NoSQL-Datenbanken

Wide Column Stores

Johannes Zschache
Sommersemester 2019

Abteilung Datenbanken, Universität Leipzig
http://dbs.uni-leipzig.de

Inhaltsverzeichnis: Wide Column Stores

Wide Column Stores

- Aka: Tabular Data Stores, Columnar Data Stores, Extensible Record Stores, Column Family Stores

- *Spezieller Key-Value Store*: **Sortiert** und **Indiziert**
  - Tabellenartige Struktur mit flexiblen relationalen Schema
  - Verschiedene Mengen von Attributen pro Reihe

- Skalierbar und fehlertolerant durch verteilte Datenspeicherung
- Geringe Latenzzeiten über direkte Lese- und Schreibzugriffe
- Unterstützung von Milliarden Zeilen, Millionen Spalten und Tausende Versionen pro Zelle

- *Grenzen:* keine Joins, referentielle Integrität, Datentypen, Transaktionen über mehrere Zeilen

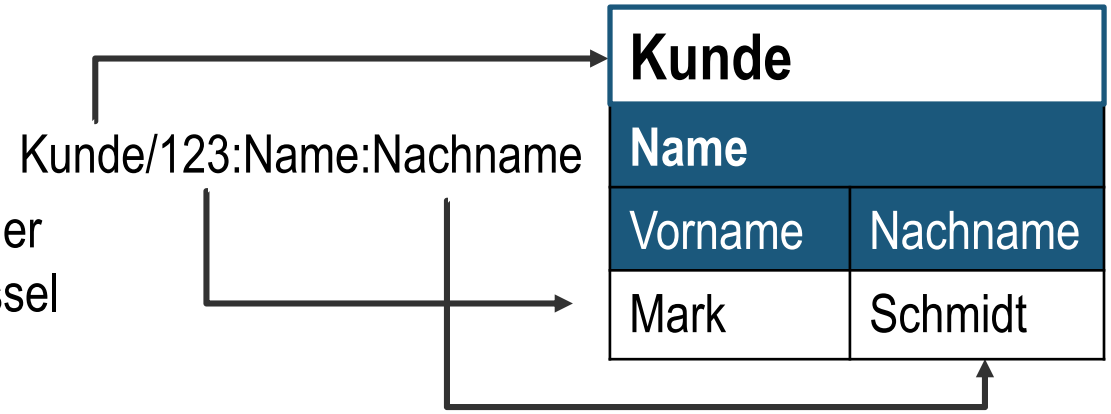
Datenmodell (1)

Kunde					
Namen		**Adressen**			
Vorname	Nachname	Straße_1	Stadt_1	Straße_2	Stadt_2
Mark	Schmidt	Hauptstr. 284	Leipzig	Nebenstr. 4	Berlin

- ***Namespace*** definiert „Tabelle“ (Tablet, Region), z.B. „Kunde“
- Namespace besteht aus mehreren ***Spaltenfamilien*** (Column Families)
  - Beispiel: Name, Adresse
  - Umfassen verwandte Spalten (ähnlichen Inhalts oder gleichen Typs)
  - *Spaltenschlüssel* = Spaltenfamilie:Spaltenname, z.B.Name:Nachname
  - Benachbarte Speicherung von Spalten einer Familie
  - Familien sind fest
  - Innerhalb Familie: flexible Erweiterbarkeit um neue Spalten
  - Wenige Spaltenfamilien (~100) mit unbegrenzter Anzahl an Spalten
- ***Spalte***: Menge von Zellen mit gleichen Spaltenschlüssel

Datenmodell (2)

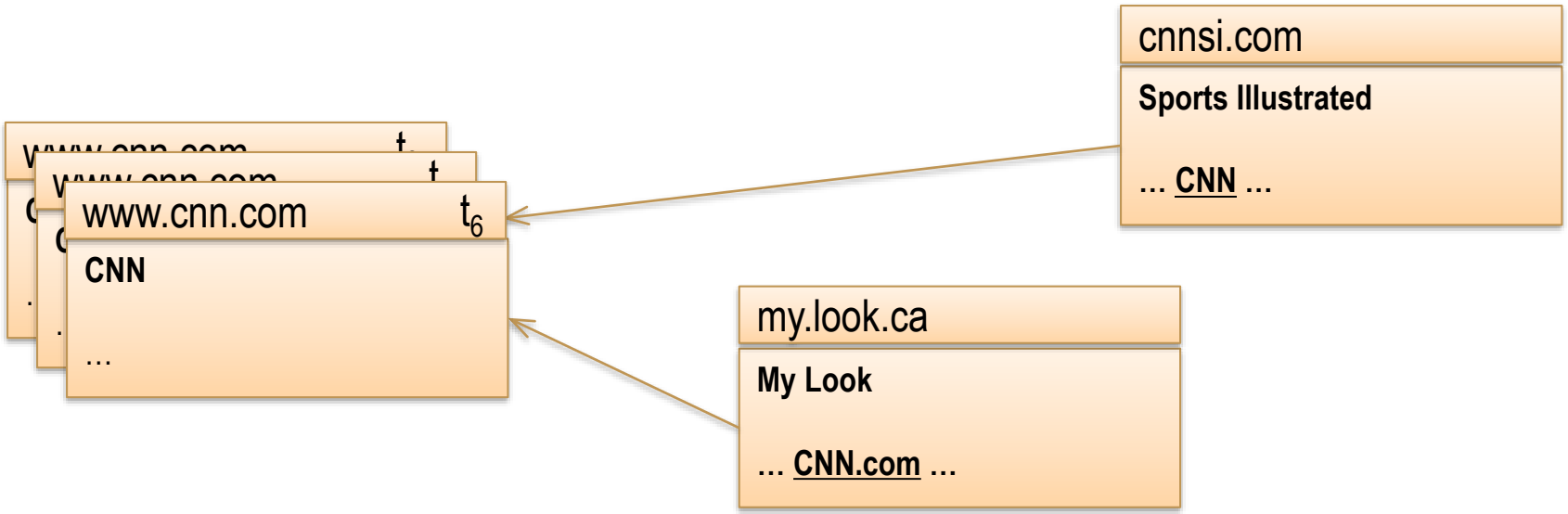
- ***Zelle***: Schlüssel-Wert-Paar
- Schlüssel: „Zeilenschlüssel:Spaltenfamilie:Spaltenname“
  - Ermöglicht schnellen direkten Datenzugriff
  - **Sortiert**: Speicherung der Daten z.B. in lexikographischer Reihenfolge der Zeilenschlüssel (je nach Implementierung)
  - **Indexiert** nach Zeilenschlüssel und Spaltenschlüssel



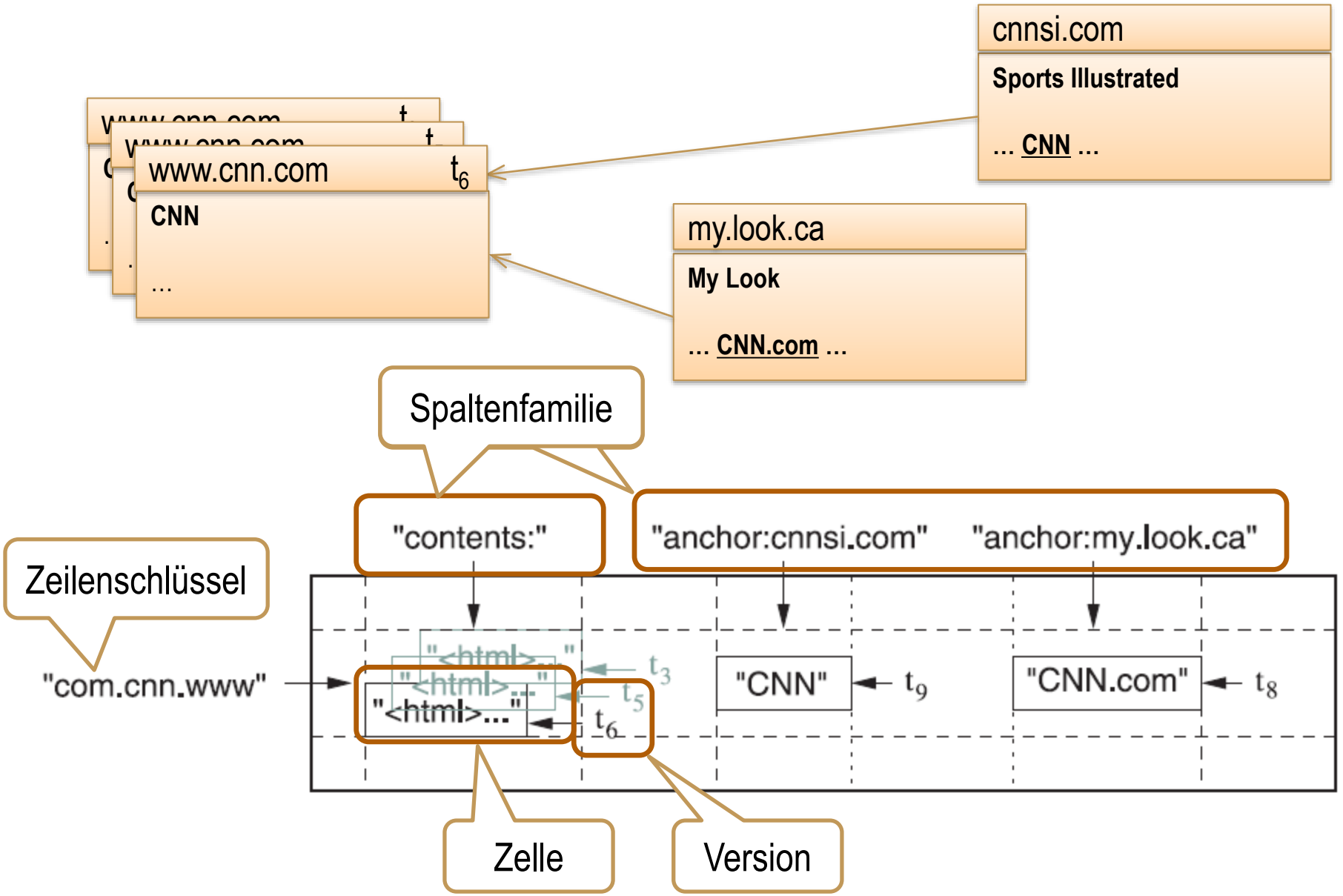
- ***Zeile***: Menge von Zellen mit gleichen Zeilenschlüssel
- Ggf. Zeitstempel
  - Mehrere Versionen pro Zelle
  - Automatische Versionierung
  - Festgelegte Versionszahl: automatisches Löschen älterer Daten

Beispiel: Web-Tabelle

- Daten: Webseiten und deren Verlinkungen
- Zeilenschlüssel: Webseiten-URL
- Szenarien
  - Direkter Zugriff durch Webcrawler zum Einfügen neuer/geänderter Webseiten
  - Batch-Auswertungen zum Aufbau eines Suchmaschinenindex und Ranking
  - Direkter Zugriff in Echtzeit für Suchmaschinennutzer, um Cache-Version von Webseiten zu erhalten



Beispiel: Web-Tabelle

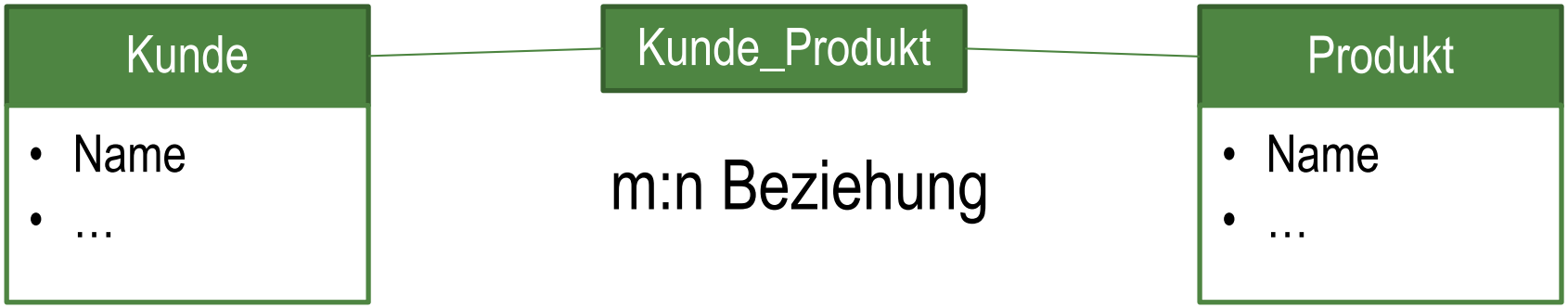


Inhaltsverzeichnis: Wide Column Stores

Designprinzipien

- Vermeidung von komplexen Datenstrukturen in Zellen (JSON, XML, …)
- Angemessene Anzahl an Versionen
- *Denormalisierung*
  - Eine Zeile pro Entität
  - Lange Zeilen aus vielen Spalten
  - Duplikate für effiziente Anfragen
  - Spaltennamen mit Werten
- *Vermeidung von „Hotspotting“ in Zeilenschlüsseln*
- *Verwendung von Indizes*

