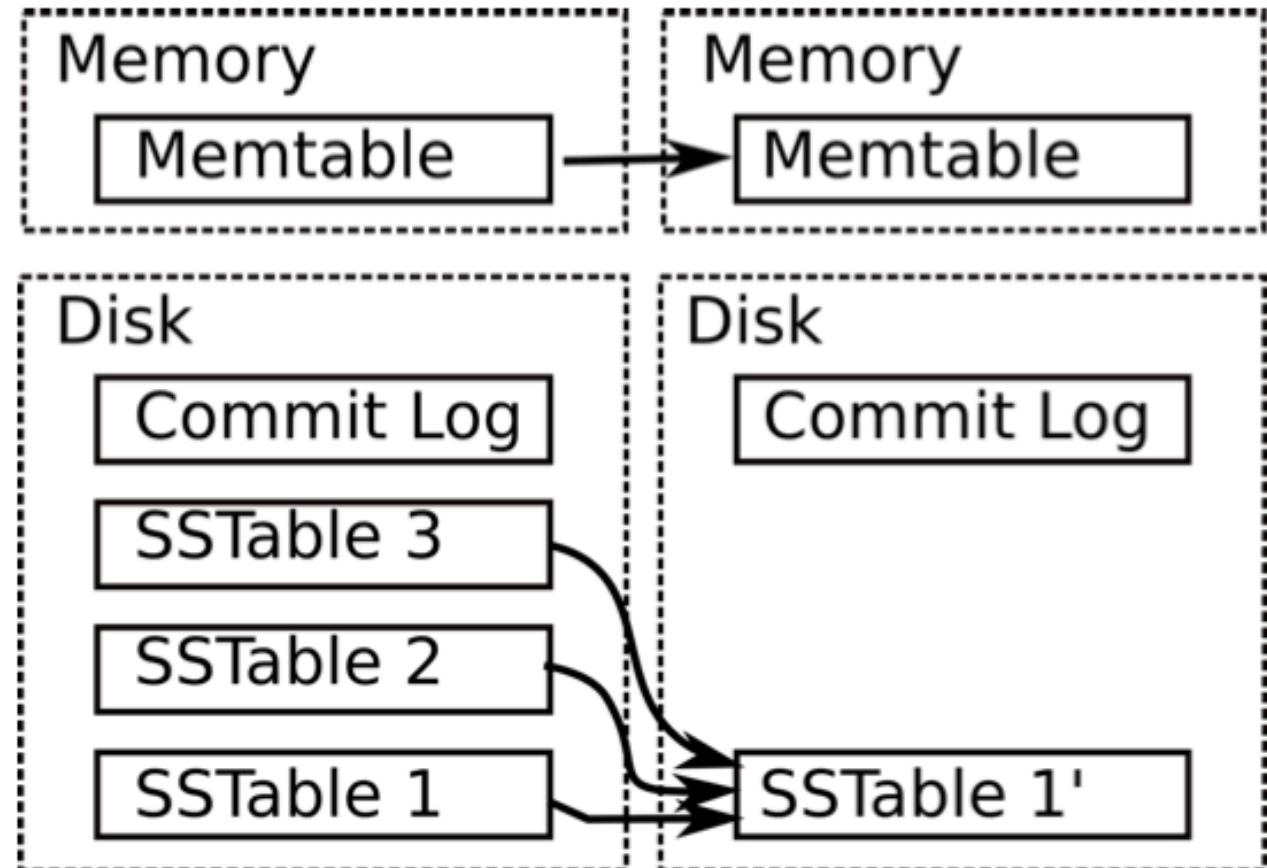
Wird auch
als Flush
bezeichnet.





LSM Verdichtung

Die Compaction verdichtet auf weniger, aber dafür größere Dateien.





Clusterverwaltung

Clients, also z. B. Anwendungen, **müssen** wissen, wo Daten zu **finden** sind (Token eines konsistenten Hashings usw.). Auch **innerhalb** müssen sich die Knoten sehr wahrscheinlich **finden** können. Der Ausfall und die Wiederherstellung von Knoten macht eine **statische** Konfigurationsdatei **sinnlos**. Irgendwie muss ein **konsistenter** Zustand des Clusters **verfügbar** sein.



Allwissender Master

In dieser Bauweise **müssen** Knoten wenn sie kommen oder gehen (oder den Zustand ändern) sich mit dem Master **unterhalten**. Der Master hat die **verbindliche** Übersicht auf die Welt um sich.

Vorteil: einfach, eine einzige konsistente Ansicht auf den Cluster

Nachteil: möglicherweise ein SPOF, außer der Master ist hochverfügbar. Nicht P tolerant (CAP)



Gossip

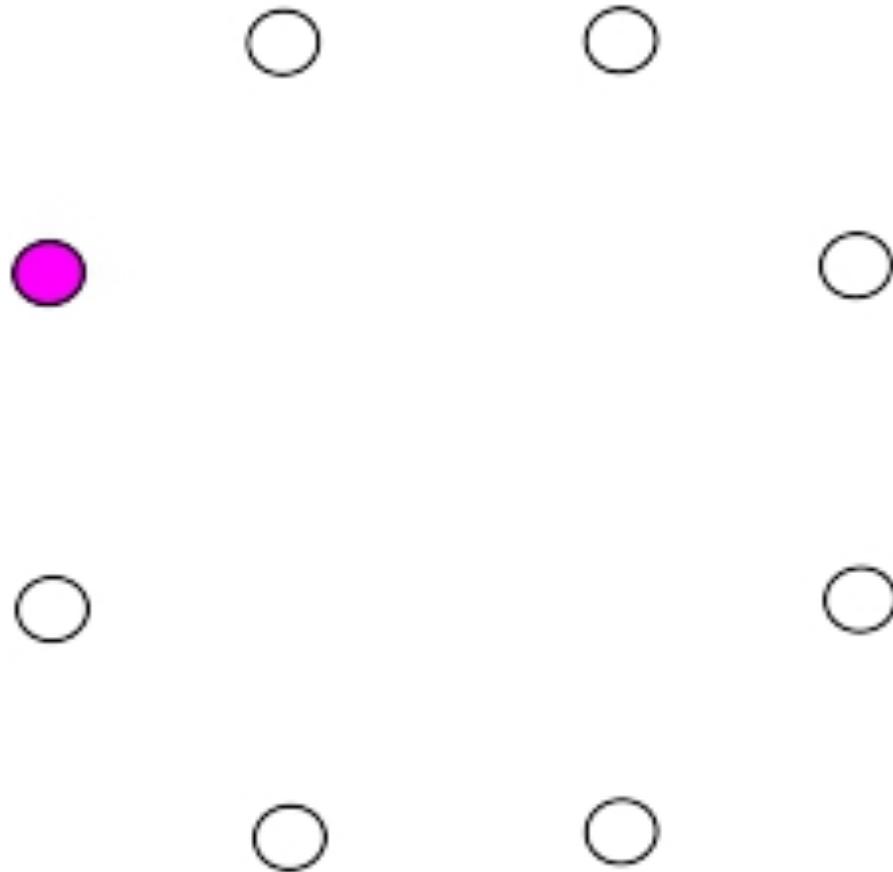
Hier **unterhalten** sich alle Knoten **untereinander**, um den Status des Clusters **abzugleichen**. Dabei **selektiert** alle t Sekunden ein Knoten einen anderen Knoten, **tauscht** mit diesem seinen Status bidirektional aus und **merkt** sich einen Zeitstempel für **seinen**, und den Status aller **andere** Knoten.

Information über den Zustand des Clusters **verbreitet** sich in **$O(lgn)$** Durchläufen (schlussendlich konsistent).

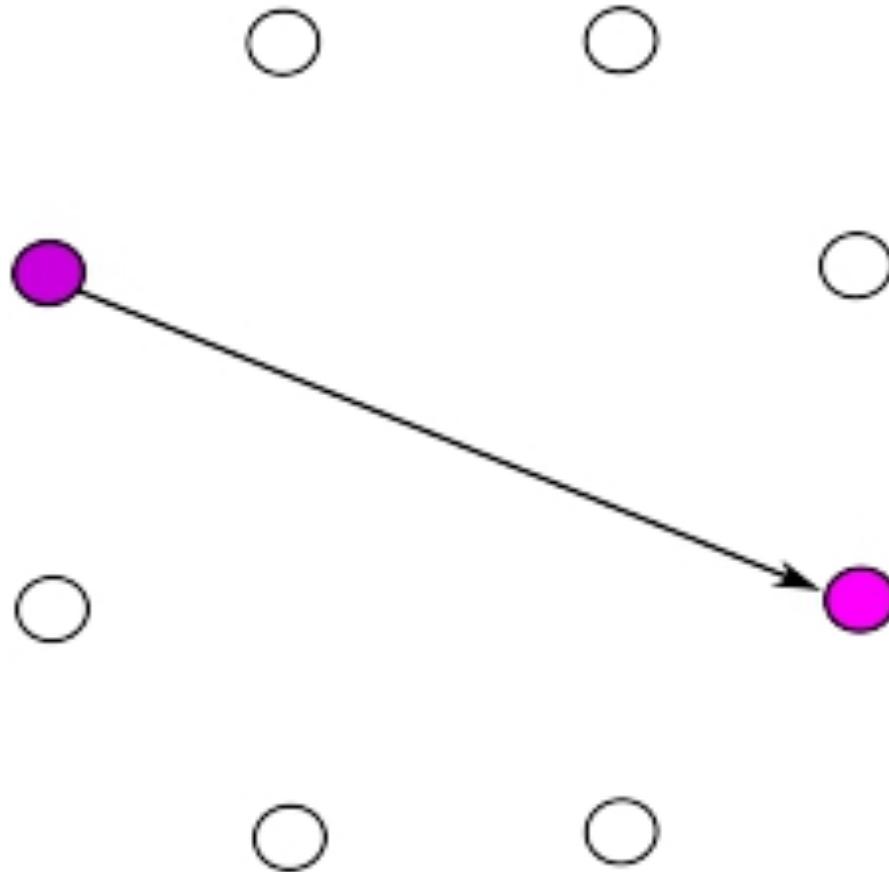
Skaliert und kein SPOF, aber Status vielleicht **nicht** immer **konsistent**.



