adesso business. people. technology.

NoSQL Deep Dive mit Cassandra

Kai Spichale

NoSQL

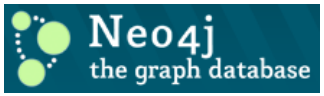
NoSQL

Wide Column Stores / Column Families

Document Stores

Graph Databases

Key Value / Tupe Stores



Apache Cassandra - eine Definition

“Apache Cassandra is an open source, distributed, decentralized, elastically scalable, highly available, fault-tolerant, tuneably consistent, column-oriented database that bases its distribution design on Amazon’s Dynamo and its data model on Google’s Bigtable. Created at Facebook, it is now used at some of the most popular sites on the Web.”

Hewitt, Eben: Cassandra – The Definite Guide, S. 14

Agenda

- Key Features
- Projektgeschichte
- Verteilung mit DHT
- Konsistenz
- Replikatverteilung
- Datenmodell
- Client API
- Fazit

Keyfeatures

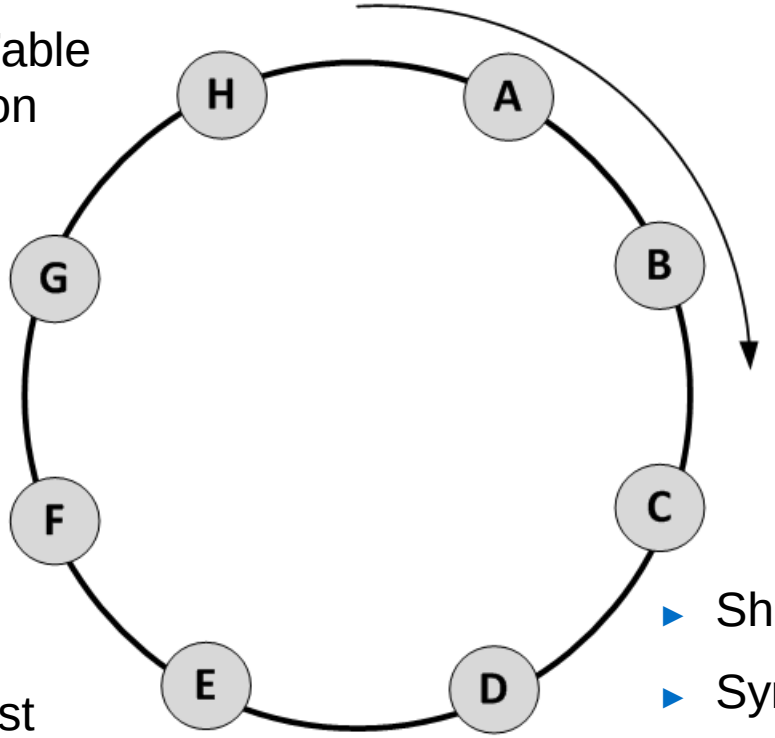
- Verteilte, hochizontal skalierbare Datenbank
- Symmetrisches Design
- Hochverfügbar
- Riesige Datenmenge (Petabytes)
- Flexible Partitionierung, Replikatverteilung
- Eventually consistent: ACID ist nicht immer notwendig
- Flexible Trade-offs zwischen Konsistenz und Performance
- Automated Provisioning (Seek Nodes)
- Erweiterbare Multi-Datacenter Unterstützung
- Schemaloses, strukturiertes Datenmodell

Projektgeschichte

- Ursprunglich von facebook entwickelt
- Projektbeginn 2007, seit 2008 Open Source
- Verwendung für Inbox Search:
  - Benutzer können ihre Nachrichten nach Absendernamen oder anderen Schlüsselwörtern durchsuchen
  - In-house System zum Indexieren (invertierte Indizes) und Speichern der Nachrichten
- Anforderung:
  - kostengünstig (Commodity Server)
  - inkrementell skalierbar

Verteilung mit DHT

- Ein **D**istributed **H**ash **T**able (DHT) ist ein Klasse von dezentralen verteilten Systemen
- O(1) Knoten-Lookup
- P2P- Netzwerk



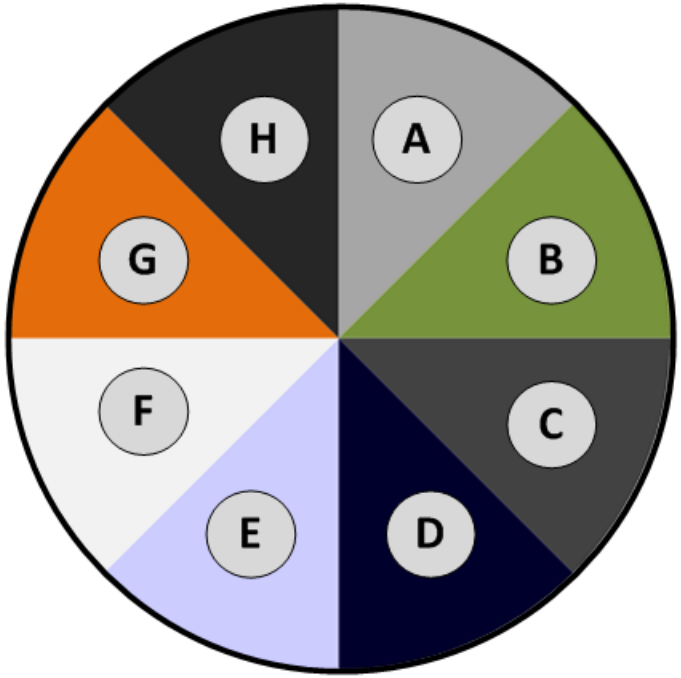
- Daten werden möglichst gleichmäßig auf die Knoten verteilt