Design: Denormalisierung



Kunde:23

Kunde		Produkte			
Name	…	Produkt1	Produkt2	Produkt3	…
Mark Jones	…	38383	48284	48284	…

Prod:38383

Produkt		Kunden		
Name	…	Kunde1	Kunde2	…
Dell Laptop	…	23	43	…

Design: Denormalisierung

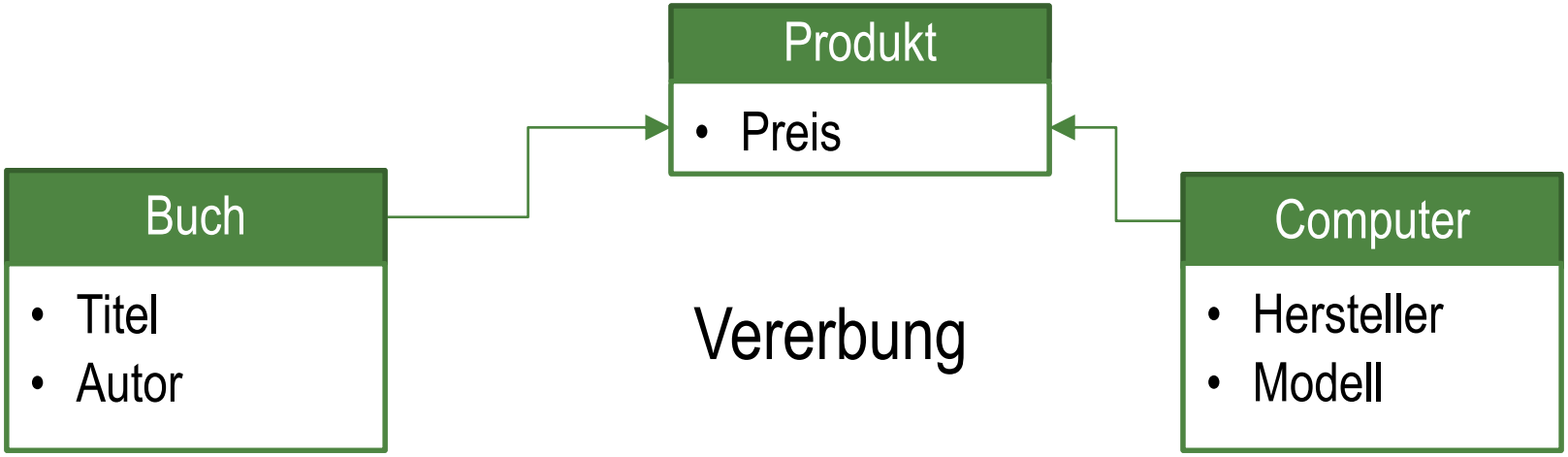
Kunde:23

Kunde		Produkte			
Name	…	38383	48284	48284	…
Mark Jones	…	Dell Laptop	Apple …	Galaxy …	

Prod:38383

Produkt		Kunden		
Name	…	23	43	…
Dell Laptop	…	Mark Jones	John Marc	…

Design: Denormalisierung



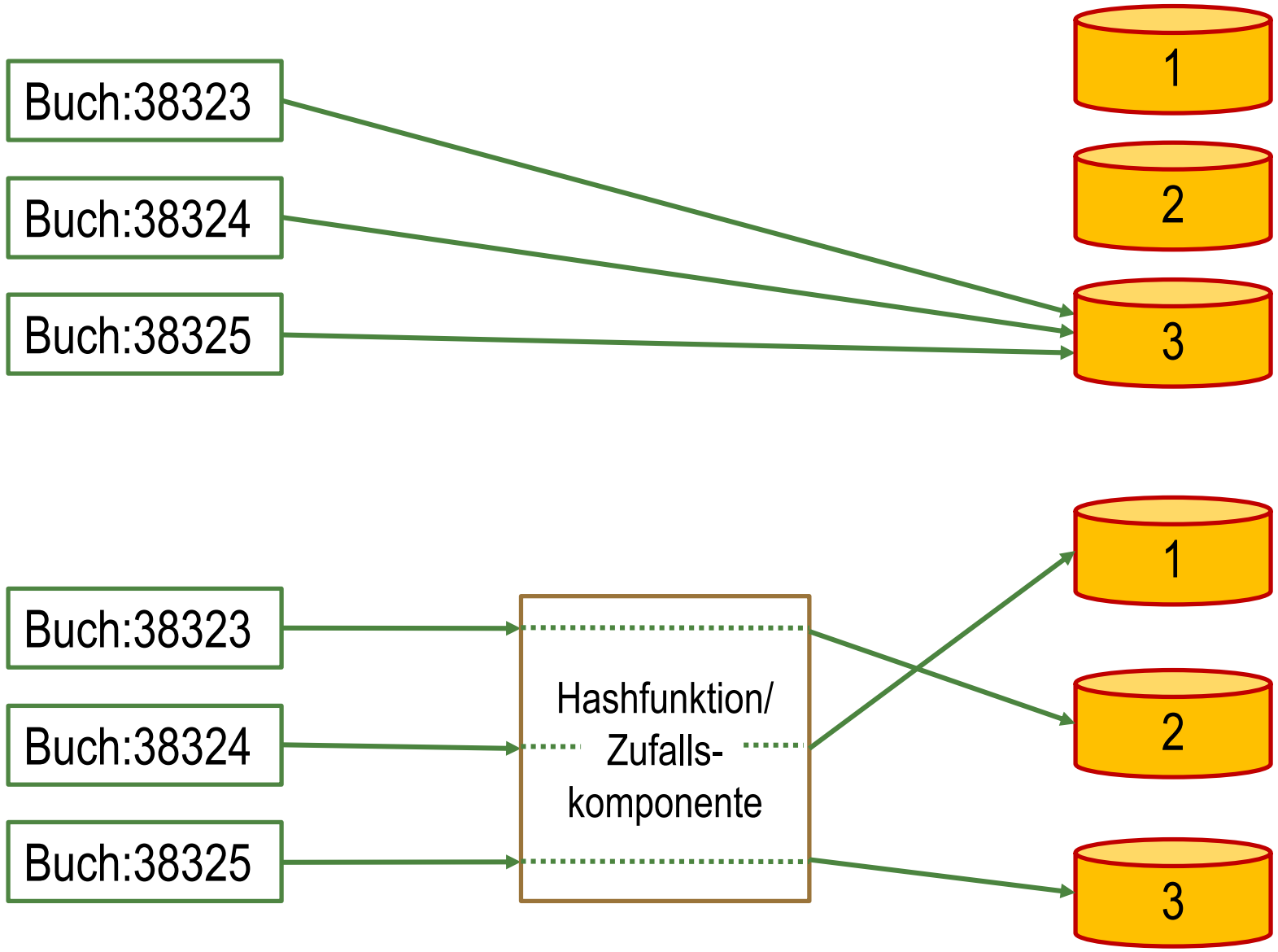
Prod:38323

Prod:38383

Details				Kunden	
Buch	**Preis**	**Titel**	**Autor**	**23**	**…**
	19.99 €	NoSQL	N. Sequel	Marc Jones	…

Computer	**Preis**	**Hersteller**	**Modell**	**43**	**…**
	299,95 €	Dell	Inspiron	John Marc	…

Design: Vermeidung von „Hotspots“



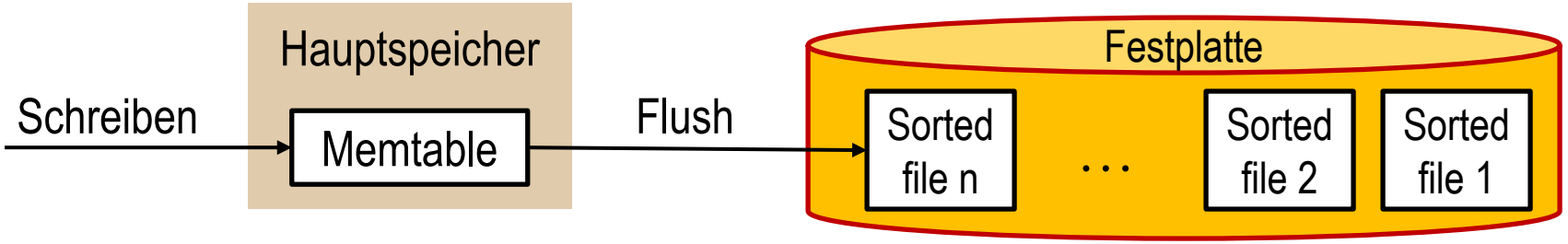
Design: Indizes

- Potentiell schnelleres Lesen vs. aufwendigeres Schreiben und höheren Speicherbedarf
- Primäre Indizes
  - Auf Zeilenschlüsseln
  - Verwaltet durch DBS
- Sekundäre Indizes
  - Auf beliebigen Attributen
  - Verwaltet durch DBS oder Anwendung
- Vermeidung, wenn
  - Sehr kleine Kardinalität des Wertebereichs (z.B. Ja/Nein bzw. 1/0)
  - Sehr große Kardinalität des Wertebereichs (z.B. Adressen)
  - Spärlich besetzte Attribute
- Realistische Testfälle um Nützlichkeit zu prüfen
- Verwendung der gleichen Datenstruktur bei manuellen Indizes

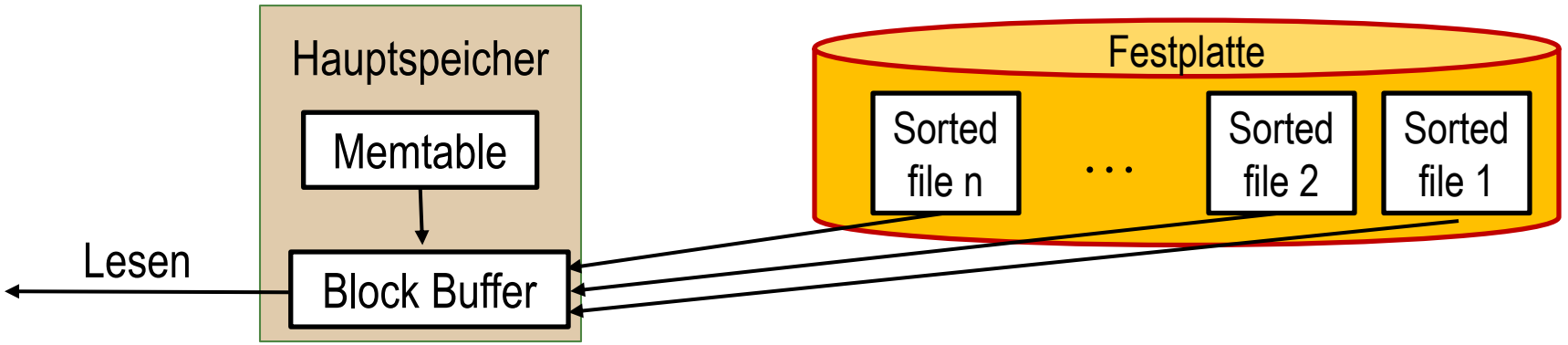
Inhaltsverzeichnis: Wide Column Stores

Unveränderliche Daten

- Optimierung des Schreibens durch Einfügen anstatt Überschreiben
  - Memtable: Buffer in Hauptspeicher; geschützt über „Write-ahead Log“ auf Festplatte
  - Sorted data files (SSTable): Sortiert (nach Schlüssel) und unveränderbar auf Festplatte
  - Löschen über „Tombstones“