Gossip - Anfangszustand

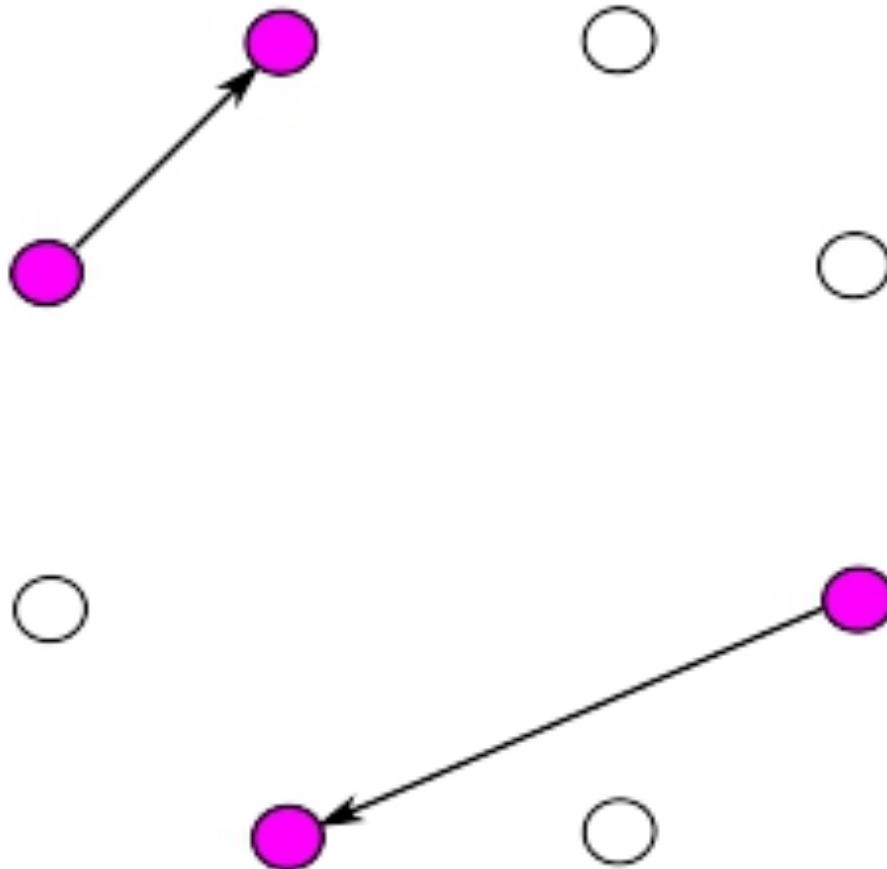


Gossip – Runde 1



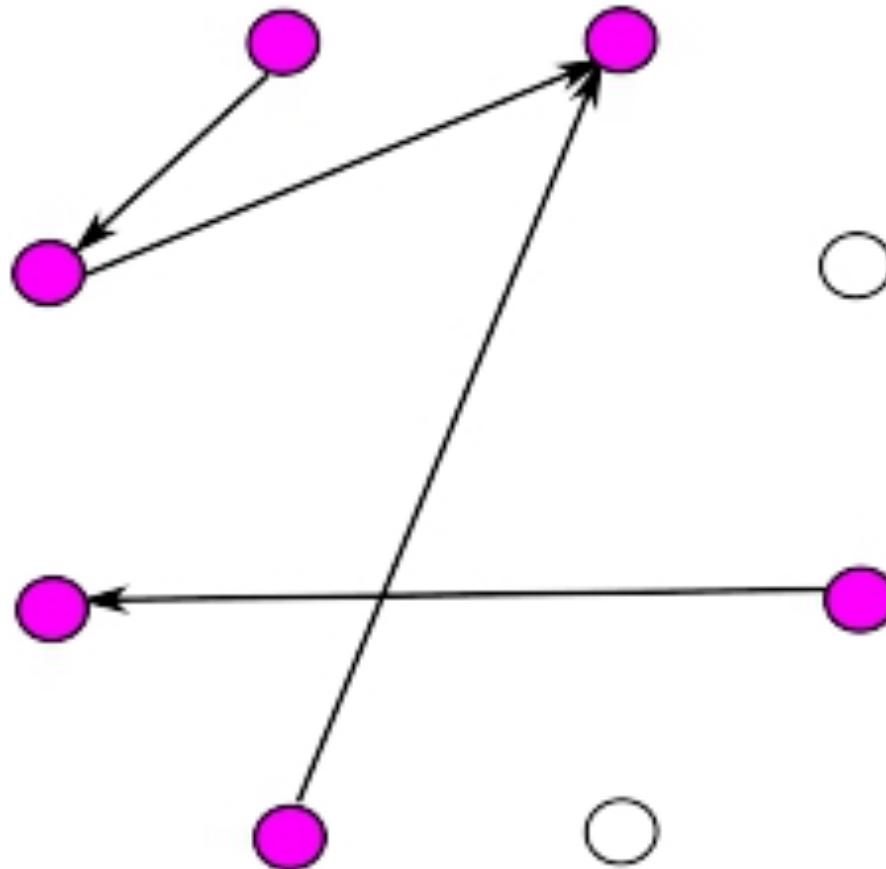


Gossip – Runde 2



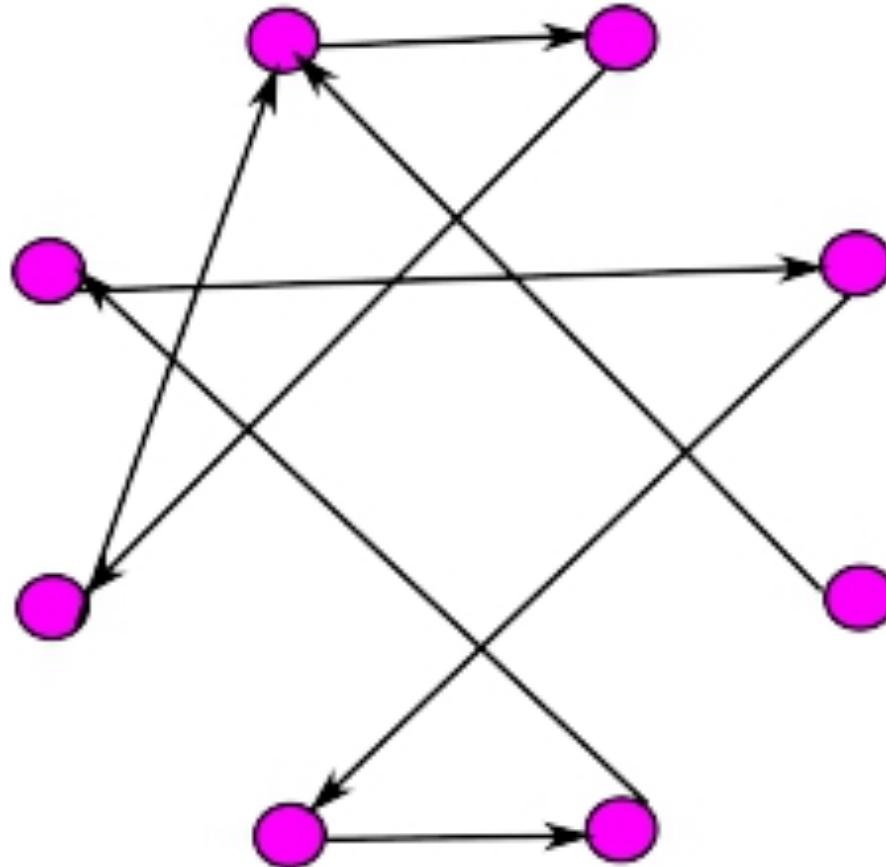


Gossip – Runde 3





Gossip – Runde 4





Einheit 5

- Rückblick auf Einheit 4
- **NoSQL „Datenbanken“ im Überblick**
 - Grundlagen
 - Gründe für NoSQL
 - Zugrundeliegende Annahmen
 - Entwurfsmuster
 - **Fragenkatalog**
- HBase als Beispiel



Skalierung & Verfügbarkeit

- Was sind die Engpässe bei der Skalierung? Wie reagiert das System wenn es überlastet wird?
- Gibt es SPOFs, d. h. einzelne Ausfallpunkte die einen Totalausfall bedeuten?
- Wenn Knoten ausfallen, kann das System immer noch alle Daten ausliefern?
- Wenn Replikate ausfallen, repliziert das System automatisch um den Mangel auszugleichen?
- Wenn neue Knoten hinzugefügt werden, gleicht sich das System automatisch aus?



Leistungsfähigkeit

- Was ist das Ziel? Batch Durchsatz oder Abfrage Latenzzeiten?
- Wie viele Suchoperationen (Seeks) beim Lesen? Beim Schreiben?
- Welche 99. Perzentil Latenzzeiten wurden in der Praxis gemessen?
- Wie beeinflussen Ausfälle die Latenzzeiten?
- Welcher Durchsatz wurde in der Praxis für die Massenbeladung gemessen?



Konsistenz

- Welches Konsistenzmodell bietet das System an?
- Welche Situation könnte eine Verletzung der Konsistenz verursachen, wenn überhaupt möglich?
- Kann die Art der Konsistenzerhaltung über die Konfiguration des Systems angepasst werden?
- Gibt es die Möglichkeit des compare-and-swap für optimistische Blockierung? Mehrere Zeilen?

Clusterverwaltung und Topologie

- Hat das System ein einzigen Master? Benutzt es Gossip, um Clusterverwaltungsinformation zu verteilen?
- Kann es Netzwerkpartitionierungen aushalten und weiterhin seinen Dienst ableisten?
- Kann es über mehrere Datencenter verteilt aufgebaut werden, für die Notfallwiederherstellung?
- Können Knoten ohne Abschaltung hinzugefügt und wieder entfernt werden?
- Gibt es Schnittstellen für die Überwachung und Metriken?



Modell und Speicherung

- Welches Datenmodell und Speicherungs-system bietet das System an?
- Ist es austauschbar?
- Welche I/O Muster verursacht das System mit verschiedenen Lasten?
- Ist das System besser für wahlfreien oder sequentiellen Zugriff geeignet? Eher mehr lesender oder mehr schreibender Zugriff?
- Gibt es Einschränkungen bei den Größen für Schlüssel, Werte oder Zeilen?
- Gibt es Komprimierung?



Zugriffsmethoden

- Welche Arten für den Datenzugriff gibt es?
Kann es aus der Sprache X heraus
angesprochen werden?
- Kann man auf der Clusterseite Daten vorfiltern
oder selektieren?
- Gibt es Werkzeuge für die Massenbe- und
entladung, für effizienten Zugriff?
- Existiert eine Backup/Restore Funktion für die
Daten?



Reifegrad

- Wer benutzt das System? Wie groß sind die Cluster auf denen es läuft und welche Last müssen sie aushalten?
- Wer entwickelt das System? Gibt es eine Entwicklergemeinschaft oder ist eine einzelne Organisation dahinter? Ist die Mitarbeit von außerhalb gerne gesehen?
- Wer unterstützt das System? Gibt es eine aktive Gemeinschaft welche bei der Installation und Fehlerbehebung hilft? Dokumentation?
- Welche open-source Lizenz?
- Was ist die Entwicklungsplanung?



Warum NoSQL System XYZ?

- Frage: Warum XYZ und nicht <*put-your-favorite-nosql-solution-here*>?
- Was gibt es an Varianten?
 - Key/Value Speicher
 - Dokumentenorientierte Speicher
 - Spaltenorientierte Speicher
 - Graph-orientierte Speicher
- Merkmale nach denen gefragt werden kann
 - Im Hauptspeicher oder persistent?
 - Strikt oder schlussendlich konsistent?
 - Verteilt oder monolithischer Rechner?



Key/Value Speicher

- Auswahl (beispielhaft)
 - MemCached,
 - Tokyo Cabinet, MemCacheDB, Membase, Redis
 - Voldemort, Dynomite, Scalaris
 - Dynamo, Dynomite
 - Berkeley DB
- Vorteile
 - Als Zwischenspeicher genutzt
 - Einfache APIs
 - Schnell
- Nachteile
 - Schlüssel müssen bekannt sein (oder berechenbar)
 - Skaliert nur mit eigener Leistung (konsistentes Hashing usw.)
 - Kann strukturierte Daten nicht darstellen



Dokumenten Speicher

- Beispiele
 - MongoDB
 - CouchDB
- Vorteile
 - Strukturierte Daten werden unterstützt
 - Schema frei
 - Unterstützt Änderungen an den Dokumenten ohne Rekonfiguration
 - Manche haben Hilfsindexe und/oder Suche
- Nachteile
 - Alles ist an einem Ort gespeichert, verträgt sich schlecht mit gemischten Daten
 - Skalierbarkeit ist entweder nicht bewiesen oder ähnlich komplex wie bei RDBMSen
 - Keine ideale oder fehlende Integration mit MapReduce (keine Massenbeladung oder Datenlokalitätsvorteile)



Spaltenorientierte Speicher

- Hybride Architekturen
 - HBase, BigTable
 - Cassandra
 - Vertica, C-Store
- Vorteile
 - Erlauben den Zugriff nur auf relevante Daten
- Nachteile
 - Begrenzter Funktionsumfang, um Modell genüge zu tun



Welches System nehmen?

- Key/Value Speicher
 - Für Zwischenspeicher/Caches
 - Einfache Daten
 - Geschwindigkeit ist wichtig
- Dokumenten Speicher
 - Sich verändernde Schemas
 - Dokumentenorientierte Funktion werden in der Anwendung benötigt
- Spaltenorientierte Speicher
 - Skalierbarkeit
 - Gemischte Benutzung



Welches System nehmen?

- Hauptspeicher oder auf Platte
 - Zwischenspeicher/Cache oder Datenbank
- Strikte Konsistenz
 - Einfach zu handhaben auf der Anwendungsseite
 - Content-management Systeme (CMS), Banking usw.
- Schlussendliche Konsistenz
 - Höhere Verfügbarkeit, könnte aber veraltete Daten lesen
 - Konflikte und Reparaturen werden in der Anwendung abgehandelt
 - Einkaufskörbe, Spiele



Einheit 5

- Rückblick auf Einheit 4
- NoSQL „Datenbanken“ im Überblick
 - Grundlagen
 - Gründe für NoSQL
 - Zugrundeliegende Annahmen
 - Entwurfsmuster
- **HBase als Beispiel**



Was ist HBase?

- Verteilt
- Spaltenorientiert
- Multi-Dimensional
- Hochverfügbar
- Hochleistungsfähig
- Speicherungssystem

Projekt Ziele

Milliarden an Zeilen * Millionen von Spalten * Tausende an
Versionen

Petabytes über tausende von gebräuchlichen Rechner



HBase ist nicht...

- Eine relationale, SQL-basierte Datenbank
 - Keine JOINs, keine Abfragemaschine, keine Typen, kein SQL (zu mindestens nicht nativ)
 - Keine Transaktionen und Hilfsindexe
- Ein leicht auszutauschender Ersatz für eine RDBMS
- Man muss entgegen dem relationalen Modell denken
 - De-normalisierte Daten
 - Breite und dünnbesetzte Tabellen
 - Man muss dem inneren Datenbankadministrator Nein sagen!