- Shared-nothing Architecture
- Symmetrisches Design:
  - Kein Single Point of Failure
  - Kein zentraler Controller
  - Keine Master/Slaves
- Clients können sich mit beliebigen Knoten verbinden

Verteilung mit DHT

Partitioner(RowKey) = Token

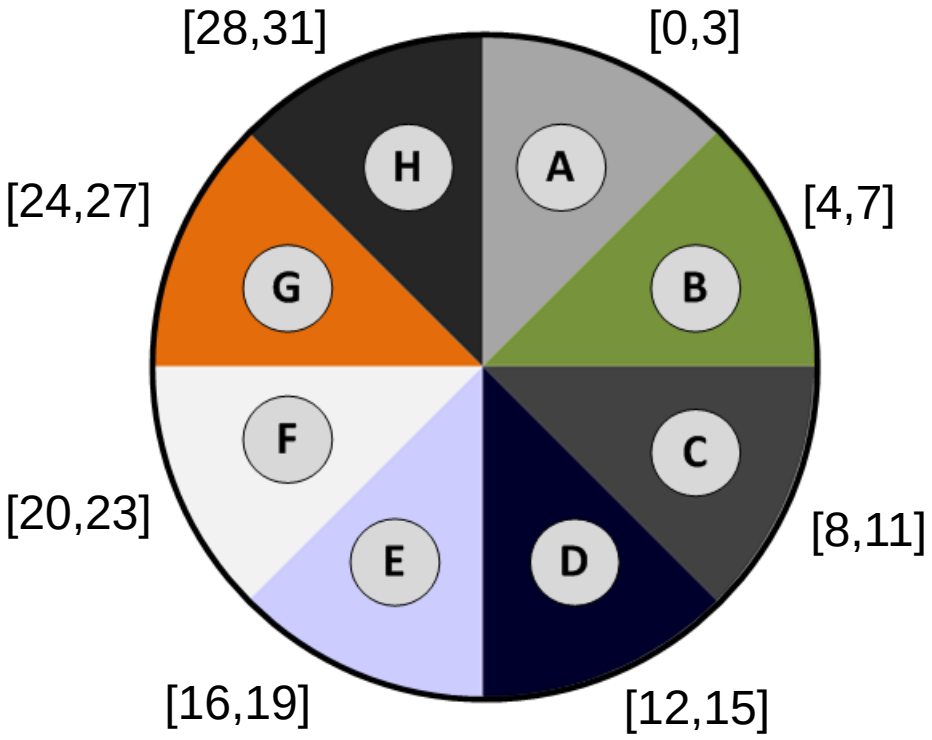
Hosts	Initial Token
A	0
B	4
C	8
D	12
E	16
F	20
G	24
H	28



Verteilung mit DHT

Partitioner(RowKey) = Token

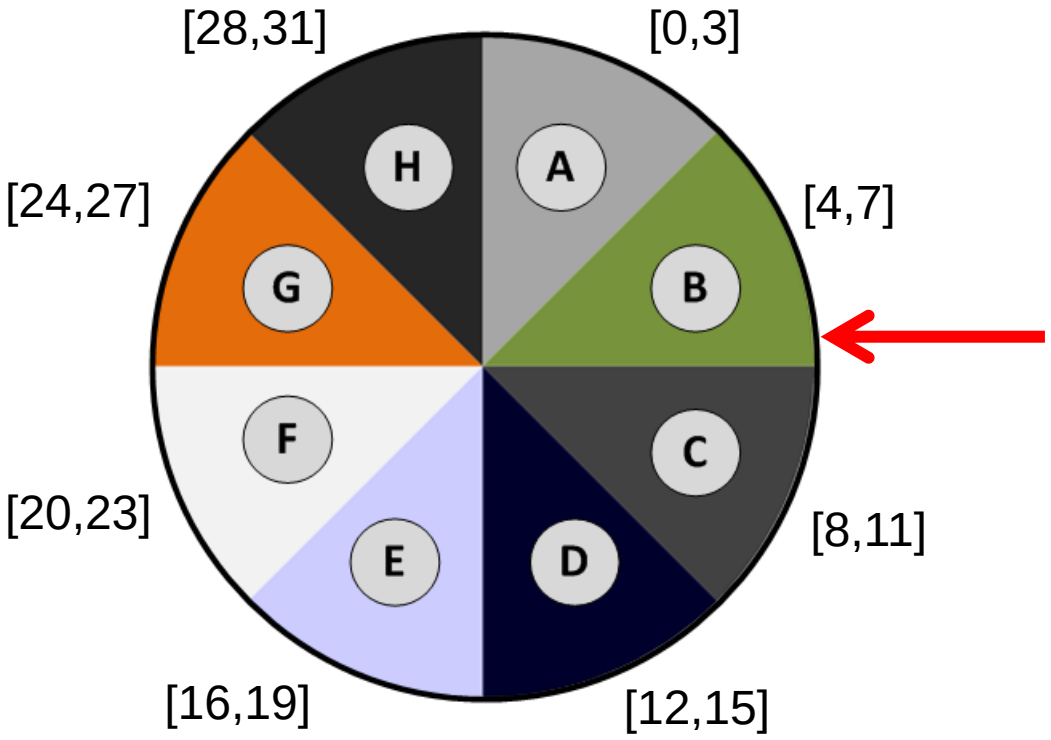
Hosts	Initial Token
A	0
B	4
C	8
D	12
E	16
F	20
G	24
H	28