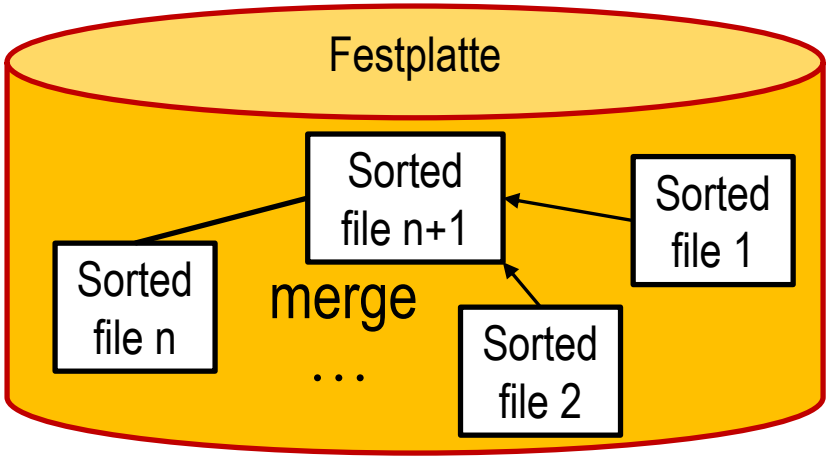


- Lesen erfordert Kombination der Daten

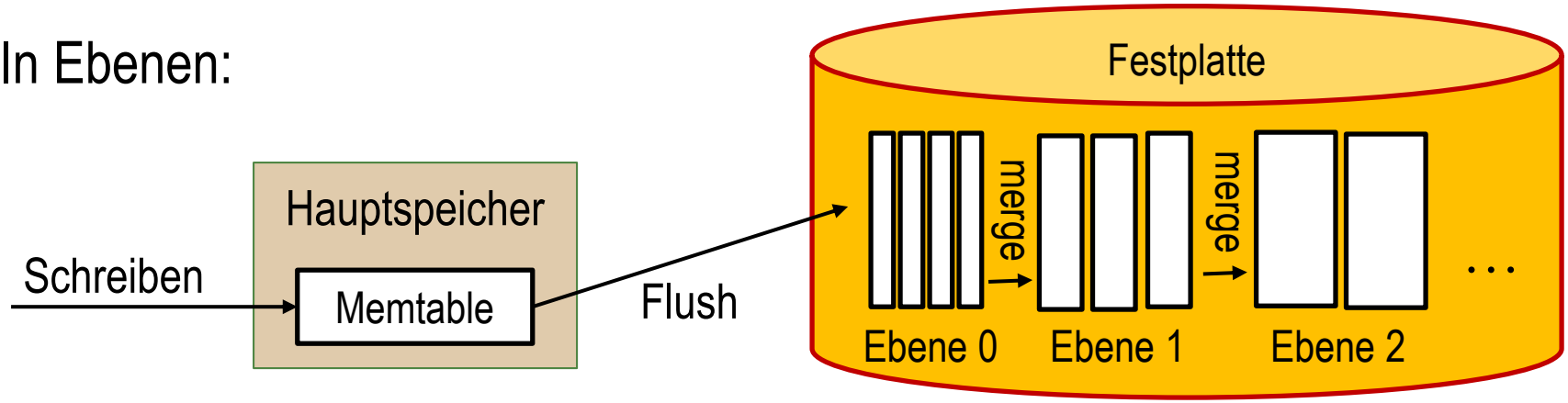


Kompaktierung

- Entfernen aktualisierter Einträge
- Auf Festplatte:

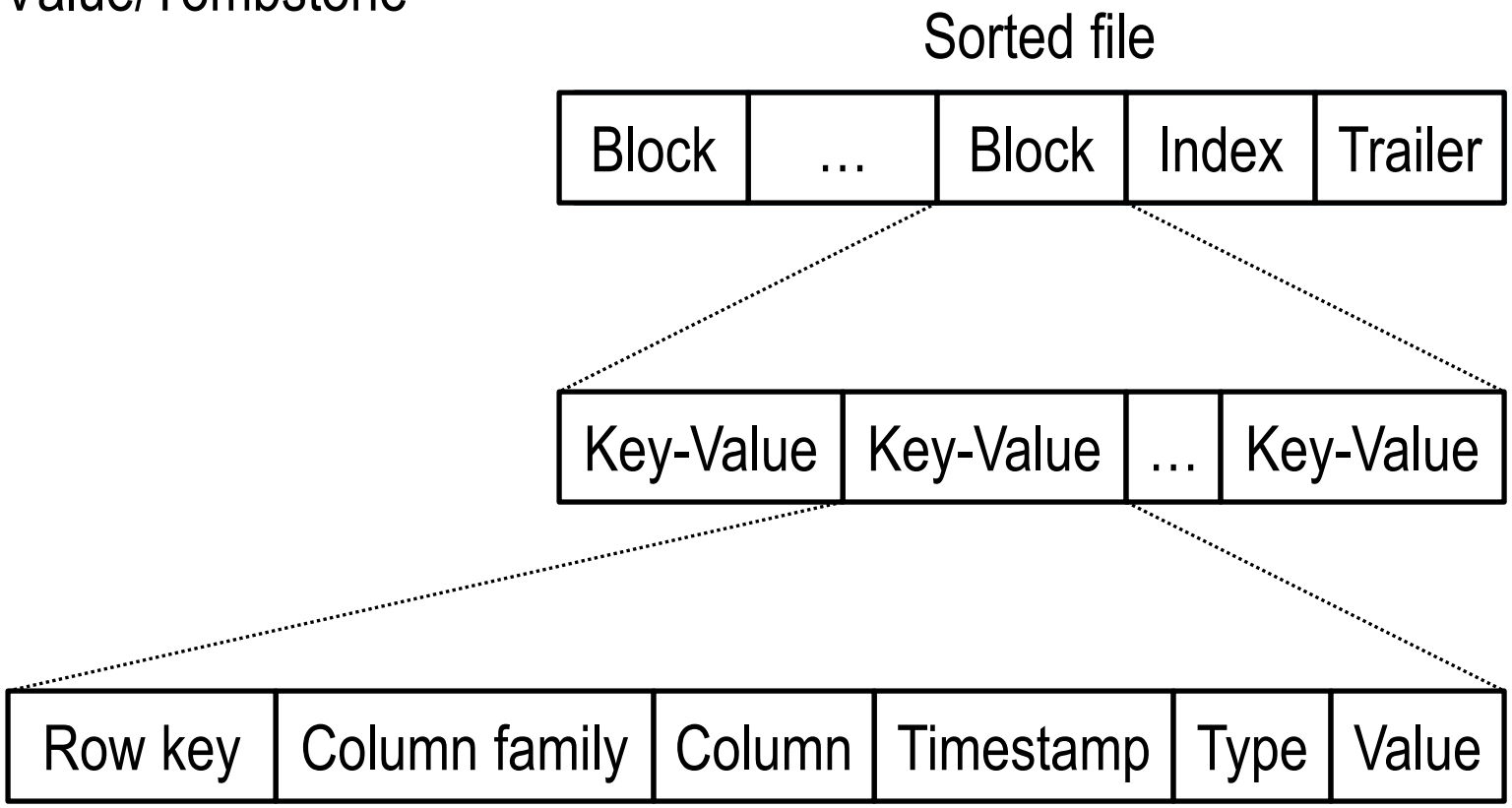


- In Ebenen:



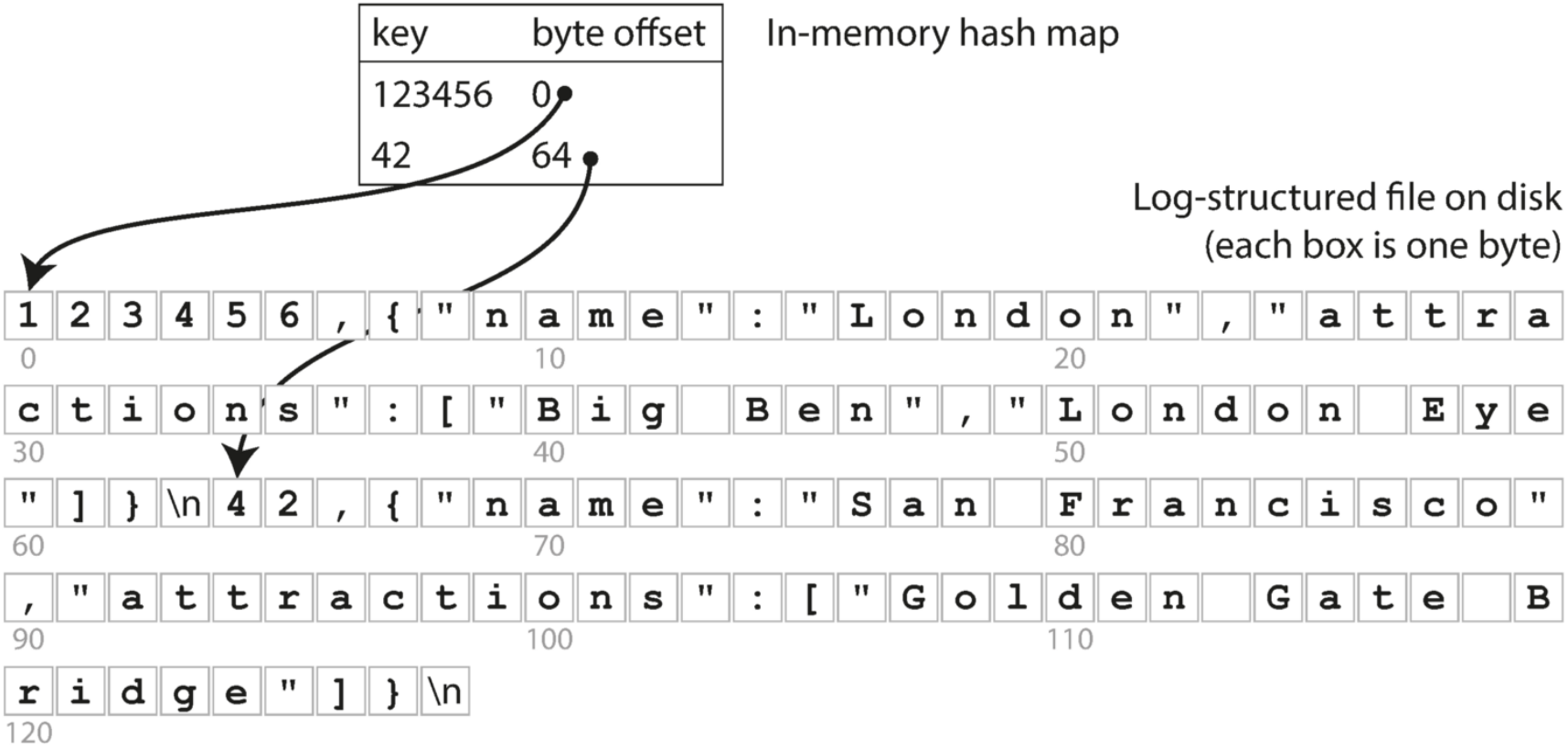
Dateiformat

- Blockgröße variabel
- Index für Blöcke (mehrstufiger Index möglich)
- Trailer: Metadaten (z.B. Anfang des Index)
- Type = Value/Tombstone



Hashtabelle (Index)