Stichwort: Impedanz Übereinstimmung



HBase Tabellen

	A	B	C	D	E
1					
2					
3					
4					
5					
6					
7					



HBase Tabellen

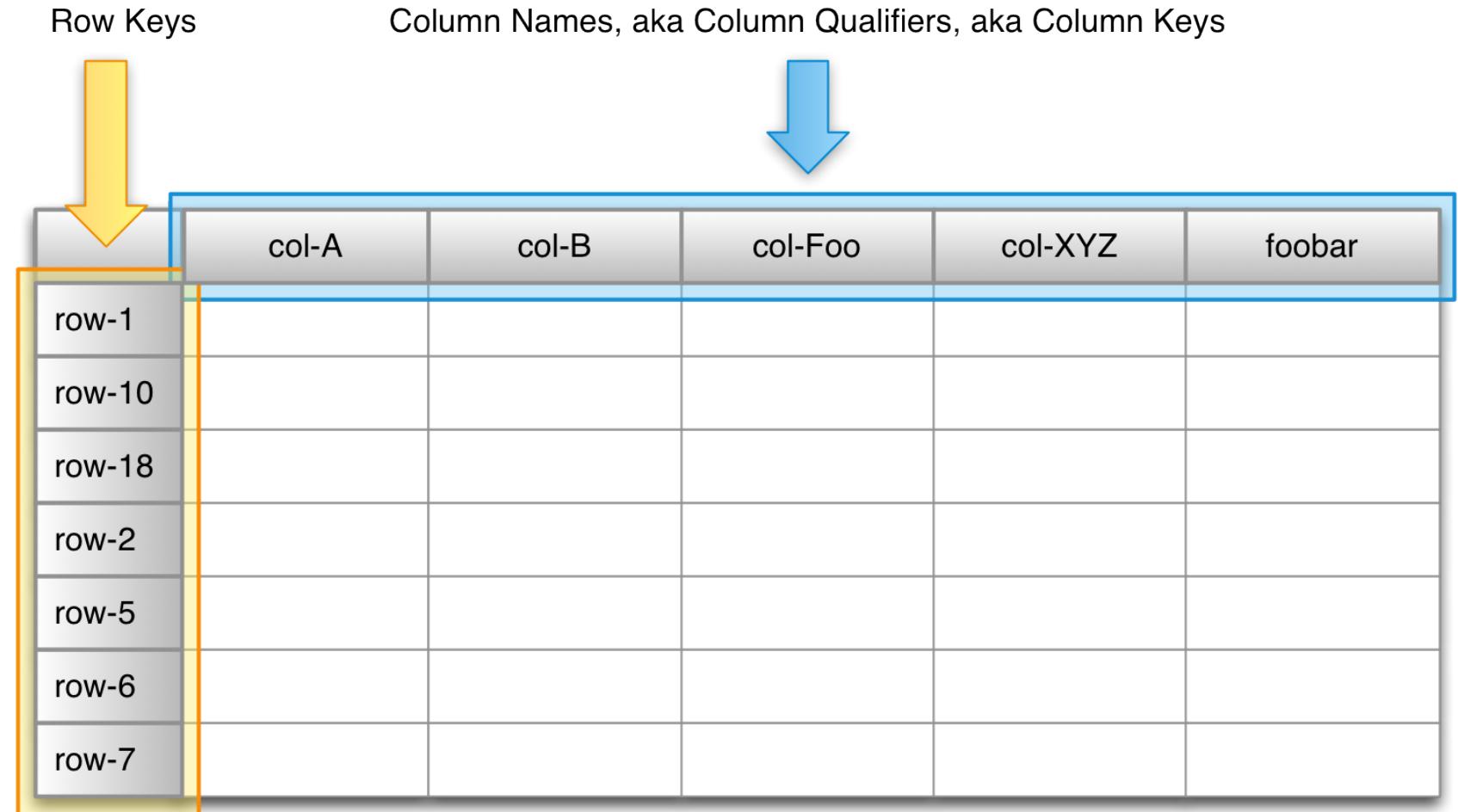
Row Keys

Column Names, aka Column Qualifiers, aka Column Keys

The diagram illustrates the structure of an HBase table. It features a grid with 7 rows labeled 1 through 7 on the left, representing Row Keys. Above the grid, a yellow arrow points down to the first row, labeled 'Row Keys'. The grid has 5 columns labeled A, B, C, D, and E at the top, representing Column Qualifiers. Above the grid, a blue arrow points down to the first column, labeled 'Column Names, aka Column Qualifiers, aka Column Keys'. The cells in the grid are empty, indicating no data has been written.

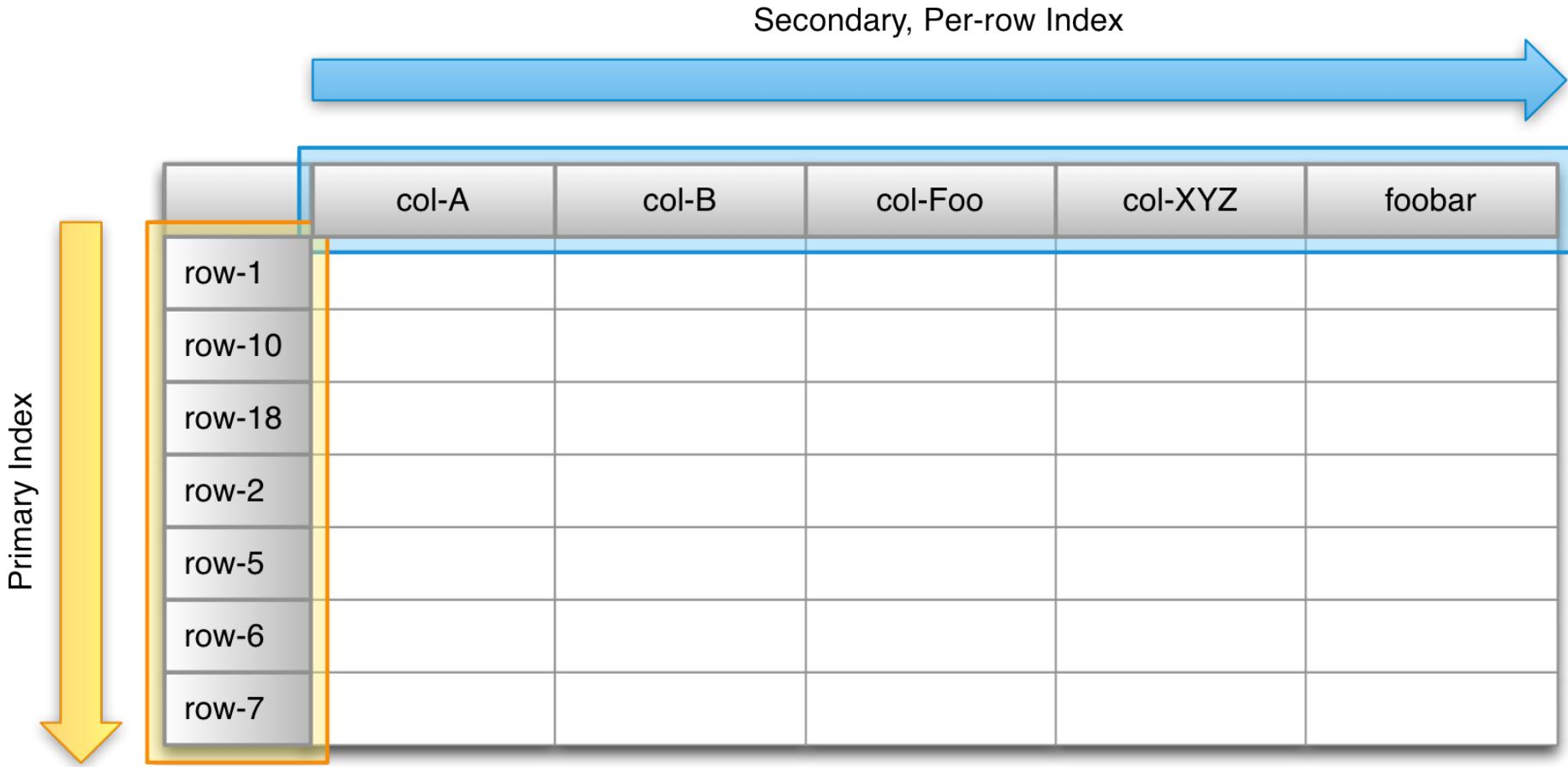
	A	B	C	D	E
1					
2					
3					
4					
5					
6					
7					

HBase Tabellen



HBase Tabellen

Ascending, Lexicographically Sorted Indexes



HBase Tabellen



Ascending, Lexicographically Sorted Indexes

Secondary, Per-row Index



	col-A	col-B	col-Foo	col-XYZ	foobar
row-1					
row-10					

Primary Index

	col-A	col-D	col-Foo2	col-XYZ	col-XYZ2
row-10					
row-18					

	20130423	20130424	20130425	20130426	20130427
row-18					
row-2					

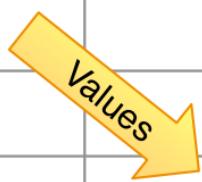


	MaxVal - ts5	MaxVal - ts4	MaxVal - ts3	MaxVal - ts2	MaxVal - ts1
row-2					



HBase Tabellen

	col-A	col-B	col-Foo	col-XYZ	foobar
row-1					
row-10					
row-18	A18	B18	Foo18	XYZ18	foobar18
row-2					
row-5					
row-6					
row-7					





HBase Tabellen

	col-A	col-B	col-Foo	col-XYZ	foobar
row-1					
row-10					
row-18	A18 - v1	B18 - v3	Foo18 - v1	XYZ18 - v2	foobar18 - v1
row-2					
row-5					
row-6					
row-7					



HBase Tabellen

	col-A	col-B	col-Foo	col-XYZ	foobar					
row-1										
row-10										
row-18	A18 - v1	▼	B18 - v3	▼	Foo18 - v1	▼	XYZ18 - v2	▼	foobar18 - v1	▼
row-2										
row-5										
row-6										
row-7										



HBase Tabellen

	col-A	col-B	col-Foo	col-XYZ	foobar
row-1					
row-10					
row-18	A18 - v1 ▼	B18 - v3 ▼	Foo18 - v1 ▼	XYZ18 - v2 ▼	foobar18 - v1 ▼
row-2		Peter - v2 Bob - v1		Mary - v1	
row-5					
row-6					
row-7					

Coordinates for a Cell: *Row Key → Column Name → Version*

HBase Tabellen



		Column Family 1			Column Family 2		
		cf1:col-A	cf1:col-B	cf2:col-Foo	cf2:col-XYZ	cf2:foobar	
Region 1	row-1						
	row-10						
	row-18	A18 - v1	▼	B18 - v3	▼	Foo18 - v1	▼
	row-2			Peter - v2 Bob - v1		Mary - v1	
Region 2	row-5						
	row-6						
	row-7						

Coordinates for a Cell: Row Key → Column Family Name → Column Qualifier → Version

HBase Tabellen



Column Family 1

	cf1:col-A		cf1:col-B	
row-1				
row-10				
row-18	A18 - v1	▼	B18 - v3	▼

Column Family 2

	cf2:col-Foo		cf2:col-XYZ		cf2:foobar	
row-1						
row-10						
row-18	Foo18 - v1	▼	XYZ18 - v2	▼	foobar18 - v1	▼

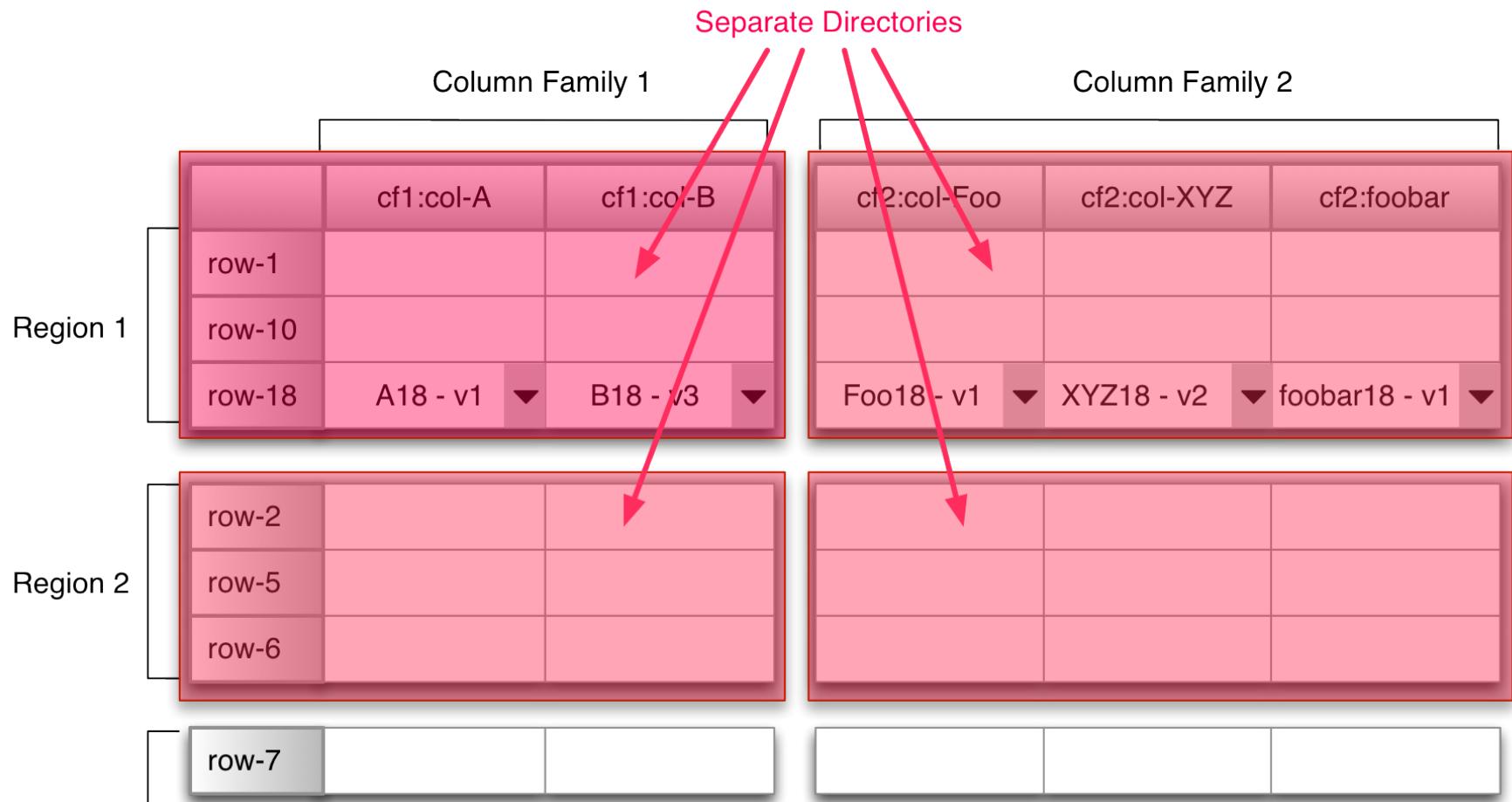
Region 1

Region 2

row-2		
row-5		
row-6		
row-7		

Physical Coordinates for a Cell: Region Directory → Column Family Directory
 → Row Key → Column Family Name → Column Qualifier → Version

HBase Tabellen



Physical Coordinates for a Cell: *Region Directory* → *Column Family Directory*
→ *Row Key* → *Column Family Name* → *Column Qualifier* → *Version*



HBase Architektur

- Eine Tabelle (Table) besteht aus einer oder mehreren Regionen (Regions)
- Eine Region wird über ihre Start- und Endschlüssel definiert (start/end key)
 - Leere Tabelle: (Table, NULL, NULL)
 - Tabelle mit zwei Regionen: (Table, NULL, “com.cloudera.www”) und (Table, “com.cloudera.www”, NULL)
- Jede Region kann auf einem verschiedenen Knoten laufen und besteht aus mehreren HDFS Dateien und Blöcken, jeder davon repliziert durch Hadoop



HBase Architektur (fort.)