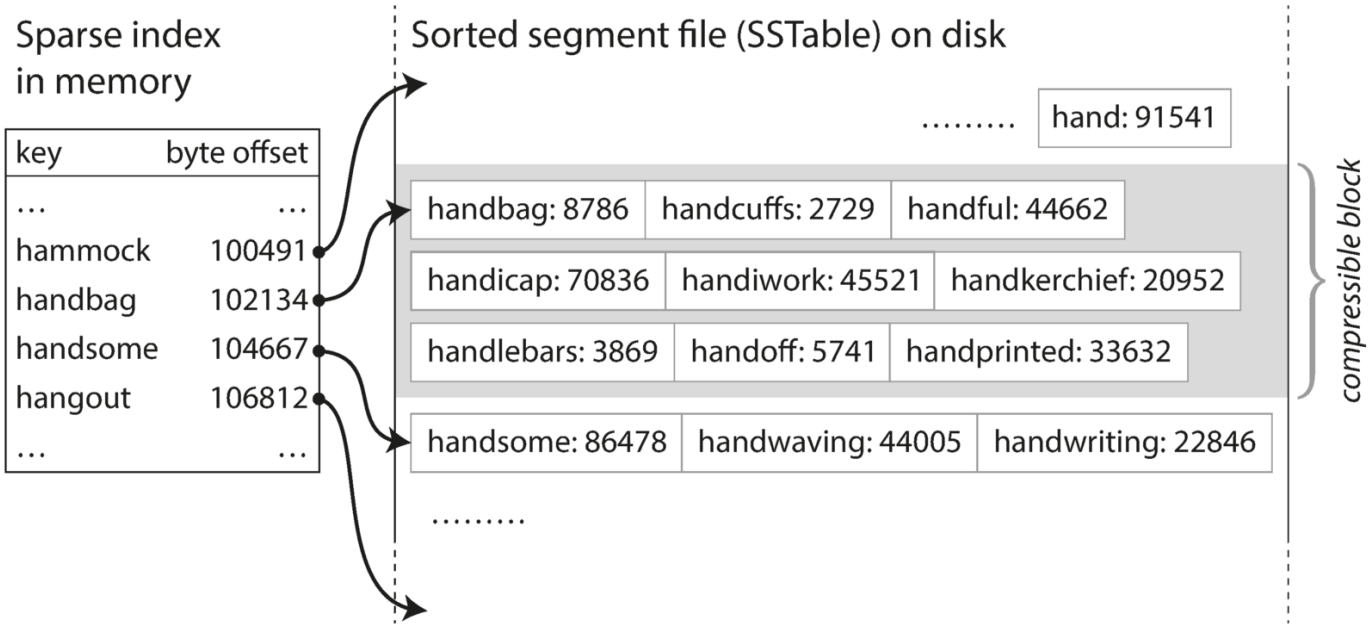
- In-Memory HashMap mit Byte-Offsets als Werten
- Beispiel: Riak (BitCask)



Quelle: https://medium.com/@shashankbaravani/database-storage-engines-under-the-hood-705418dc0e35

Sorted String Tables (SSTable)

- Hashtabelle mit sortierten Schlüsseln
- Vorteile:
  - Schnellere Kompaktierung (Merge)
  - Bereichsanfragen
  - Effektivere Kompression zur Reduzierung des Festplatten I/O

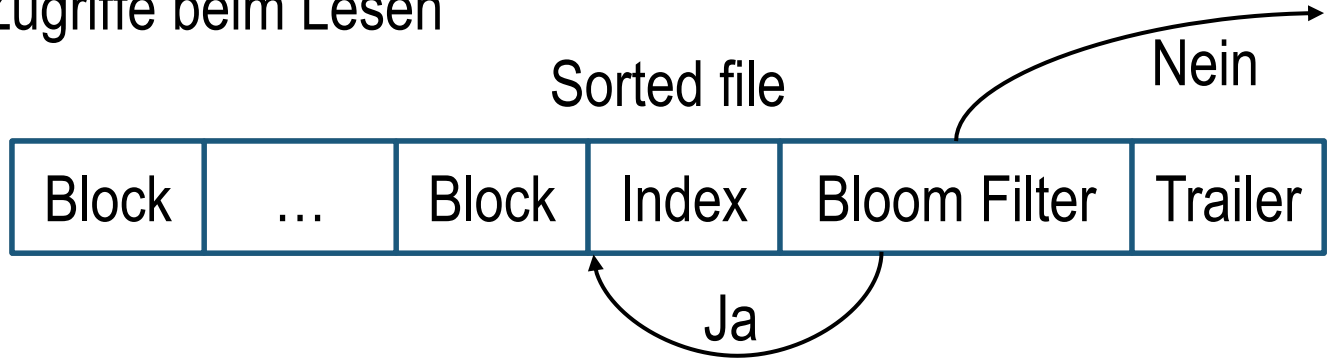


Quelle: https://medium.com/@shashankbaravani/database-storage-engines-under-the-hood-705418dc0e35

Inhaltsverzeichnis: Wide Column Stores

Bloom Filter

- Probabilistischer Test, ob ein Element in einer Menge enthalten ist
- Falsch positive Ergebnisse möglich
- Reduzierung der Zugriffe beim Lesen

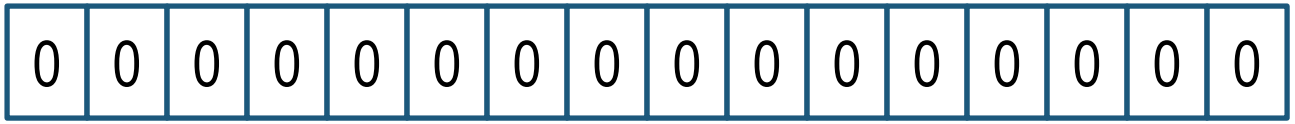


- Falls positives Ergebnis im Bloom Filter: Lade und durchsuche Index
- Falls negatives Ergebnis im Bloom Filter: Nächste Datei
- Bloom Filter: Bitvektor der Länge m und n Hash-Funktionen

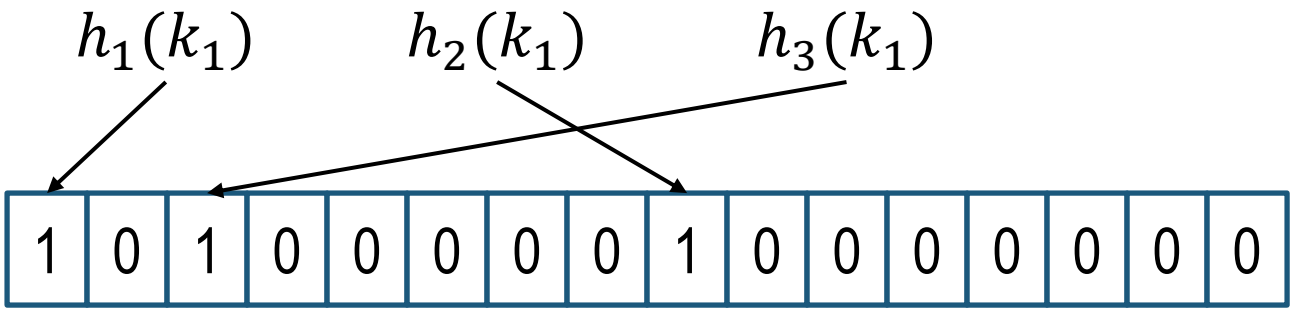
$$h_i : k \longmapsto \{1 \ldots m\}$$

Bloom Filter: Beispiel

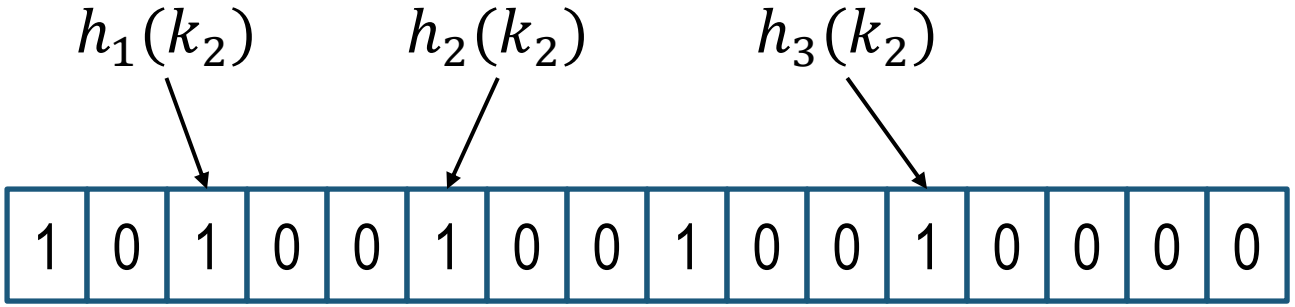
- m =16 und $n = 3$



- Schlüssel k_1

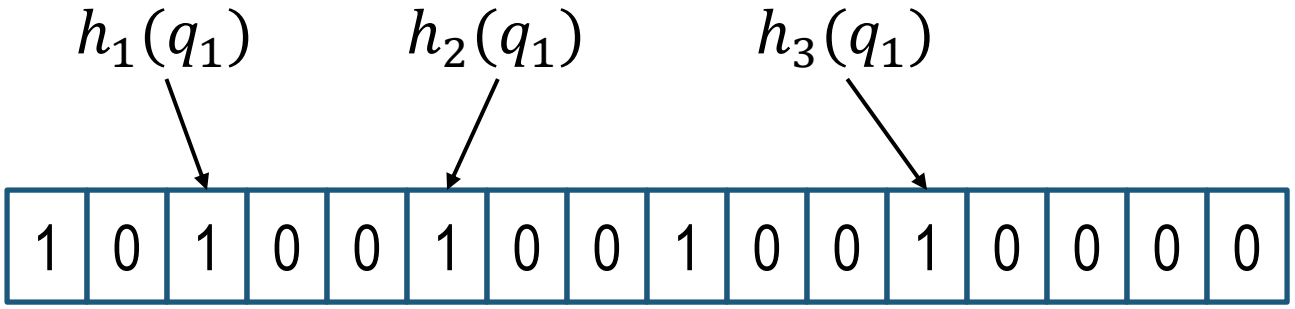


- Schlüssel k_2

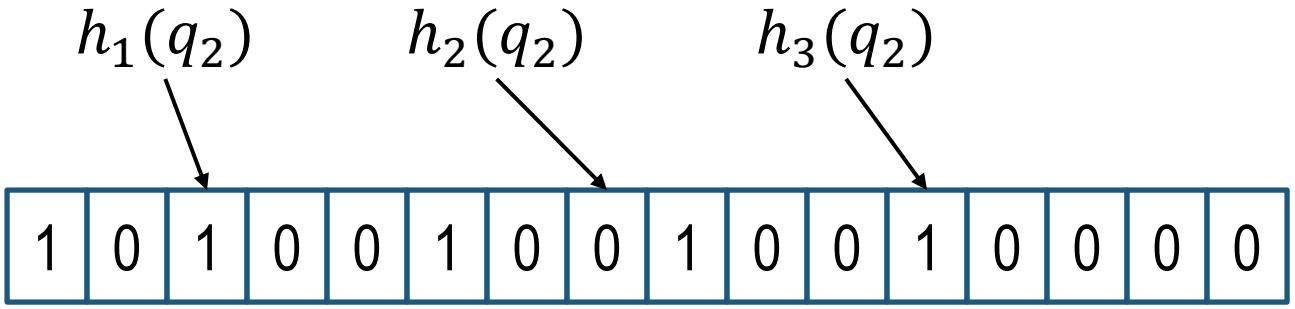


Bloom Filter: Richtige Entscheidungen

- Anfrage $q_1 = k_2$: Richtig positive Entscheidung

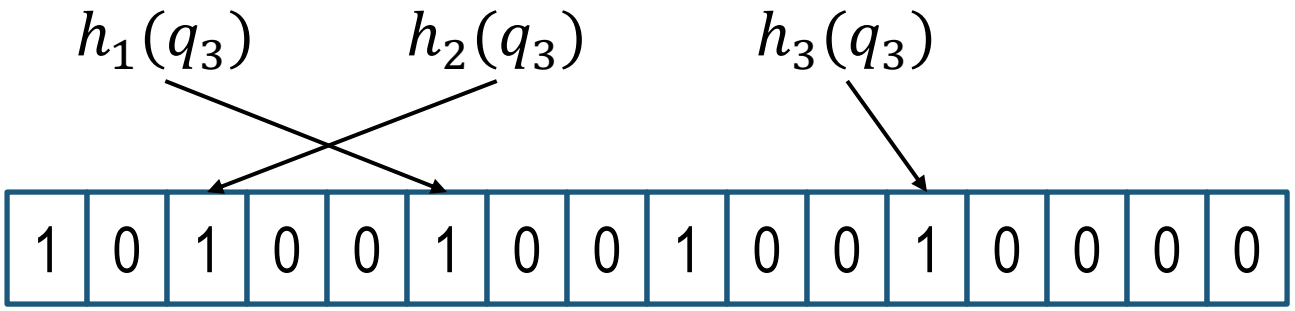


- Anfrage $q_2 \neq k_2$: Richtig negative Entscheidung



Bloom Filter: Falsche Entscheidung

- Anfrage $q_3 \neq k_2$: Falsch positive Entscheidung



- (approx.) Wahrscheinlichkeit, dass falsch positive Entscheidung nach x Eingaben:

$$p_{err} = \left(1 - e^{-nx/m}\right)^n$$

wobei ein Minimum durch $n_{opt} \approx \frac{9m}{13x}$ gegeben ist

$m = 4x$	p_{err}
$n = 1$	0.221
$n = 2$	0.155
$n = 3$	0.147
$n = 4$	0.160

Inhaltsverzeichnis: Wide Column Stores

DB Ranking

Quelle: http://db-engines.com/en/ranking/

12 systems in ranking, May 2019

Rank May 2019	Rank Apr 2019	Rank May 2018	DBMS	Database Model	Score May 2019	Score Apr 2019	Score May 2018
1.	1.	1.	Cassandra	Wide column	125.72	+2.11	+7.89
2.	2.	2.	HBase	Wide column	59.77	+1.11	-0.18
3.	3.	3.	Microsoft Azure Cosmos DB	Multi-model	27.59	+1.32	+10.06
4.	4.	4.	Datastax Enterprise	Wide column, Multi-model	9.59	+0.42	+2.21
5.	5.	↑ 6.	Microsoft Azure Table Storage	Wide column	4.42	-0.08	+1.02
6.	6.	↓ 5.	Accumulo	Wide column	4.17	+0.05	+0.24
7.	7.	7.	Google Cloud Bigtable	Wide column	2.06	+0.25	+1.21
8.	8.	↑ 9.	ScyllaDB	Wide column	1.55	-0.07	+1.08
9.	9.	↓ 8.	MapR-DB	Multi-model	0.74	-0.02	+0.18
10.	10.	10.	Sqrrl	Multi-model	0.60	+0.01	+0.27
11.	11.		YugaByte DB	Multi-model	0.38	+0.03	
12.	12.	↓ 11.	Alibaba Cloud Table Store	Wide column	0.20	-0.01	+0.20

Apache HBase

- HBase ist Hadoops open-source Implementierung von Googles BigTable

- DB für wirklich große Datensätze (mehrere GB)
- Horizontale Skalierbarkeit

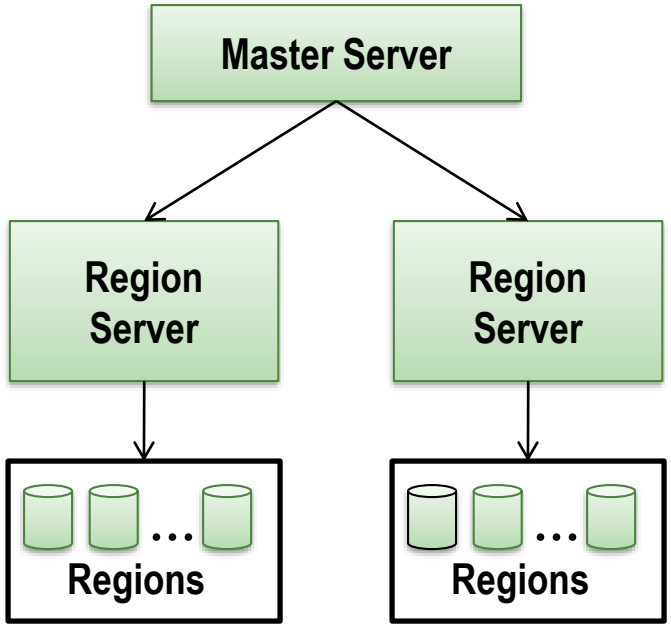
- Master-Slave Replikation
- Starke Konsistenzgarantien

BigTable	HBase
Tablet	Region
Master Server	HBase Master
Tablet Server	HBase Region Server
GFS	HDFS
SSTable File	MapFile

- Stärke: Durchsuchen riesiger Datensätze
- Anwendung: Grundlage für Logging oder Suche

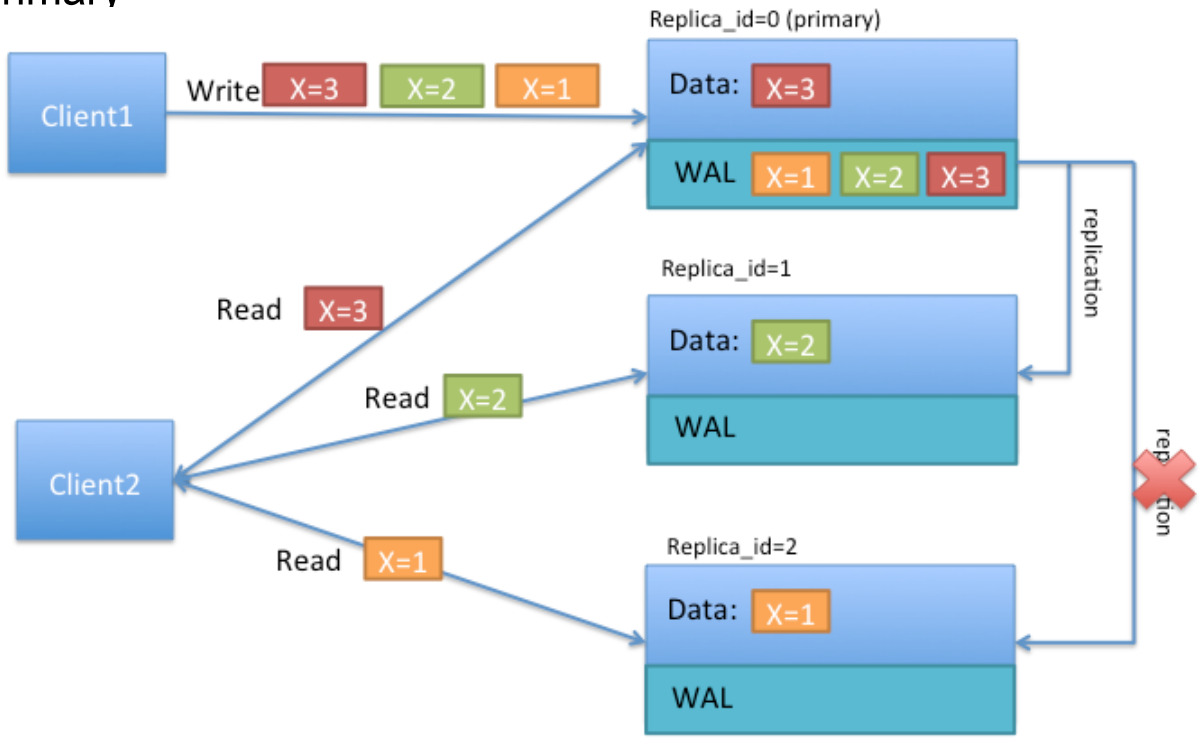
HBase: Architektur

- Datenpartitionierung
  - Zeilen in Tabelle lexikographisch sortiert nach Schlüssel
  - Horizontale Partitionierung von Tabellen in Regions (5-20GB Daten)
  - Verteilung der Regions auf mehrere RegionServer
- Master Server
  - Zuordnung: Region ↔ RegionServer
  - Wiederherstellung bei Ausfall eines RegionServer
- RegionServer
  - Verwaltung von 20-200 Regions
  - Koordination von Lese- und Schreibzugriffen
  - Region-Split bei zu großer Region
- Basiert auf
  - **HDFS:** Datenspeicher
  - **ZooKeeper:** Metadaten zur Lokalisierung der Server/Kataloge; Unterstützung der Zuordnung von Regions bei Serverausfällen



HBase: Konsistenz

- Konfigurierbar: Strong Consistency oder Timeline Consistency
- Strong: Schreib-/Leseoperationen nur auf Primary Region Server
- Timeline: Leseoperationen auch auf Secondary Region Server (Replikat)
  - Ermöglicht höhere Verfügbarkeit
  - Zuerst Leseversuch auf Primary
  - Falls Timeout: Anfrage an alle Secondary
  - Erstes Ergebnis wird zurückgegeben



Quelle: https://hbase.apache.org/images/timeline_consistency.png

HBase: Demo

- Installation: https://hbase.apache.org/book.html#quickstart

```
bin/start-hbase.sh
bin/hbase shell
status
```

- Beispiel aus Redmond & Wilson (2012): Erstellen eines Wikis
  - Erstellen eines Namespace 'wiki' mit Spaltenfamilie `'text'`

```
create 'wiki', 'text'
```

  - Daten schreiben (Zeilenschlüssel: 'Home')

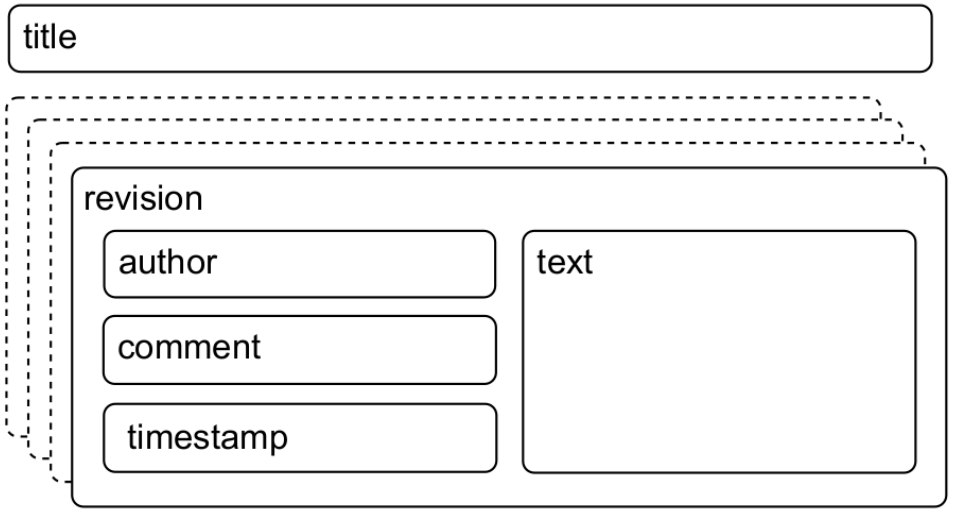
```
put 'wiki', 'Home', 'text:', 'Welcome to the wiki!'
```

  - Daten lesen

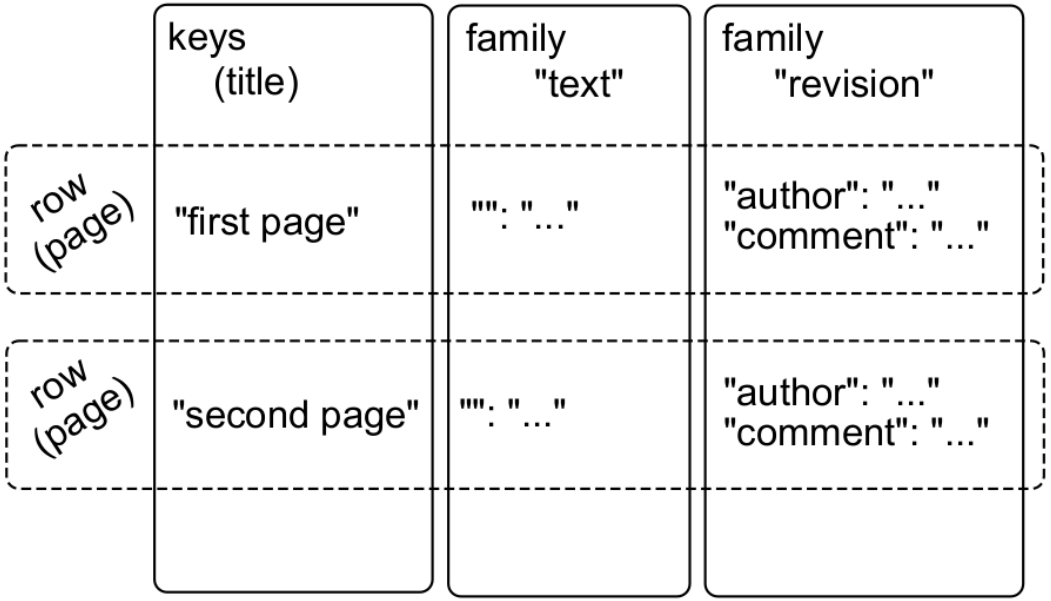
```
get 'wiki', 'Home'
get 'wiki', 'Home', 'text:'
```

HBase: Wiki

- Anforderungen:
  - Eindeutiger Titel
  - Beliebig viele Änderungen
  - Änderung besteht aus Datum, Text, Autor, Kommentar



- Umsetzung in HBase:



HBase: Konfiguration und JRuby

- Speichere alle Versionen (Default: 1)

```
disable 'wiki'
alter 'wiki', { NAME => 'revision', VERSIONS =>
         org.apache.hadoop.hbase.HConstants::ALL_VERSIONS }
```