- Zwei Knotentypen in HBase:
Master und RegionServer
- Spezielle Katalogtabellen –ROOT– (bis 0.94) und .META. speichern Schema Informationen und wo Regionen sich befinden
- Master Server ist verantwortlich die RegionServer zu überwachen und Regionen gleichmäßig und komplett zu verteilen
- Benutzt ZooKeeper als verteilten Koordinationsdienst
 - Für Master Auswahl und Server Verfügbarkeit



HBase Tabellen

- Tabellen sind nach Zeilenschlüssel in lexikographischer Reihenfolge sortiert
- Tabellen Schemas definieren nur die Spaltenfamilien
 - Jede Familie besteht aus einer oder mehreren Spalten
 - Jede Spalte hat eine oder mehrere Versionen
 - Spalten existieren nur wenn sie geschrieben werden, d. h. NULLs kosten nichts
 - Spalten innerhalb einer Familie werden auch sortiert und in Blöcken gespeichert
 - Alles ausser Tabellennamen sind byte []
- Koordinaten eines Werts (einer Zelle):
(Tabellen, Zeile, Familie:Spalte, Zeitstempel) -> Wert

HBase Tabelle als Datenstruktur



SortedMap(

RowKey, List(

SortedMap(

Column, List(

Value, Timestamp

)

)

)

)

SortedMap(RowKey, List(SortedMap(Column, List(Value, Timestamp))))



Don't think
static columns:

	col1	col2	col3
row1	Σ	Σ	Σ
row2	Σ	Σ	Σ
row3	Σ	Σ	Σ
	⋮	⋮	⋮



Think:

row1 → col A = Value,
col B = Value,
col C = Value

row2 → col X = Value,
col Y = Value,
col Z = Value



Beispiel: Web Crawl

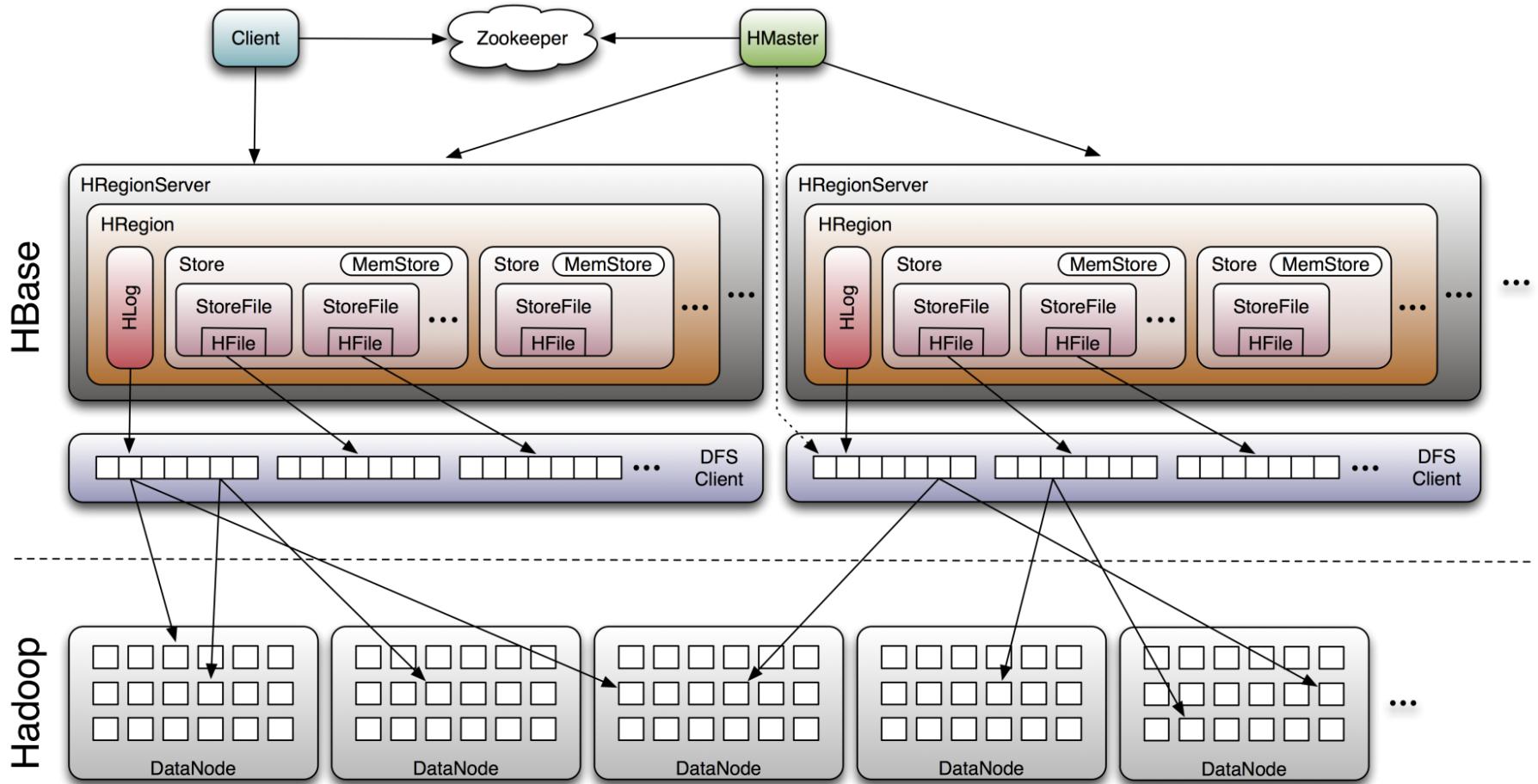
- Kanonischer Anwendungsfall für BigTable
- Speichern des Web Crawl Daten
 - Tabelle **webtable** mit Familie **content** und **meta**
 - Zeilenschlüssel ist umgedrehte URL mit Spalten
 - *content:data* speichert die Rohdaten
 - *meta:language* speichert die Sprache laut HTTP Header
 - *meta:type* speichert den Content-Type HTTP Header
 - Während die Rohdaten bearbeitet werden, werden weitere Spalten für Links auf Seiten und Bilder in den Familien **links** and **images** angelegt
 - *links:<rurl>* Spalte für jeden Hyperlink
 - *images:<rurl>* Spalte für jedes Bild



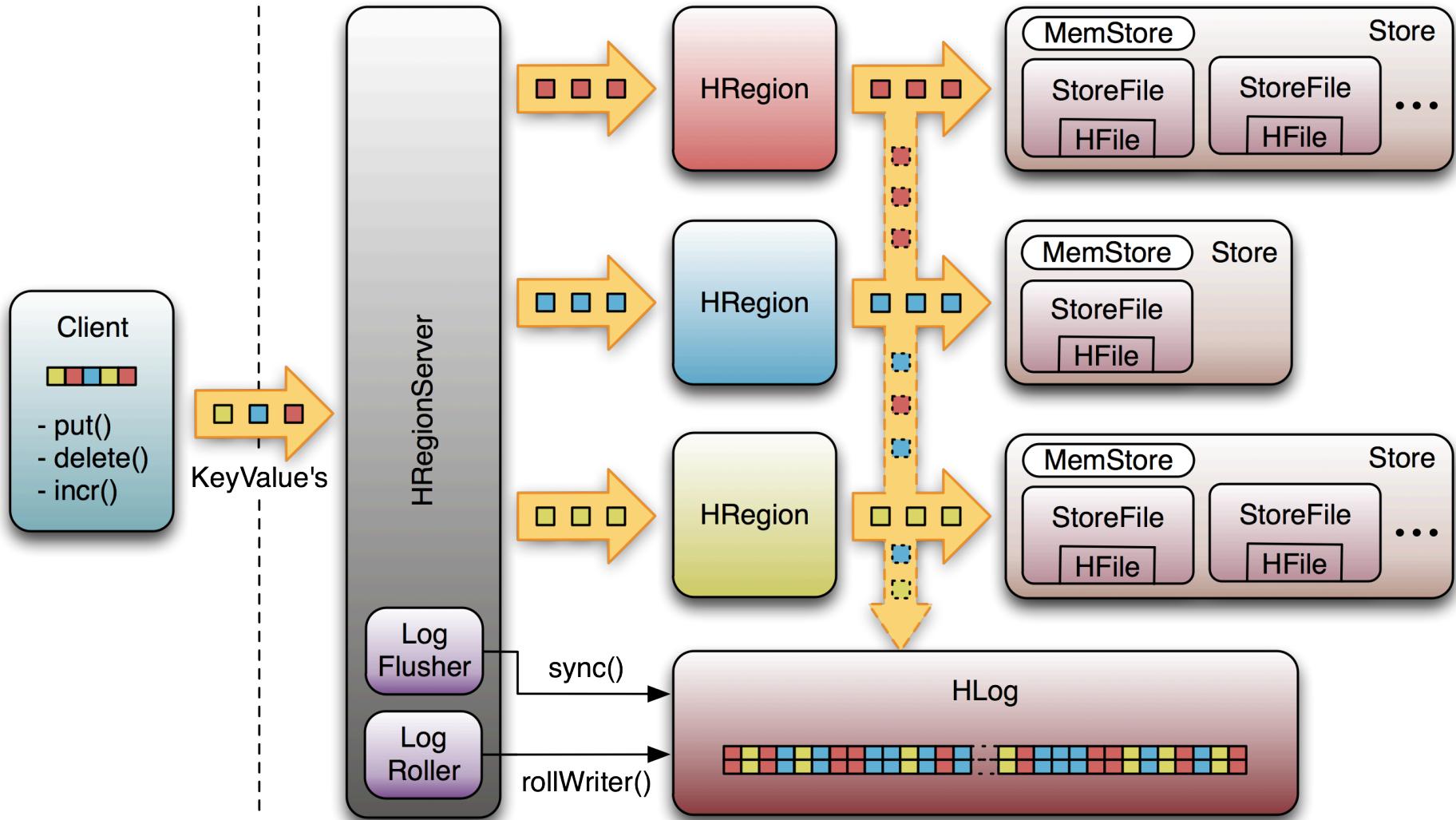
HBase Architektur (fort.)

- Basiert auf Log-Structured Merge-Trees (LSM-Trees)
- Änderungen werden zuerst in das Write-ahead Log geschrieben
- Daten werden im Hauptspeicher gehalten und auf die Platten geschrieben, basierend auf deren Größe oder Zeit
- Kleine Flush Dateien werden verdichtet im Hintergrund, um die Anzahl der Dateien klein zu halten
- Beim Lesen wird erst im Hauptspeicher gelesen und dann auf der Platte in den Dateien
- Das Löschen von Daten wird über Grabmal-Marker erledigt
- Atomarität auf Zeilenebene, egal wie viele Spalten
 - Damit braucht man kein komplexes Blockieren

HBase Architektur (fort.)