- Zusätzlich: Kompression und Bloom Filter für das Attribut „text“

```
alter 'wiki', {NAME=>'text', VERSIONS =>
         org.apache.hadoop.hbase.HConstants::ALL_VERSIONS, COMPRESSION=>'GZ',
BLOOMFILTER=>'ROW'}
enable 'wiki'
```

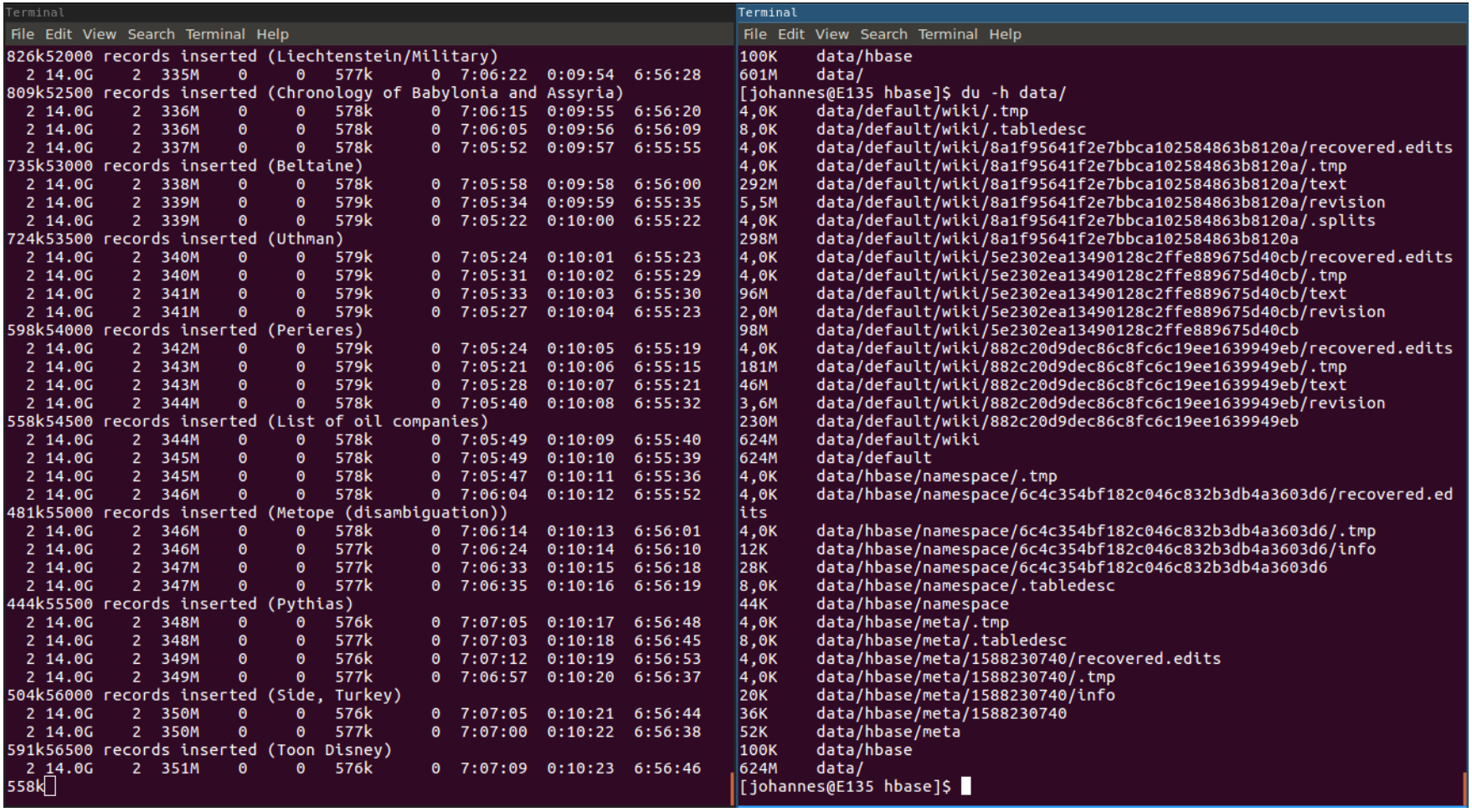
- HBase Shell versteht JRuby:

```
import 'org.apache.hadoop.hbase.client.HTable'
import 'org.apache.hadoop.hbase.client.Put'
def jbytes( *args )
  args.map { |arg| arg.to_s.to_java_bytes }
end
table = HTable.new( @hbase.configuration, "wiki" )
p = Put.new( *jbytes( "Home" ) )
p.add( *jbytes( "text", "", "Hello world" ) )
p.add( *jbytes( "revision", "author", "jimbo" ) )
p.add( *jbytes( "revision", "comment", "my first edit" ))
table.put( p )
```

HBase: Import aus Wikipedia

- Code: https://pragprog.com/titles/rwdata/source_code

```
curl https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-
articles.xml.bz2 | bzcat | bin/hbase shell ../code/hbase/import_from_wikipedia.rb
```



HBase: Region Split

- Teilung einer Region in zwei neue Regions, wenn „zu groß“
- Default: wachsende Größe für Split (256MB, 512MB, 1152MB, …)
- Alte Regions werden durch Merge entfernt
- Konsole (siehe vorherige Folie): `du -h hbase/data`
- HBase Shell:

```
hbase(main):015:0> scan 'hbase:meta', { COLUMNS => [ 'info:server', 'info:regioninfo' ] }
ROW                          COLUMN+CELL
 hbase:namespace,,15281832 column=info:regioninfo, timestamp=1528183235660, value={ENCODED => 6c4c354bf
 34427.6c4c354bf182c046c83 182c046c832b3db4a3603d6, NAME => 'hbase:namespace,,1528183234427.6c4c354bf18
 2b3db4a3603d6.            2c046c832b3db4a3603d6.', STARTKEY => '', ENDKEY => ''}
 hbase:namespace,,15281832 column=info:server, timestamp=1528183235660, value=E135:46449
 34427.6c4c354bf182c046c83
 2b3db4a3603d6.
 wiki,,1528189965315.5e230 column=info:regioninfo, timestamp=1528189983093, value={ENCODED => 5e2302ea1
 2ea13490128c2ffe889675d40 3490128c2ffe889675d40cb, NAME => 'wiki,,1528189965315.5e2302ea13490128c2ffe8
 cb.                       89675d40cb.', STARTKEY => '', ENDKEY => 'First-order m'}
 wiki,,1528189965315.5e230 column=info:server, timestamp=1528189983093, value=E135:46449
 2ea13490128c2ffe889675d40
 cb.
 wiki,First-order m,152818 column=info:regioninfo, timestamp=1528189983077, value={ENCODED => 882c20d9d
 9965315.882c20d9dec86c8fc ec86c8fc6c19ee1639949eb, NAME => 'wiki,First-order m,1528189965315.882c20d9d
 6c19ee1639949eb.          ec86c8fc6c19ee1639949eb.', STARTKEY => 'First-order m', ENDKEY => ''}
 wiki,First-order m,152818 column=info:server, timestamp=1528189983077, value=E135:46449
 9965315.882c20d9dec86c8fc
 6c19ee1639949eb.
3 row(s) in 0.2200 seconds
```

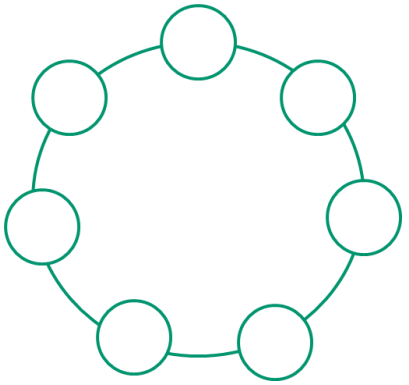
Inhaltsverzeichnis: Wide Column Stores

Apache Cassandra

Quelle: http://de.slideshare.net/DataStax/an-overview-of-apache-cassandra

Apache Cassandra

- Lakshman and Malik: “A Decentralized Structured Storage System” [LM10]
- Flexible Tabellen; Verschachtelte Zeilen
- Schnelles Schreiben
- Hochverfügbar und nahtlos skalierbar
- Peer-to-Peer mit asynchroner Multi-Master Replikation
- Fehlertoleranz (ohne „Single Point of Failure“)



- Gossip Kommunikation, Anti-Entropy Synchronisation, Hinted Handoff
- **SQL-ähnliche Anfragesprache (CQL)**
- Nutzerdefinierte Datentypen; Mengenwertige Werte
- Eventually consistent, aber **konfigurierbar über R/W-Quoren und Paxos**
- Anwendung: Viele ungleich verteilte Schreib- und wenige vorhersagbare Leseoperationen, z.B. Speicherung von Nutzeraktivitäten auf sozialen Netzwerken, Empfehlungen/Reviews oder statistische Analysen

CQL – Cassandra Query Language

- Ausführung über CQL Shell: `cqlsh`
- Leicht lesbar, doch tieferes Verständnis der zugrundeliegenden Architektur für fehlerlosen Einsatz erforderlich
- Erstellen eines Namespace (Datenbank):

```
CREATE KEYSPACE animalkeyspace WITH REPLICATION = {
  'class' : 'SimpleStrategy' , 'replication_factor' : 3 };
USE animalkeyspace;
```

- Erstellen einer Column Family (Tabelle):

```
CREATE TABLE Monkey (
  identifier uuid,
  species text,
  nickname text,
  population int,
  PRIMARY KEY ((identifier), species));
```

CQL: Datentypen

Datentyp	Beschreibung
ascii, text/varchar	Zeichenkette in ASCII bzw. UTF-8 Kodierung
boolean	true oder false
tinyint, smallint, int, bigint, varint	Ganze Zahl (8-/16-/32-/64-bit/beliebig)
float, double, decimal	Fließkommazahl (32-/64-bit/beliebig)
date, time, timestamp	Datum, Zeit, Datum und Zeit
duration	Zeitdauer, z.B. 12h30m
counter	Ganze Zahl (64-bit), nur In-/Dekrement
inet	IP-Adresse (IPv4 oder IPv6)
uuid, timeuuid	UUID Version 4/1
blob	Binärer Datentyp

CQL: Primärschlüssel

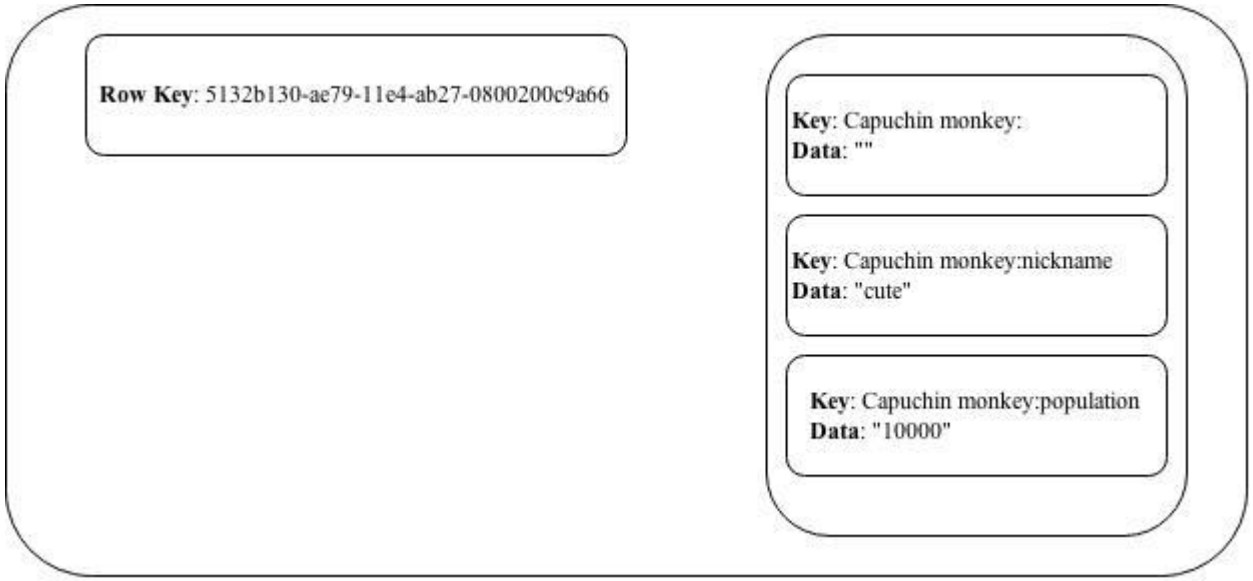
- Eindeutig & notwendig
- Mehrere Spalten möglich
- Zwei Bestandteile
  1. **Partition Key**: mind. eine Spalte; definiert die **Partition** (und somit das Replica-Set) eines Eintrags
  2. **Clustering Columns**: Bestimmung der Reihenfolge innerhalb einer Partition
- Wahl des Primärschlüssel beeinflusst Leistung

```
CREATE TABLE t (
    a int,
    b int,
    c int,
    d int,
    PRIMARY KEY ((a, b), c, d)
);

SELECT * FROM t;
   a | b | c | d
  ---+---+---+---
   0 | 0 | 0 | 2    // row 1
   0 | 0 | 2 | 1    // row 2
   0 | 1 | 5 | 2    // row 3
   0 | 1 | 5 | 4    // row 4
   1 | 1 | 4 | 5    // row 5
```

CQL: INSERT (1)

```
INSERT INTO Monkey (identifier, species, nickname, population)
VALUES ( 5132b130-...-200c9a66, 'Capuchin monkey', 'cute', 100000);
```

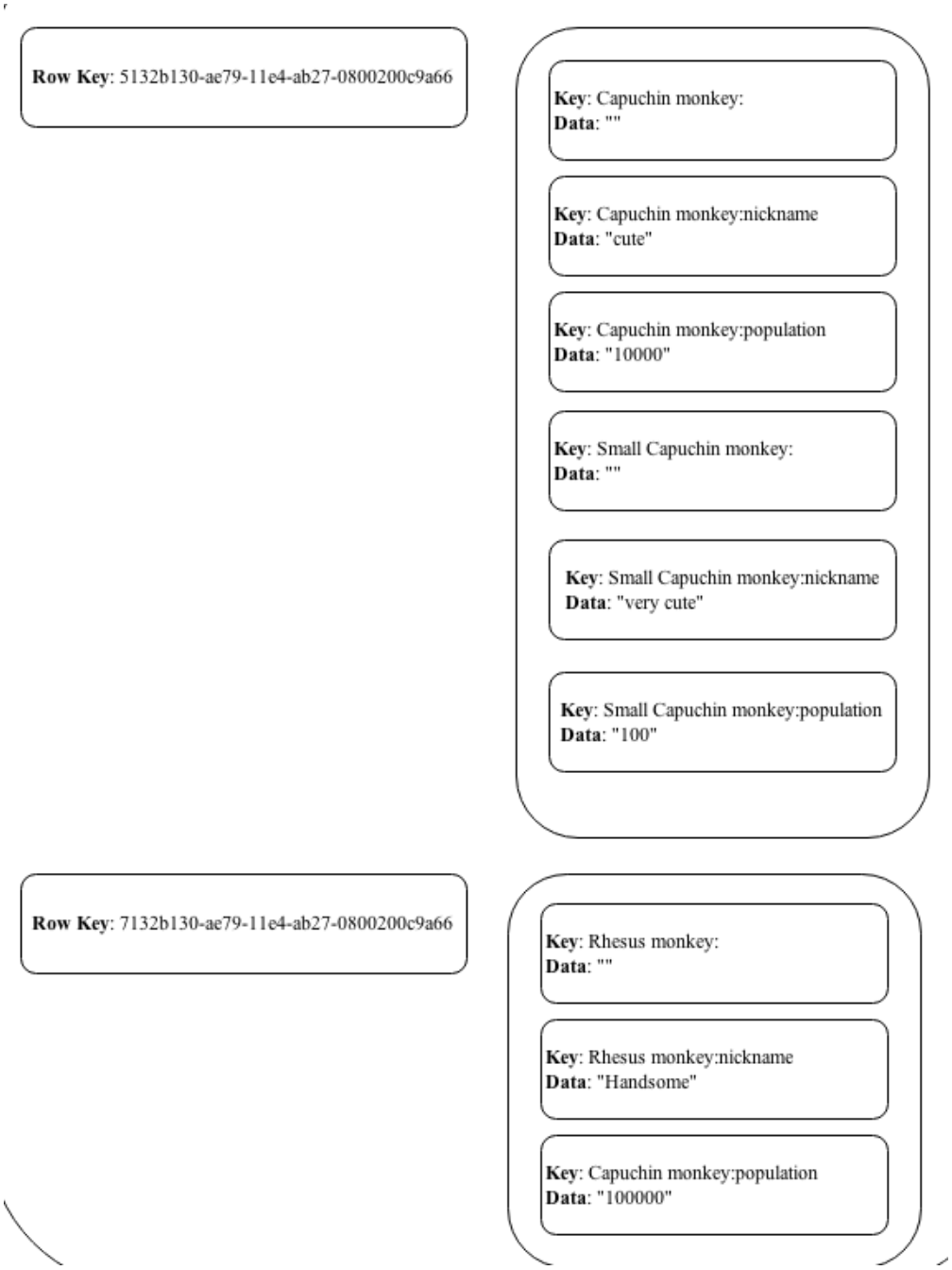
- Struktur in SSTable:



Quelle: http://abiasforaction.net/a-practical-introduction-to-cassandra-query-language/

CQL: INSERT (2)

```
INSERT INTO monkey
  (identifier, species,
   nickname, population)
 VALUES (
  5132b130-...-200c9a66,
  'Small Capuchin monkey',
  'very cute',100);

INSERT INTO monkey
  (identifier, species,
   nickname, population)
 VALUES (
  7132b130-...-200c9a66,
  'Rhesus Monkey',
  'Handsome', 100000);
```



CQL: SELECT (1)

```
SELECT identifier, nickname FROM Monkey
  WHERE species = 'Capuchin monkey’;

SELECT nickname AS monkey_kname,
       species AS monkey_species FROM Monkey;

SELECT COUNT(*) AS monkey_count FROM Monkey;
```

- **Nur 1 Tabelle pro Anfrage**: Keine Joins oder Subqueries
- Cassandra strebt eine Bearbeitungszeit an, die linear nur mit der *Größe der Ergebnismenge* (und nicht mit der Tabellengröße) wächst
  - WHERE ist nur auf Spalten des Primärschlüssels oder indizierten Spalten möglich
  - Anfragen erzwingen: ALLOW FILTERING

```
CREATE INDEX ON
  Monkey(nickname);
```

```
SELECT * FROM Monkey
  WHERE population < 1000 ALLOW FILTERING;
```

CQL: SELECT (2)

- Beschränkung der Operatoren:
  - Für Partition Key: nur Gleichheit (=, IN)
  - IN nur auf letzten Teil des Primärschlüssels oder letzten Teil des Partition Key
- Anfragen nur über benachbarte Bereiche einer Partition, z.B:
  - Möglich:

```
CREATE TABLE posts (
    userid text,
    blog_title text,
    posted_at timestamp,
    content text,
  PRIMARY KEY (userid,
    blog_title, posted_at));
```

```
SELECT * FROM posts
 WHERE userid = 'john doe'
   AND blog_title='John''s Blog'
   AND posted_at >= '2012-01-01' AND posted_at < '2012-01-31‘;
```

  - Nicht möglich:

```
SELECT * FROM posts
 WHERE userid = 'john doe'
   AND posted_at >= '2012-01-01' AND posted_at < '2012-01-31‘;
```

CQL: SELECT (3)

- Tuple im WHERE möglich
  - z.B. alle Einträge, welche dem Eintrag mit blog_title = 'John"s Blog' und posted_at = '2012-01-01' folgen:

```
SELECT * FROM posts
 WHERE userid = 'john doe'
   AND (blog_title, posted_at) > ('John''s Blog', '2012-01-01');
```

  - Gibt evtl. auch Einträge, die vor dem 1.1.2012 erschienen sind ...

- GROUP BY: Nur über Spalten des Primärschlüssels in deren Reihenfolge
  - Möglich:

```
SELECT userid, blog_title, max(posted_at) FROM posts
   GROUP BY userid, blog_title;
```

  - Nicht möglich

```
SELECT userid, blog_title, max(posted_at) FROM posts
   GROUP BY blog_title;
```

CQL: SELECT (3)

- ORDER BY: Nur in der gegebenen Reihenfolge

```
SELECT userid, blog_title FROM posts WHERE userid = 'john doe‘
ORDER BY (blog_title ASC, posted_at ASC);
```

- Oder in umgekehrter Reihenfolge

```
SELECT userid, blog_title FROM posts WHERE userid = 'john doe‘
ORDER BY (blog_title DESC, posted_at DESC);
```

- Abweichungen müssen zu Beginn definiert werden:

```
CREATE TABLE posts (userid text, ..., PRIMARY KEY (userid,
    blog_title, posted_at))
WITH CLUSTERING ORDER BY (blog_title DESC, posted_at ASC);
```

```
SELECT userid, blog_title FROM posts WHERE userid = 'john doe‘
ORDER BY (blog_title DESC, posted_at ASC);
```

```
SELECT userid, blog_title FROM posts WHERE userid = 'john doe‘
ORDER BY (blog_title ASC, posted_at DESC);
```

CQL: Mengenwertige Datentypen (1)

- Für „kleine“ Mengen von Daten, z.B. E-Mail-Adressen eines Nutzers
- **Set:**

```
CREATE TABLE images (
    name text PRIMARY KEY,
    owner text,
    tags set<text>);

INSERT INTO images (name, owner, tags)
   VALUES ('cat.jpg', 'jsmith', { 'pet', 'cute' });

UPDATE images SET tags = tags + { 'gray', 'cuddly' }
   WHERE name = 'cat.jpg';

UPDATE images SET tags = tags - { 'cat' }
   WHERE name = 'cat.jpg';
```

CQL: Mengenwertige Datentypen (2)

- **List**:

```
CREATE TABLE plays (
    id text PRIMARY KEY,
    game text,
    scores list<int>);
INSERT INTO plays (id, game, scores)
   VALUES ('123-afde', 'quake', [17, 4, 2]);

UPDATE plays SET scores = scores + [ 14, 21 ]
   WHERE id = '123-afde';
UPDATE plays SET scores = [ 3 ] + scores WHERE id = '123-afde';
UPDATE plays SET scores[1] = 7 WHERE id = '123-afde';

DELETE scores[1] FROM plays WHERE id = '123-afde';
UPDATE plays SET scores = scores - [ 12, 21 ]
   WHERE id = '123-afde';
```

CQL: Mengenwertige Datentypen (3)