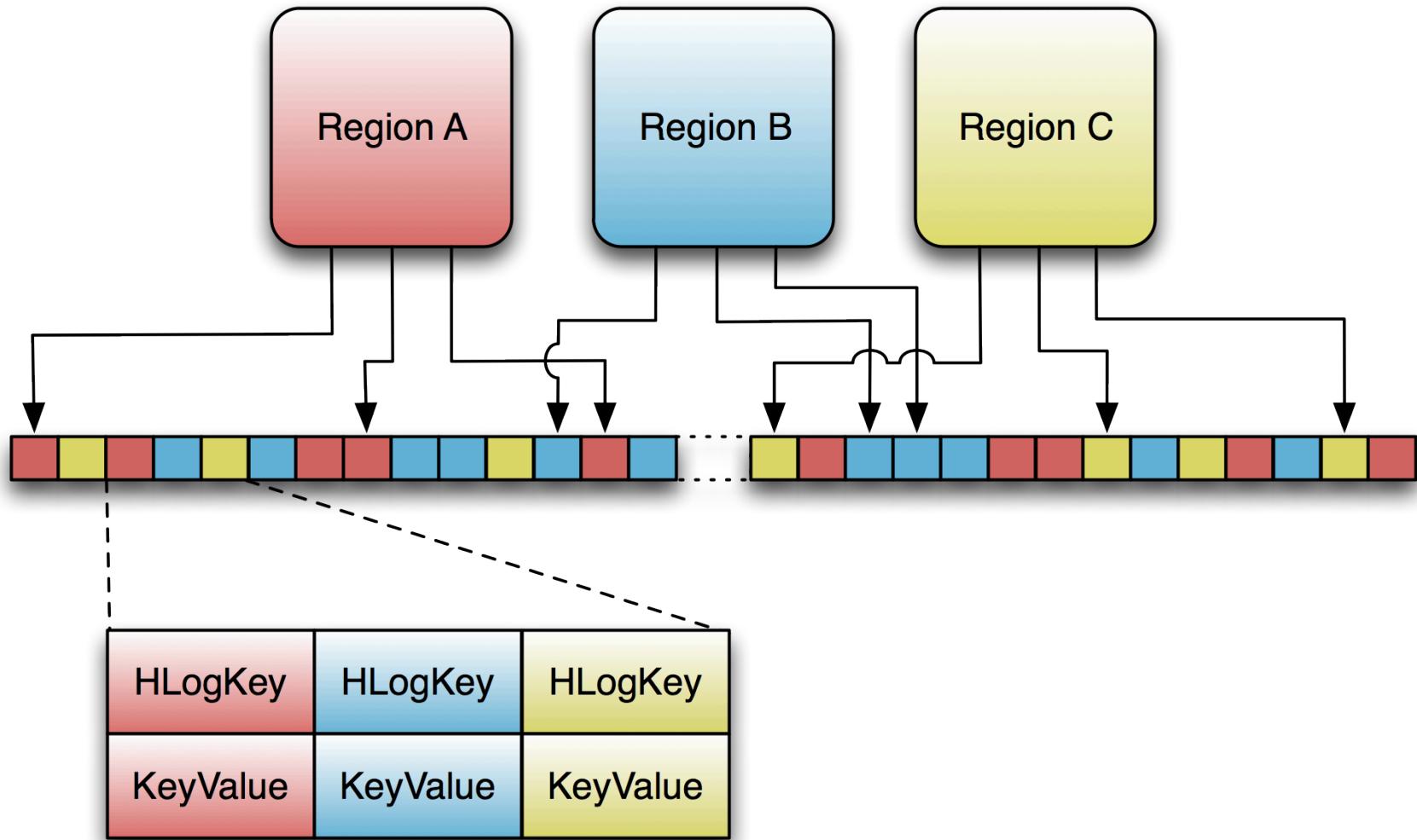


Write-Ahead Log (WAL) Fluss

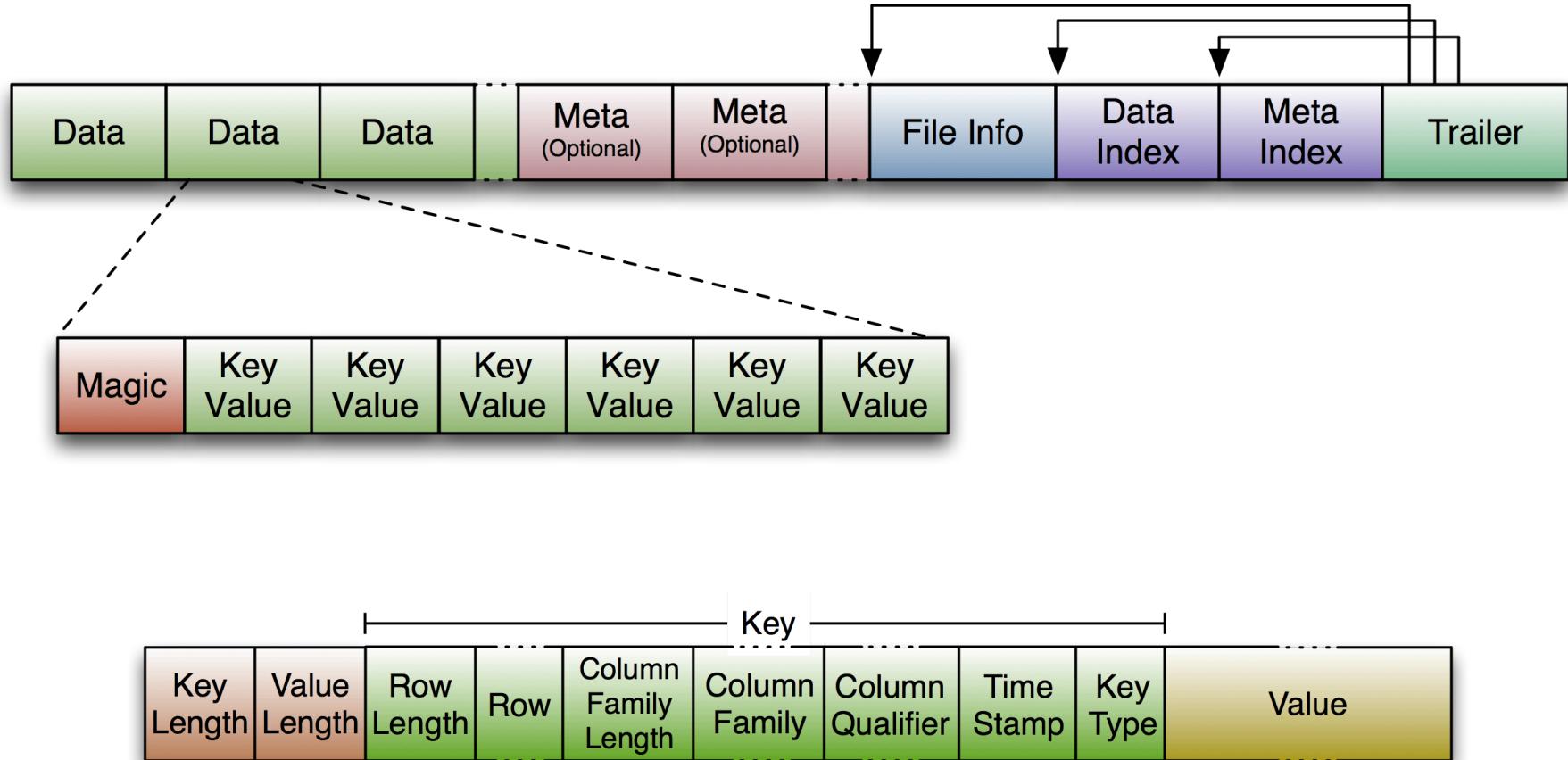




Write-Ahead Log (fort.)



HFile und KeyValue





Ansicht der Rohdaten

```
$ ./bin/hbase org.apache.hadoop.hbase.io.hfile.HFile -f file:///tmp/  
hbase-larsgeorge/hbase/testtable/272a63b23bdb5fae759be5192cab0ce/  
f1/4992515006010131591 -p  
  
K: row1/f1:/1290345071149/Put/vlen=6 V: value1  
K: row2/f1:/1290345078351/Put/vlen=6 V: value2  
K: row3/f1:/1290345089750/Put/vlen=6 V: value3  
K: row4/f1:/1290345095724/Put/vlen=6 V: value4  
K: row5/f1:c1/1290347447541/Put/vlen=6 V: value5  
K: row6/f1:c2/1290347461068/Put/vlen=6 V: value6  
K: row7/f1:c1/1290347581879/Put/vlen=7 V: value10  
K: row7/f1:c1/1290347469553/Put/vlen=6 V: value7  
K: row7/f1:c10/1290348157074/DeleteColumn/vlen=0 V:  
K: row7/f1:c10/1290347625771/Put/vlen=7 V: value11  
K: row7/f1:c11/1290347971849/Put/vlen=7 V: value14  
K: row7/f1:c12/1290347979559/Put/vlen=7 V: value15  
K: row7/f1:c13/1290347986384/Put/vlen=7 V: value16  
K: row7/f1:c2/1290347569785/Put/vlen=6 V: value8  
K: row7/f1:c3/1290347575521/Put/vlen=6 V: value9  
K: row7/f1:c8/1290347638008/Put/vlen=7 V: value13  
K: row7/f1:c9/1290347632777/Put/vlen=7 V: value12
```



MemStores

- After data is written to the WAL the RegionServer saves KeyValues in **memory store**
- Flush to disk based on size, see
hbase.hregion.memstore.flush.size
- Default size is **64MB**
- Uses **snapshot** mechanism to write flush to disk while still serving from it and accepting new data at the same time
- Snapshots are released when flush has succeeded



Block Cache

- Acts as very large, in-memory **distributed cache**
- Assigned a large part of the JVM **heap** in the RegionServer process, see *hfile.block.cache.size*
- Optimizes **reads** on subsequent columns and rows
- Has **priority** to keep “in-memory” column families in cache

```
if(inMemory) {  
    this.priority = BlockPriority.MEMORY;  
} else {  
    this.priority = BlockPriority.SINGLE;  
}
```

- Cache needs to be used properly to get best read performance
 - Turn off block cache on operations that cause large churn
 - Store related data “close” to each other
- Uses **LRU** cache with threaded (asynchronous) evictions based on priorities



Compactions

- General Concepts
 - Two types: **Minor** and **Major** Compactions
 - Asynchronous and transparent to client
 - Manage file bloat from MemStore flushes
- Minor Compactions
 - Combine last “few” flushes
 - Triggered by number of storage files
- Major Compactions
 - Rewrite **all** storage files
 - Drop deleted data and those values exceeding TTL and/or number of versions
 - Triggered by time threshold
 - Cannot be scheduled automatically starting at a specific time (bummer!)
 - May (most definitely) tax overall HDFS IO performance

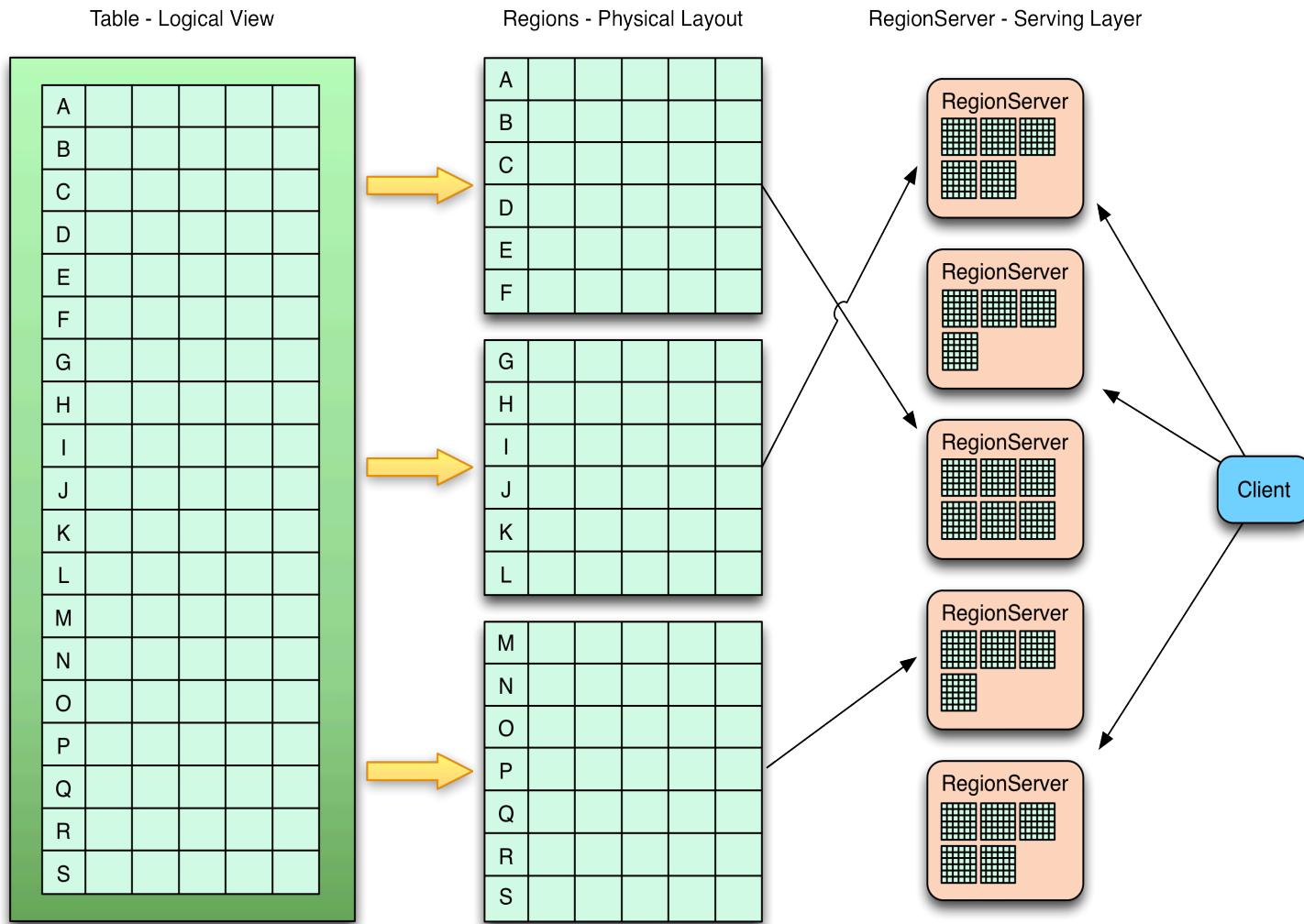
Tip: Disable major compactions and schedule to run manually (e.g. cron) at off-peak times



Region Splits

- Triggered by configured maximum file size of any store file
 - This is checked directly **after** the compaction call to ensure store files are actually approaching the threshold
- Runs as **asynchronous** thread on RegionServer
- Splits are **fast** and nearly instant
 - Reference files point to original region files and represent each half of the split
- Compactions take care of splitting original files into new region directories

Auto Sharding



Auto Sharding and Distribution



- Unit of scalability in HBase is the *Region*
- Sorted, contiguous range of rows
- Spread “randomly” across RegionServer
- Moved around for load balancing and failover
- Split automatically or manually to scale with growing data
- Capacity is solely a factor of cluster nodes vs. regions per node