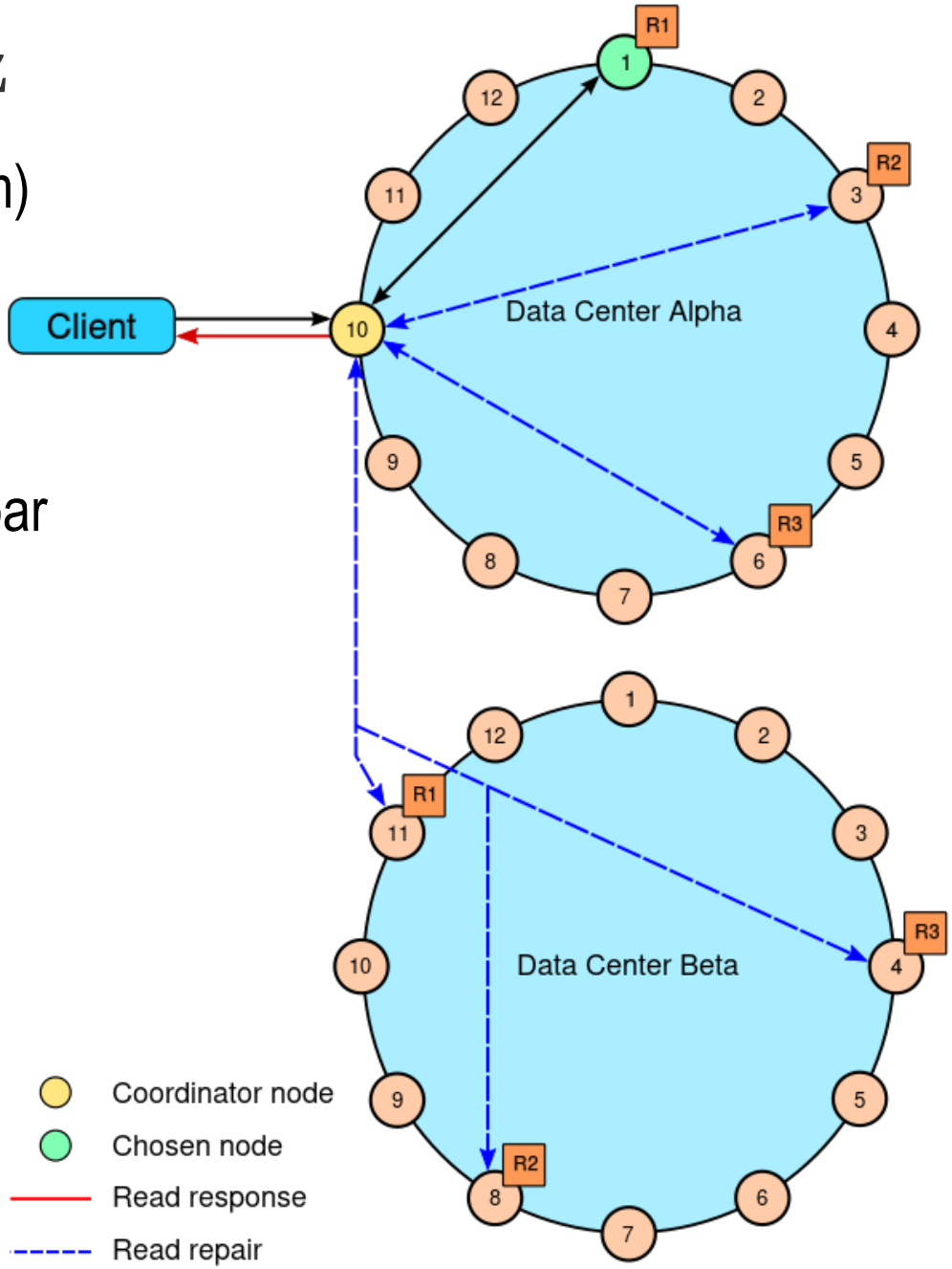
- **Map**:

```
CREATE TABLE users (
    id text PRIMARY KEY,
    favs map<text, text>);
INSERT INTO users (id, name, favs)
   VALUES ('jsmith', { 'fruit' : 'Apple', 'band' : 'Beatles' });

UPDATE users SET favs['author'] = 'Ed Poe' WHERE id = 'jsmith';
UPDATE users SET favs = favs +
    { 'movie' : 'Cassablanca', 'band' : 'ZZ Top' }
  WHERE id = 'jsmith';

DELETE favs['author'] FROM users WHERE id = 'jsmith';
UPDATE users SET favs = favs - { 'movie', 'band'}
  WHERE id = 'jsmith';
```

Inhaltsverzeichnis: Wide Column Stores

Cassandra: Konsistenz

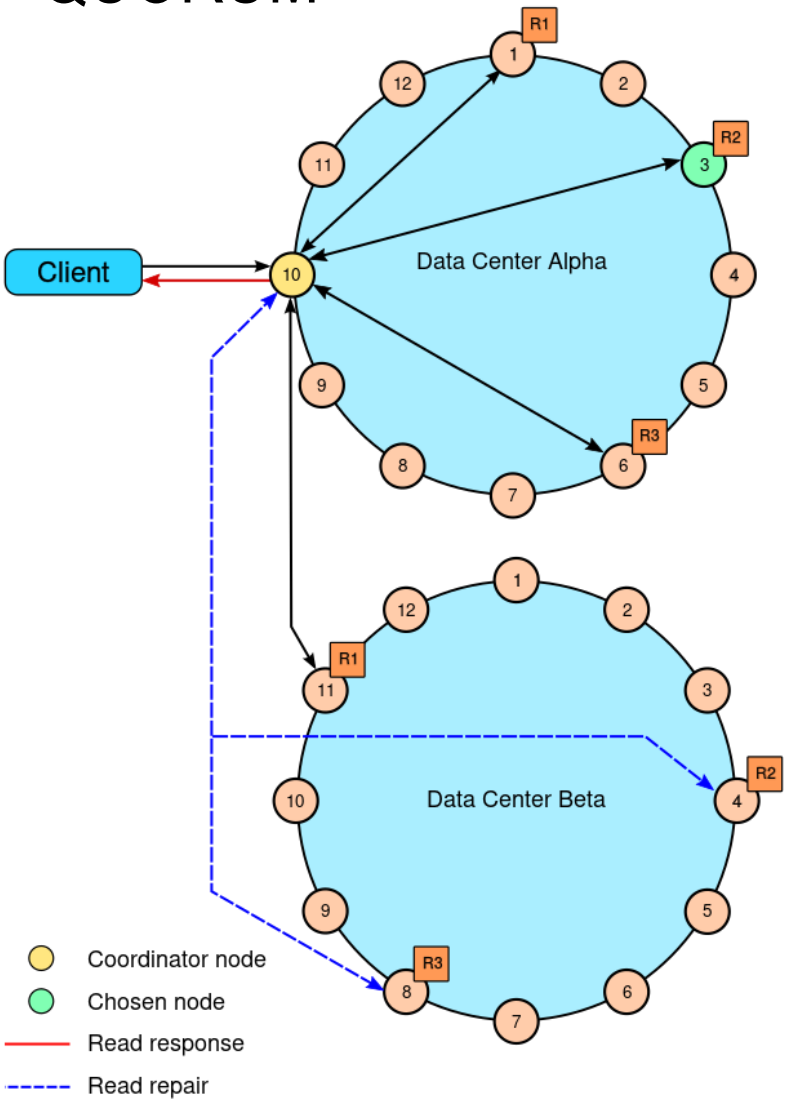
- Cassandra ist AP (CAP Theorem)
- Read-Repair-Prozesse: Asynchrone Wiederherstellung der Konsistenz
- Stärkere Konsistenz konfigurierbar über Read/Write-Quoren
- CQL: `CONSISTENCY <level>;`
- Default: ONE (Abbildung)



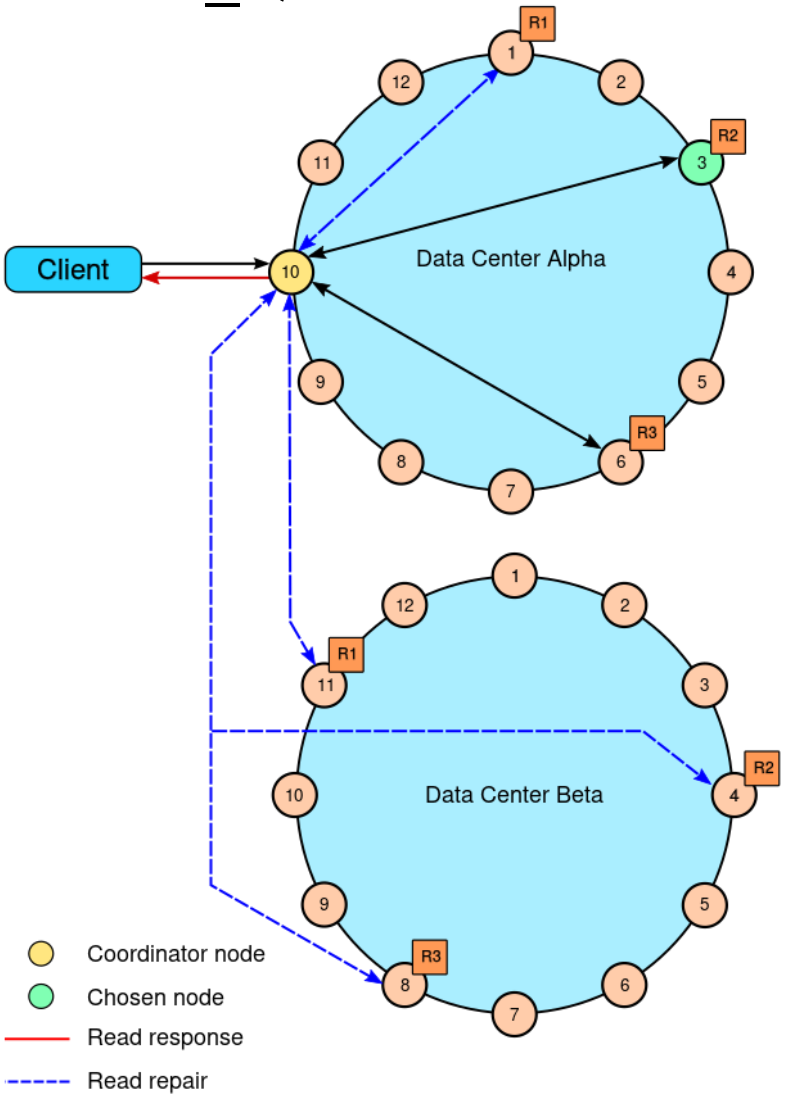
Quelle: https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlClientRequestsReadExp.html

Cassandra: Read/Write-Quoren (1)

- QUORUM



- LOCAL_QUORUM



Cassandra: Read/Write-Quoren (2)

CONSISTENCY	Lesen	Schreiben
QUORUM	Mehrheit der Replikate muss einen Wert liefern	Mehrheit der Replikate muss den Wert erfolgreich geschrieben haben
LOCAL_QUORUM	QUORUM beschränkt auf Rechenzentrum des Koordinators	QUORUM beschränkt auf Rechenzentrum des Koordinators
EACH_QUORUM	Nicht unterstützt	QUORUM für jedes Rechenzentrum
ONE	Ein Replikat muss Wert liefern	Ein Replikat muss den Wert erfolgreich geschrieben haben
LOCAL_ONE	…	…
TWO	…	…
THREE	…	…
ALL	…	…
ANY	Nicht unterstützt	Wie ONE, aber „Hinted Handoff„ erlaubt
SERIAL	Auch Werte ohne „Commit“ (durch ANY) werden gelesen	Nicht unterstützt
LOCAL_SERIAL	SERIAL beschränkt auf Rechenzentrum des Koordinators	Nicht unterstützt

Cassandra: Lightweight Transactions (1)

- Tabelle `payments` mit Primärschlüssel
  `(payment_time, customer_id, payment_no)`
- Nutzer A:

```
CONSISTENCY ALL;
INSERT INTO payments (payment_time, customer_id, payment_no,
amount) VALUES ('2016-11-02 12:23', 123, 1, 12.00);
```

- Nutzer B:

```
CONSISTENCY ALL;
INSERT INTO payments (payment_time, customer_id, payment_no,
amount) VALUES ('2016-11-02 12:23', 123, 1, 10.00);
```

- Default: INSERT in CQL überprüft nicht die Existenz eines Schlüssel bevor Wert überschrieben wird (neue Version angelegt)

Cassandra: Lightweight Transactions (2)

- Transaktionen in Cassandra möglich
- IF NOT EXISTS (INSERT)

```
INSERT INTO payments (payment_time, customer_id, payment_no,
amount) VALUES ('2016-11-02 12:23', 123, 1, 10.00)
IF NOT EXISTS;
```

- IF <cond> (UPDATE)

```
UPDATE payments SET amount = 10.00
WHERE payment_date = 2016-11-02 12:23:34Z
AND customer_id = 123 IF amount = 12.00;
```

- Keine Blockierung
- Aufwendiger Konsensus zwischen Replikaten über *Paxos-Protokoll*
- Transaktionen sind auf eine Partition beschränkt

Inhaltsverzeichnis: Wide Column Stores

Zusammenfassung: Wide Column Stores

- Hohe Skalierbarkeit und Flexibilität
- Geeignet für Anwendungen, welche eine hohe Schreibleistung oder die Verwendung mehrerer Server erfordert
- Trotz der stark überlappenden Terminologie ist die Ähnlichkeit zu relationalen DBS nur oberflächlich
- Richtige Verwendung erfordert, im Einklang mit der Datenstruktur und der Implementierung zu arbeiten

HBase	Cassandra
- Master-Slave Replikation - Starke Konsistenzgarantien - Teil von Hadoop	- Multi-Master Replikation - Hohe Verfügbarkeit falls eventual consistent - Optional: stärkere Konsistenz und Lightweight Transactions - Cassandra Query Language

Referenzen

- [HBase] https://hbase.apache.org/book.html
- [White15] White, Tom. Hadoop:The Definitive Guide. O‘Reilly, 4th ed. 2015
- [BigTable] Fay Chang, Jeffrey Dean, Sanjay Ghemawat et al. Bigtable: A Distributed Storage System for Structured Data. OSDI’06

- [Cassandra] http://cassandra.apache.org/doc/latest/
- [LM10] Lakshman, Avinash, and Prashant Malik. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review 44.2 (2010): 35-40.
- [CH16] Jeff Carpenter and Eben Hewitt. Cassandra: The Definitive Guide. O‘Reilly. 2nd ed. 2011