Column Family vs. Column

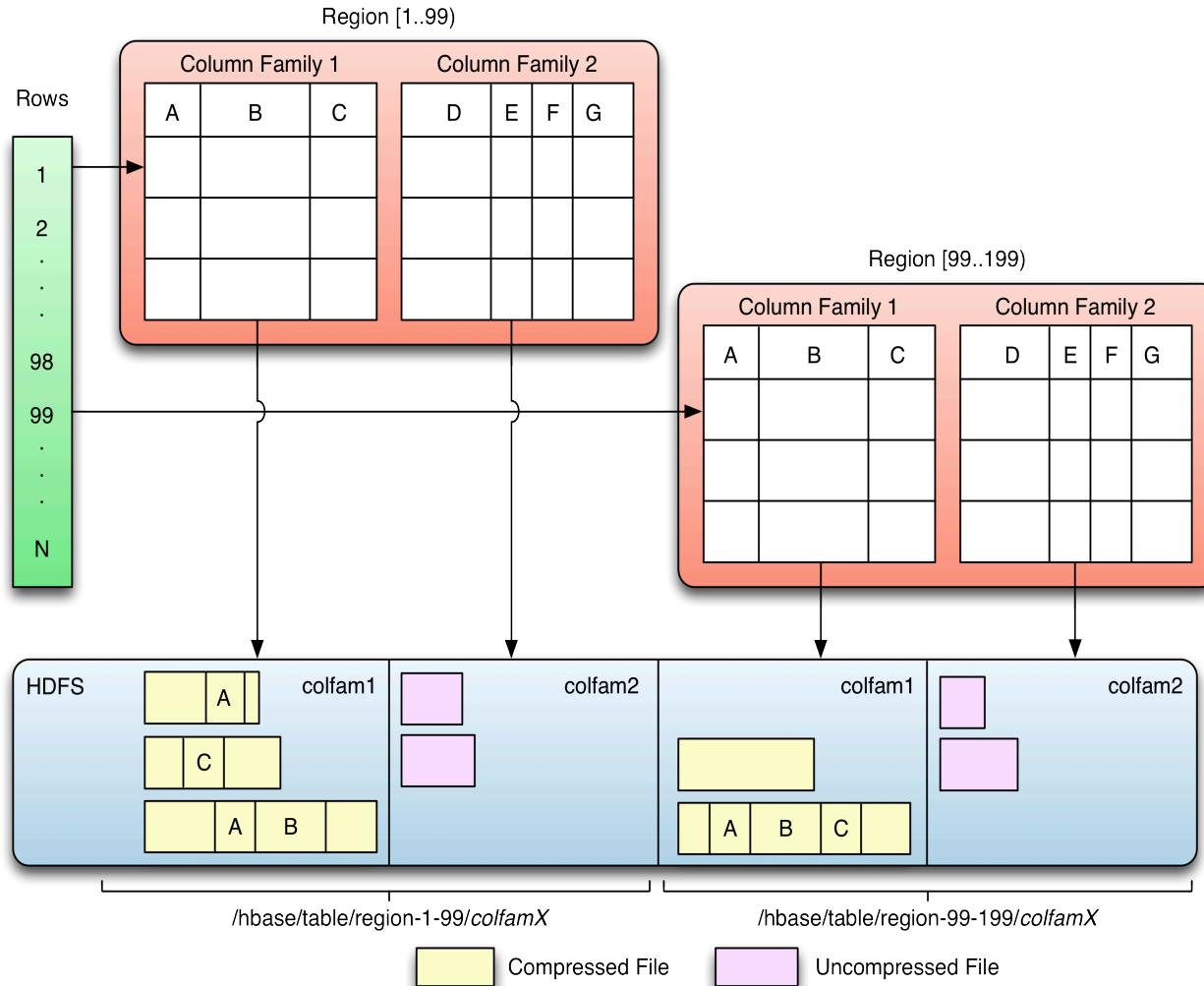
- Use only a few column families
 - Causes many files that need to stay open per region plus class overhead per family
- Best used when logical separation between data and meta columns
- Sorting per family can be used to convey application logic or access pattern



Storage Separation

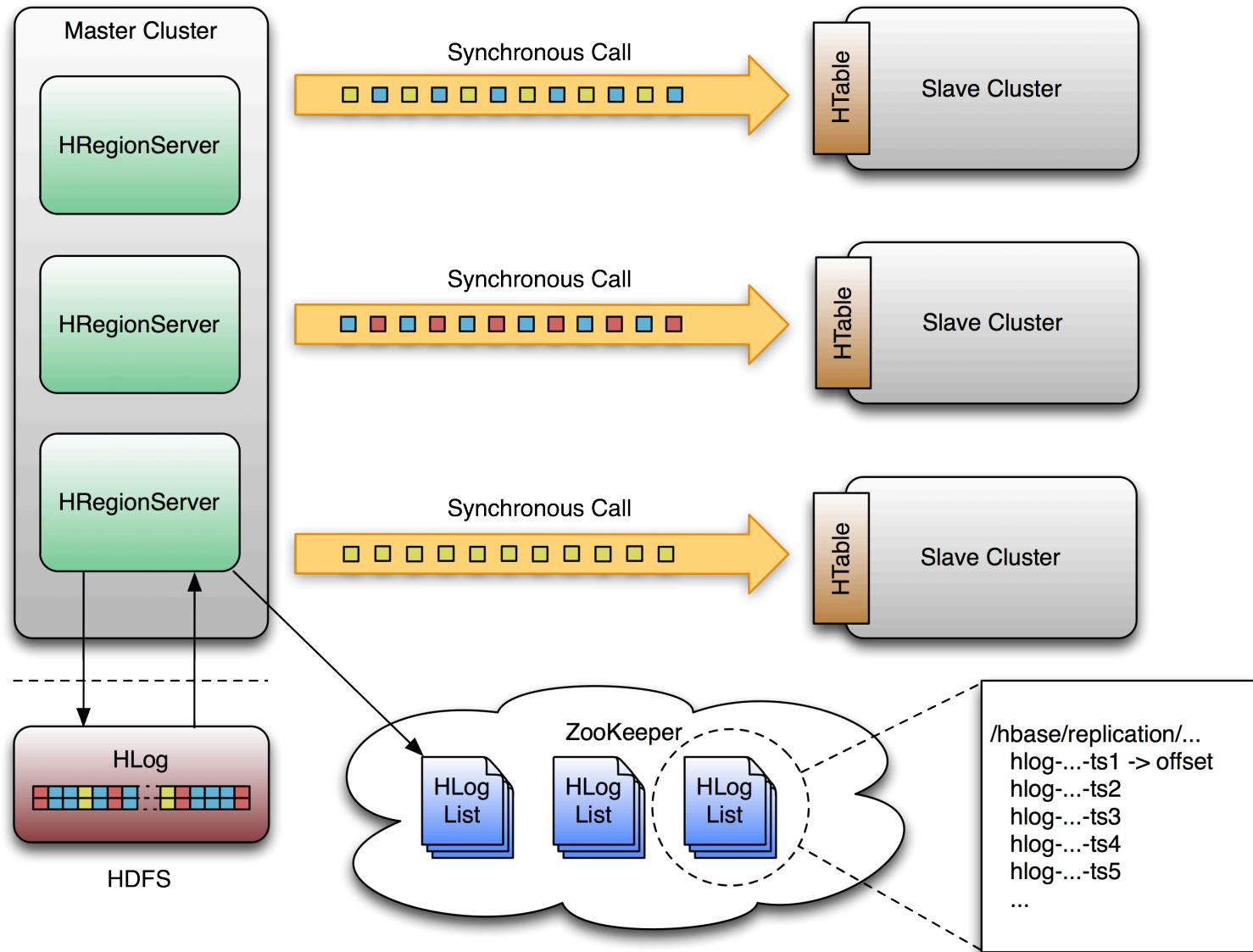
- Column Families allow for separation of data
 - Used by Columnar Databases for fast analytical queries, but on column level only
 - Allows different or no compression depending on the content type
- Segregate information based on access pattern
- Data is stored in one or more storage file, called HFiles

Column Families





Replication





Java API

- CRUD
 - get: retrieve an entire, or partial row (R)
 - put: create and update a row (CU)
 - delete: delete a cell, column, columns, or row (D)

```
Result get(Get get) throws IOException;
```

```
void put(Put put) throws IOException;
```

```
void delete(Delete delete) throws IOException;
```



Java API (cont.)

- CRUD+SI
 - scan: Scan any number of rows (S)
 - increment: Increment a column value (I)

```
ResultScanner getScanner(Scan scan) throws IOException;
```

```
Result increment(Increment increment) throws IOException ;
```



Java API (cont.)

- CRUD+SI+CAS
 - Atomic compare-and-swap (CAS)
- Combined get, check, and put operation
- Helps to overcome lack of full transactions



Batch Operations

- Support Get, Put, and Delete
- Reduce network round-trips
- If possible, batch operation to the server to gain better overall throughput

```
void batch(List<Row> actions, Object[] results)  
    throws IOException, InterruptedException;
```

```
Object[] batch(List<Row> actions)  
    throws IOException, InterruptedException;
```



Filters

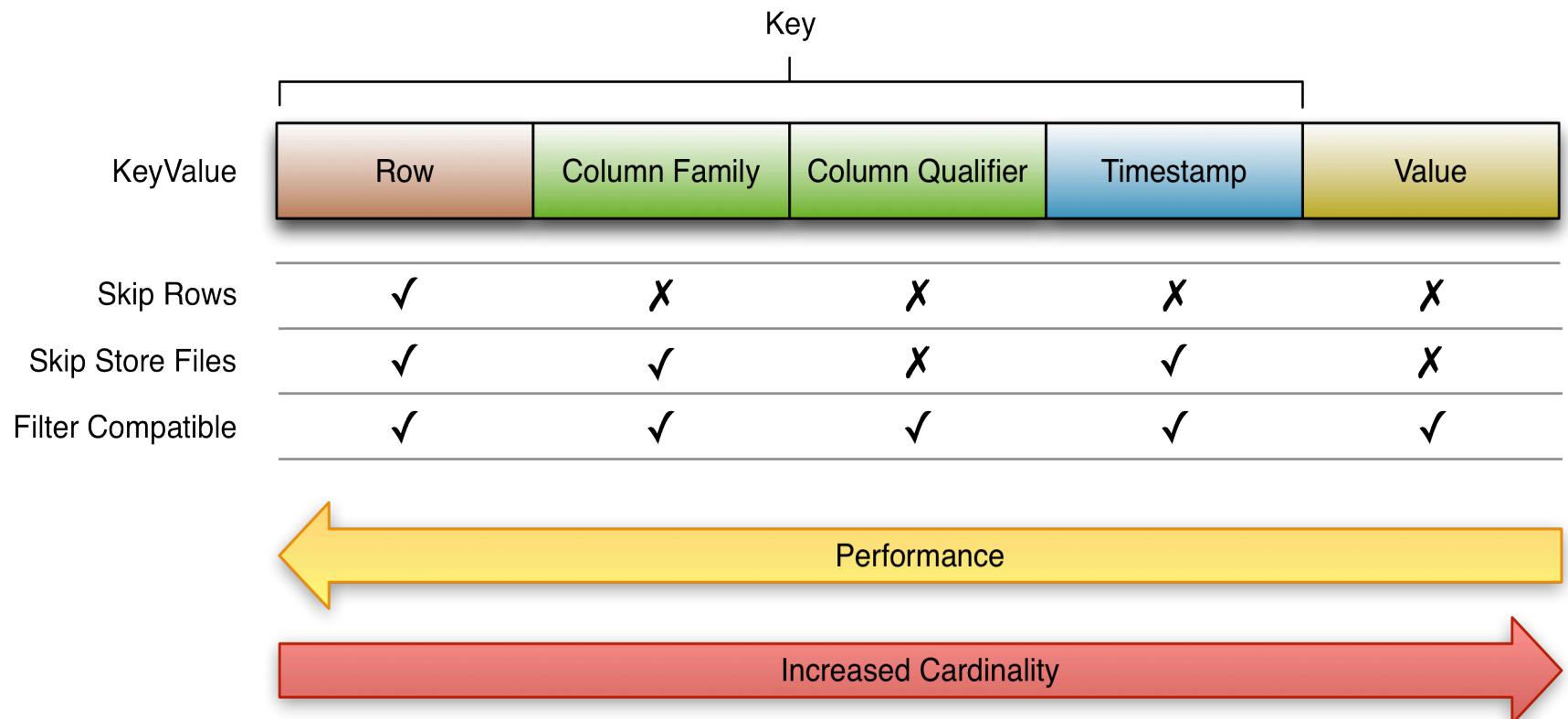
- Can be used with Get and Scan operations
- Server side hinting
- Reduce data transferred to client
- Filters are no guarantee for fast scans
 - Still full table scan in worst-case scenario
 - Might have to implement your own
- Filters can hint next row key



Advanced Techniques

- Key/Table Design
- DDI
- Salting
- Hashing vs. Sequential Keys
- ColumnFamily vs. Column
- Using BloomFilter
- Data Locality
- checkAndPut() and checkAndDelete()
- Coprocessors

Key Cardinality





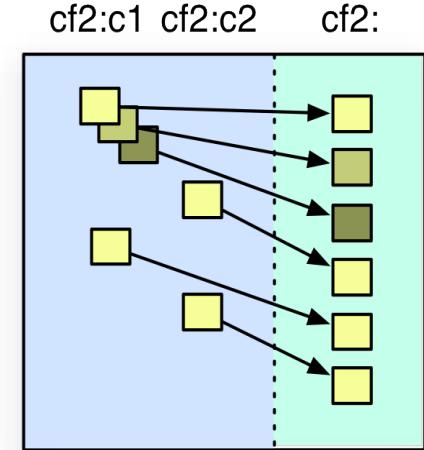
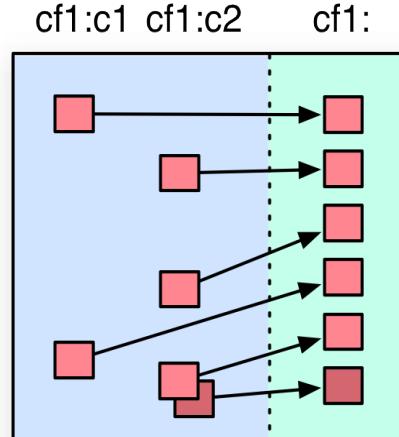
Key Cardinality

- The best performance is gained from using row keys
- Time range bound reads can skip store files
 - So can Bloom Filters
- Selecting column families reduces the amount of data to be scanned
- Pure value based filtering is a full table scan
 - Filters often are too, but reduce network traffic

Fold, Store, and Shift

	cf1:c1	cf1:c2	cf2:c1	cf2:c2
r1	█			
r2		█		
r3			█	
r4		█		█
r5	█		█	
r6		█		█

Fold



Store

r1 : cf1 : c1 : t1 : <value> \x00
 r1 : cf1 : c1-<value> : t1 : \x00
 r1-<value> : cf1 : c1 : t1 : \x00

 = Same Storage Requirements

Shift

r1 : cf1 : c1 : t1 : <value>
 r2 : cf1 : c2 : t1 : <value>
 r4 : cf1 : c2 : t1 : <value>
 r5 : cf1 : c1 : t1 : <value>
 r6 : cf1 : c2 : t2 : <value>
 r6 : cf1 : c2 : t1 : <value>

StoreFile "cf1/1234"

r3 : cf2 : c1 : t3 : <value>
 r3 : cf2 : c1 : t2 : <value>
 r3 : cf2 : c1 : t1 : <value>
 r4 : cf2 : c2 : t1 : <value>
 r5 : cf2 : c1 : t1 : <value>
 r6 : cf2 : c2 : t1 : <value>

StoreFile "cf2/5678"



Fold, Store, and Shift

- Logical layout does not match physical one
- All values are stored with the full coordinates, including: Row Key, Column Family, Column Qualifier, and Timestamp
- Folds columns into “row per column”
- NULLs are cost free as nothing is stored
- Versions are multiple “rows” in folded table



Key/Table Design

- Crucial to gain best performance
 - Why do I need to know? Well, you also need to know that RDBMS is only working well when columns are indexed and query plan is OK
- Absence of secondary indexes forces use of *row key* or *column name* sorting
- Transfer multiple indexes into one
 - Generate large table -> Good since fits architecture and spreads across cluster



DDI

- Stands for Denormalization, Duplication and Intelligent Keys
- Needed to overcome shortcomings of architecture
- Denormalization -> Replacement for JOINs
- Duplication -> Design for reads
- Intelligent Keys -> Implement indexing and sorting, optimize reads



Pre-materialize Everything

- Achieve one read per customer request if possible
- Otherwise keep at lowest number
- Reads between 10ms (cache miss) and 1ms (cache hit)
- Use MapReduce to compute exacts in batch
- Store and merge updates live
- Use incrementColumnValue

Motto: “Design for **Reads**”

Tall-Narrow vs. Flat-Wide Tables

- Rows do not split
 - Might end up with one row per region
- Same storage footprint
- Put more details into the row key
 - Sometimes *dummy* column only
 - Make use of partial key scans
- Tall with Scans, Wide with Gets
 - Atomicity only on row level
- Example: Large graphs, stored as adjacency matrix



Example: Mail Inbox

```
<userId> : <colfam> : <messageId> : <timestamp> : <email-message>
```

```
12345 : data : 5fc38314-e290-ae5da5fc375d : 1307097848 : "Hi Lars, ..."  
12345 : data : 725aae5f-d72e-f90f3f070419 : 1307099848 : "Welcome, and ..."  
12345 : data : cc6775b3-f249-c6dd2b1a7467 : 1307101848 : "To Whom It ..."  
12345 : data : dcbee495-6d5e-6ed48124632c : 1307103848 : "Hi, how are ..."
```

Or

```
12345-5fc38314-e290-ae5da5fc375d : data : 1307097848 : "Hi Lars, ..."  
12345-725aae5f-d72e-f90f3f070419 : data : 1307099848 : "Welcome, and ..."  
12345-cc6775b3-f249-c6dd2b1a7467 : data : 1307101848 : "To Whom It ..."  
12345-dcbee495-6d5e-6ed48124632c : data : 1307103848 : "Hi, how are ..."
```

→ Same Storage Requirements



Partial Key Scans

Key	Description
<userId>	Scan over all messages for a given user ID
<userId>-<date>	Scan over all messages on a given date for the given user ID
<userId>-<date>-<messageId>	Scan over all parts of a message for a given user ID and date
<userId>-<date>-<messageId>-<attachmentId>	Scan over all attachments of a message for a given user ID and date



Sequential Keys

```
<timestamp><more key>: {CF: {CQ: {TS : Val}}}
```

- Hotspotting on Regions: **bad!**
- Instead do one of the following:
 - Salting
 - Prefix `<timestamp>` with distributed value
 - Binning or bucketing rows across regions
 - Key field swap/promotion
 - Move `<more key>` before the timestamp (see OpenTSDB later)
 - Randomization
 - Move `<timestamp>` out of key



Salting

- Prefix row keys to gain spread
- Use well known or numbered prefixes
- Use modulo to spread across servers
- Enforce common data stay close to each other for subsequent scanning or MapReduce processing

0_rowkey1, 1_rowkey2, 2_rowkey3
0_rowkey4, 1_rowkey5, 2_rowkey6

- Sorted by prefix first

0_rowkey1
0_rowkey4
1_rowkey2
1_rowkey5

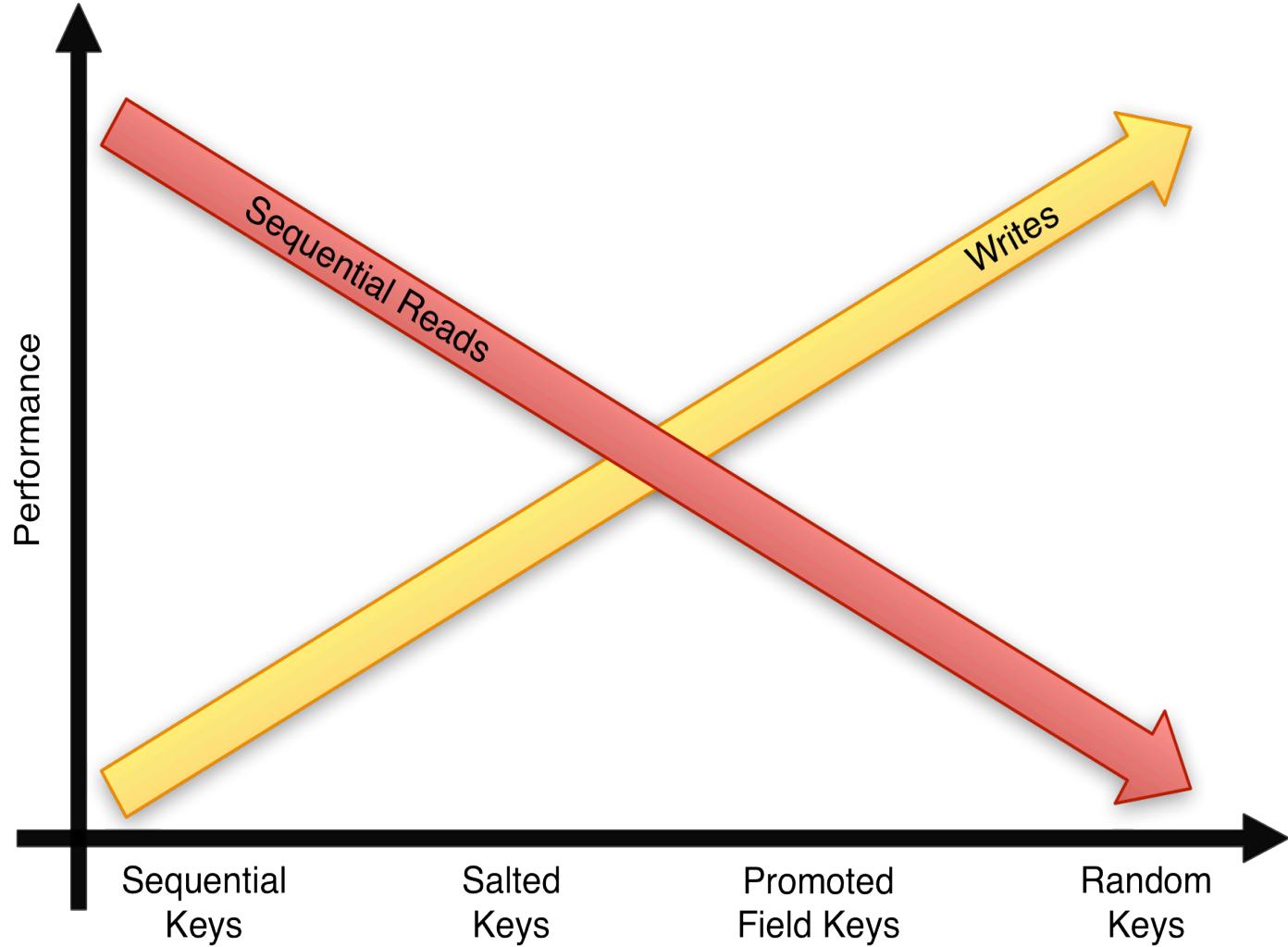
...



Hashing vs. Sequential Keys

- Uses hashes for best spread
 - Use for example MD5 to be able to recreate key
 - Key = MD5(customerID)
 - Counter productive for range scans
- Use sequential keys for locality
 - Makes use of block caches
 - May tax one server overly, may be avoided by salting or splitting regions while keeping them small

Key Design





Key Design Summary

- Based on access pattern, either use sequential or random keys
- Often a combination of both is needed
 - Overcome architectural limitations
- Neither is necessarily bad
 - Use bulk import for sequential keys and reads
 - Random keys are good for random access patterns



ColumnFamily vs. Column

- Use only a few column families
 - Causes many files that need to stay open per region plus class overhead per family
- Best used when logical separation between data and meta columns
- Sorting per family can be used to convey application logic or access pattern
- Define compression or in-memory attributes to optimize access and performance



Using Bloomfilters

- Defines a filter that allows to determine if a store file does **not** contain a row or column
- Error rate can control overhead but is usually very low, 1% or less
- Stored with each storage file on flush and compactions
- Good for large regions with many distinct row keys and many expected misses
- Trick: “Optimize” compaction to gain advantage while scanning files



Data Locality

- Provided by DFSClient
- Transparent for Hbase
- After restart, data may not be local
 - Work is done to improve on this
- Over time and caused by compactions data is stored where it is needed, i.e. local to RegionServer
- Could enforce major compaction before starting MapReduce jobs

checkAndPut() and checkAndDelete()

- Helps with atomic operations on single row
- Absence of value is treated as check for non-existence

```
public boolean checkAndPut(final byte[] row,  
    final byte[] family, final byte[] qualifier,  
    final byte[] value, final Put put)
```

```
public boolean checkAndDelete(final byte[] row,  
    final byte[] family, final byte[] qualifier,  
    final byte[] value, final Delete delete)
```



Locks

- Locks can be set explicitly for client operations
- Lock a row from modifications by other clients
 - Clients block on locked rows → keep locking reasonably short!
- Use HTable's *lockRow* to acquire and *unlockRow* to release
- Locks are guarded by leases on RegionServer and configured with *hbase.regionserver.lease.period*
 - By default set to 60 seconds
 - Leases are refreshed by any mutation call, e.g. `get()`, `put()` or `delete()`.



Coprocessors

- New addition to feature set
- Based on talk by Jeff Dean at LADIS 2009
 - Run arbitrary code on each region in RegionServer
 - High level call interface for clients
 - Calls are addressed to rows or ranges of rows while Coprocessors client library resolves locations
 - Calls to multiple rows are atomically split
 - Provides model for distributed services
 - Automatic scaling, load balancing, request routing



Coprocessors in HBase

- Use for efficient computational parallelism
- Secondary indexing (HBASE-2038)
- Column Aggregates (HBASE-1512)
 - SQL-like sum(), avg(), max(), min(), etc.
- Access control (HBASE-3025, HBASE-3045)
 - Provide basic access control
- Table Metacolumns
- New filtering
 - predicate pushdown
- Table/Region access statistics
- HLog extensions (HBASE-3257)



Summary

- Design for Use-Case
 - Read, Write, or Both?
- Avoid Hotspotting
- Consider using IDs instead of full text
- Leverage Column Family to HFile relation
- Shift details to appropriate position
 - Composite Keys
 - Column Qualifiers



Summary (cont.)

- Schema design is a combination of
 - Designing the keys (row and column)
 - Segregate data into column families
 - Choose compression and block sizes
- Similar techniques are needed to scale most systems
 - Add indexes, partition data, consistent hashing
- Denormalization, Duplication, and Intelligent Keys (DDI)



Comparison with RDBMS

- Very simple example use-case
 - Please note: **not** an example of how to implement this with HBase necessarily
- System to store a shopping cart
 - Customers, Products, Orders



Simple SQL Schema

```
CREATE TABLE customers (
    customerid UUID PRIMARY KEY,
    name TEXT, email TEXT)
```

```
CREATE TABLE products (
    productid UUID PRIMARY KEY,
    name TEXT, price DOUBLE)
```

```
CREATE TABLE orders (
    orderid UUID PRIMARY KEY,
    customerid UUID INDEXED REFERENCES(customers.customerid),
    date TIMESTAMP, total DOUBLE)
```

```
CREATE TABLE orderproducts (
    orderid UUID INDEXED REFERENCES(orders.orderid),
    productid UUID REFERENCES(products.productid))
```



Simple HBase Schema

CREATE TABLE customers (content, orders)

CREATE TABLE products (content)

CREATE TABLE orders (content, products)



Efficient Queries with Both

- Get name, email, orders for customers
- Get name, price for product
- Get customer, stamp, total for order
- Get list of products in order



Where SQL Makes Life Easy

- Joining
 - In a single query, get all products in an order with their product information
- Secondary Indexing
 - Get customerid by email
- Referential Integrity
 - Deleting an order would delete links out of ‘orderproducts’
 - ID updates propagate
- Realtime Analysis
 - GROUP BY and ORDER BY allow for simple statistical analysis

Where HBase Makes Life Easy



- Dataset Scale
 - We have 1M customers and 100M products
 - Product information includes large text datasheet or PDF files
 - Want to track every time a customer looks at a product page
- Read/Write Scale
 - Tables distributed across nodes means reads/writes are fully distributed
 - Writes are extremely fast and require no index updates
- Replication
 - Comes for free
- Batch Analysis
 - Massive and convoluted SQL queries executed serially become efficient MapReduce jobs distributed and executed in parallel



Conclusion

- For small instances of simple/straightforward systems, relational databases offer a much more convenient way to model and access data
 - Can outsource most work to transaction and query engine
 - HBase will force you to pull complexity into Application layer
- Once you need to scale, the properties and flexibility of HBase can relieve you from the headaches associated with scaling an RDBMS



Einheit 5

- Rückblick auf Einheit 4
- NoSQL „Datenbanken“ im Überblick
 - Grundlagen
 - Gründe für NoSQL
 - Zugrundeliegende Annahmen
 - Entwurfsmuster
 - Fragenkatalog
- **HBase als Beispiel**



Example: Facebook Insights

- > 20B Events per Day
- 1M Counter Updates per Second
 - 100 Nodes Cluster
 - 10K OPS per Node
- “Like” button triggers AJAX request
- Event written to log file
- 30mins current for website owner
 - Web → Scribe → Ptail → Puma → HBase



HBase Counters

- Store counters per Domain and per URL
 - Leverage HBase *increment* (atomic read-modify-write) feature
- Each row is one specific Domain or URL
- The columns are the counters for specific metrics
- Column families are used to group counters by time range
 - Set time-to-live on CF level to auto-expire counters by age to save space, e.g., 2 weeks on “Daily Counters” family



Key Design

- **Reversed Domains**
 - Examples: “com.cloudera.www”, “com.cloudera.blog”
 - Helps keeping pages *per site* close, as HBase efficiently scans blocks of sorted keys
- **Domain Row Key =**
MD5(Reversed Domain) + Reversed Domain
 - Leading MD5 hash spreads keys randomly across all regions for load balancing reasons
 - Only hashing the domain groups per site (and per subdomain if needed)
- **URL Row Key =**
MD5(Reversed Domain) + Reversed Domain + URL ID
 - Unique ID per URL already available, make use of it



Insights Schema

Row Key: *Domain Row Key*

Columns:

Hourly Counters CF					Daily Counters CF					Lifetime Counters CF				
6pm Total	6pm Male	6pm US	7pm	I/I Total	I/I Male	I/I US	2/I	Total	Male	Female	US	...
100	50	92	45		1000	320	670	990		10000	6780	3220	9900	

Row Key: *URL Row Key*

Columns:

Hourly Counters CF					Daily Counters CF					Lifetime Counters CF				
6pm Total	6pm Male	6pm US	7pm	I/I Total	I/I Male	I/I US	2/I	Total	Male	Female	US	...
10	5	9	4		100	20	70	99		100	8	92	100	



Beispiel: Hush

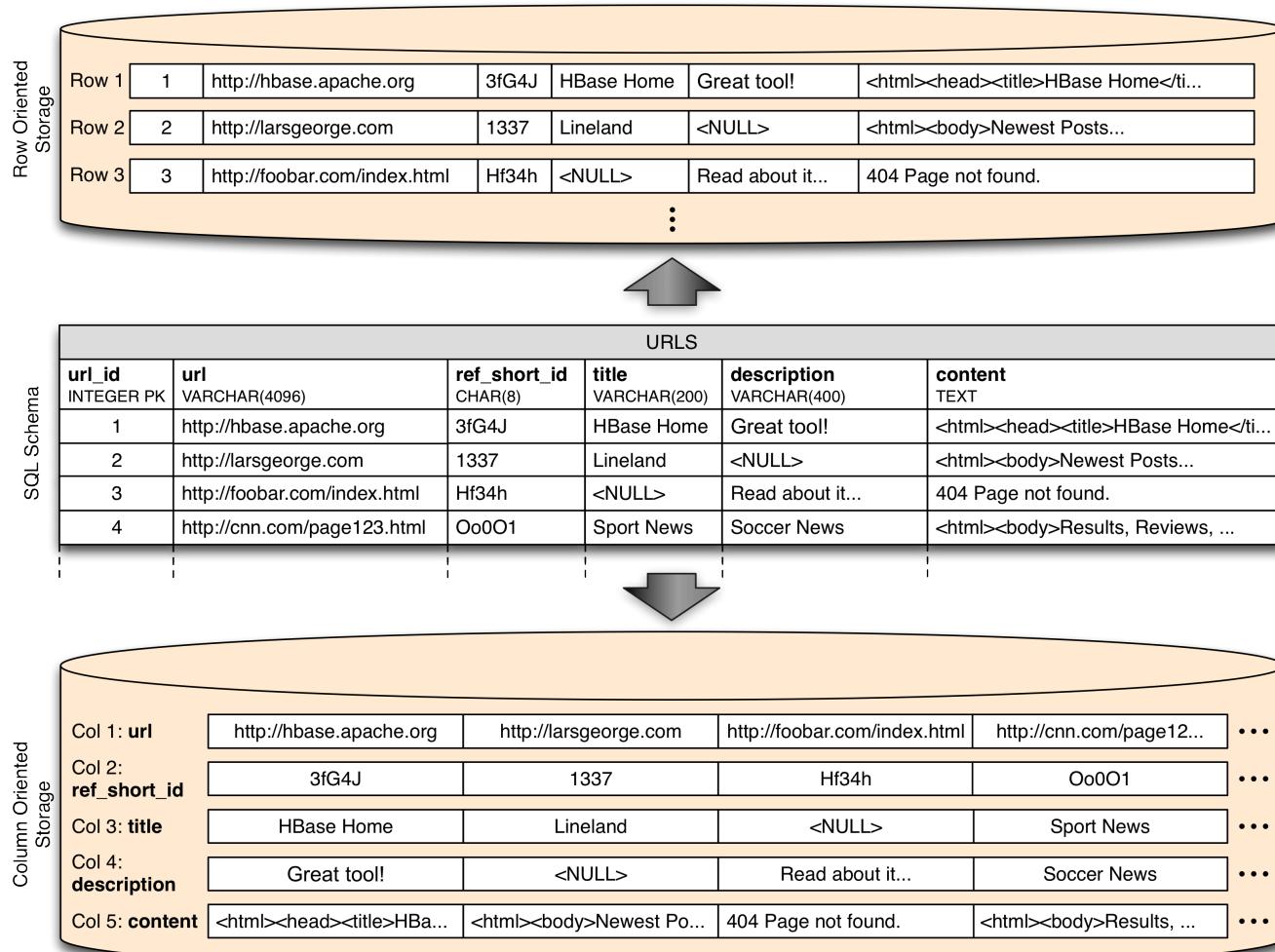
Table: shorturl		
Row Key:	shortId	
Family:	data:	Columns: url, refShortId, userId, clicks
	stats-daily: [ttl: 7days]	Columns: YYYYMMDD, YYYYMMDD\x00<country-code>
	stats-weekly: [ttl: 4weeks]	Columns: YYYYWW, YYYYWW\x00<country-code>
	stats-monthly: [ttl: 12months]	Columns: YYYYMN, YYYYMM\x00<country-code>

Table: url		
Row Key:	MD5(url)	
Family:	data: [compressed]	Columns: refShortId, title, description
	content: [compressed]	Columns: raw

Table: user-shorturl		
Row Key:	username\x00shortId	
Family:	data:	Columns: timestamp

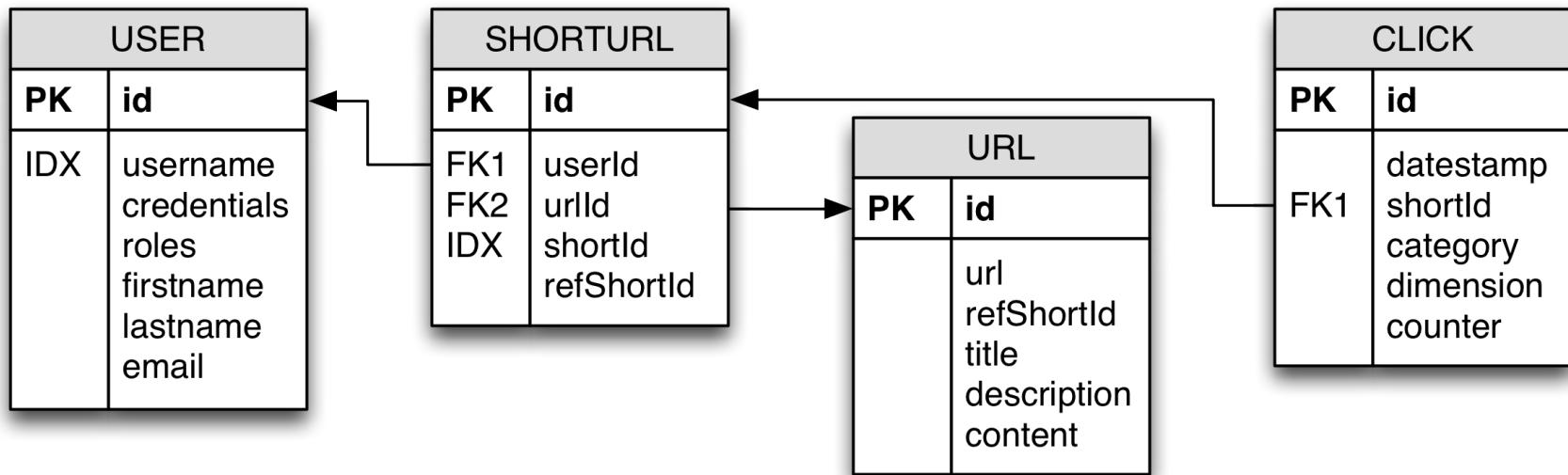
Table: user		
Row Key:	username	
Family:	data:	Columns: credentials, roles, firstname, lastname, email

Beispiel: Hush

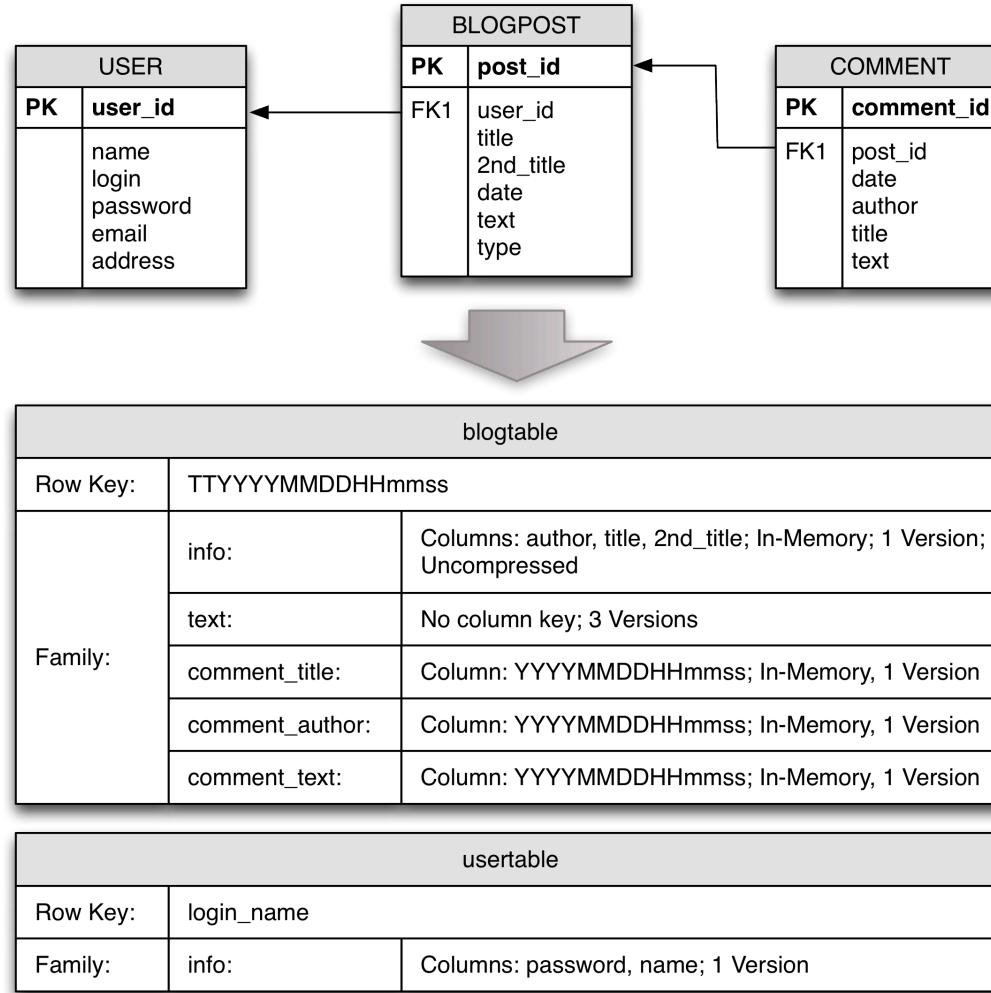




Beispiel: Hush



Beispiel: Blog Posts





Einheit 5

An dieser Stelle endet die fünfte Einheit, welche die NoSQL Speichersysteme vorstellt und am Beispiel von HBase vertieft.

In der nächsten Einheit befassen wir uns mit Datenpipelines und die komplexere Verarbeitung über mehrere Systeme hinweg.

Bis bald!



Übung 5

Ziele:

- NoSQL Tabellen in HBase erstellen, Daten laden und abfragen
- Hive oder Impala Anbindung an HBase testen



Übung 5

Code:

<https://github.com/larsgeorge/fh-muenster-bde-lesson-5>



Quellen

- SQL Grundlagen
 - Codd
 - http://en.wikipedia.org/wiki/Codd%27s_12_rules
 - http://en.wikipedia.org/wiki/Relational_model
- NoSQL Grundlagen
 - ACID, BASE, CAP
 - Arten und Formen
 - Ian Varley's Master Thesis
 - [http://ianvarley.com/UT/MR/
Varley_MastersReport_Full_2009-08-07.pdf](http://ianvarley.com/UT/MR/Varley_MastersReport_Full_2009-08-07.pdf)
- HBase
 - „HBase – The Definitive Guide“ von Lars George