Verteilung mit DHT

Partitioner(RowKey) = Token

Hosts	Initial Token
A	0
B	4
C	8
D	12
E	16
F	20
G	24
H	28



Beispiel:
Partitioner(„Cassandra“) = 7

Verteilung mit DHT

Hosts	Initial Token
A	0
B	4
C	8
D	12
E	16
F	20
G	24
H	28

Hosts	Initial Token
A	0
B	21267647932558653966460912964485513215
C	42535295865117307932921825928971026430
D	63802943797675961899382738893456539645
E	85070591730234615865843651857942052860
F	106338239662793269832304564822427566075
G	127605887595351923798765477786913079290
H	148873535527910577765226390751398592505

Tokens sind Integer von 0 bis 2^127

i * (2^127 / N) für i = 0 .. N-1

z.B. Random Partitioner mit MD5

Verteilung mit DHT

- Verteilung der Daten im Cluster durch Partitioner

- **Random Partitioner** (Consistent Hashing)
  - Basiert auf MD5
  - Implizites Load Balancing

- **Order Preserving Partitioner**
  - Geeignet für Range Slices
  - Evtl. ungleiche Verteilung der Daten im Cluster

- Weitere Implementierungen von org.apache.cassandra.dht.IPartitioner

CAP-Theorem

- CAP:
  - Strong **C**onsistency
  - High **A**vailability
  - **P**artition Tolerance

- CAP-Theorem nach Eric Brewer besagt, dass ein verteiltes System nicht gleichzeitig alle drei Anforderungen erfüllen kann, sondern höchstens zwei.

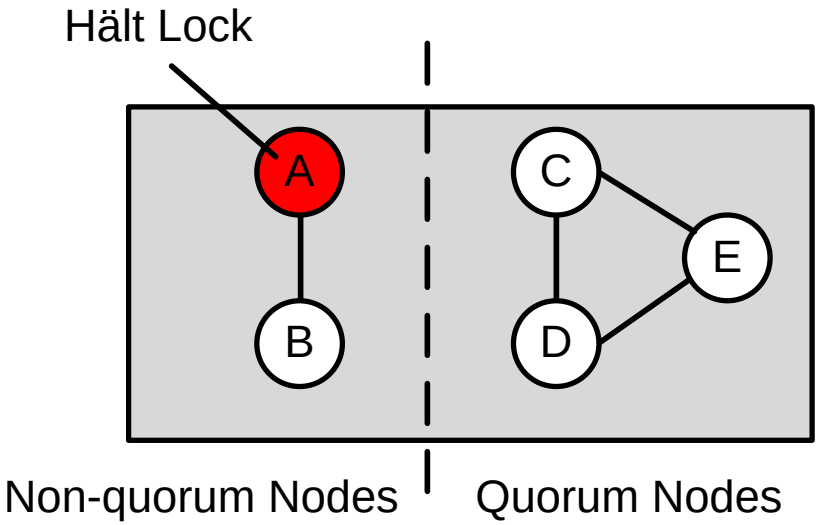
- Cassandra ist ein AP-System

CAP-Theorem

- CA-System:
  - Jeder lebende Knoten, der Requests empfängt, sendet Responses
  - Netzwerkpartitionen werden nicht berücksichtigt (Implementierungsdetail)
- CP-System:
  - Kann bei Netzwerkpartitionen betrieben werden
  - Knoten können absichtlich deaktiviert werden, um Konsistenz sicherzustellen
  - Verfügbarkeit wird gegen Konsistenz eingetauscht
- AP-System:
  - Verfügbar und tolerant gegenüber Partitionen
  - Konsistenz kann nicht garantiert werden

CAP-Theorem

- B und C erhalten Request:
- CA:
  - B kann antworten
  - C wird blockiert
- CP:
  - C kann antworten
  - Lock von A wird aufgehoben
- AP:
  - B und C können antworten
  - Eventuell inkonsistente Daten



Konsistenz

- Vertikale vs. horizontale Skalierung
- ACID vs. BASE (basically available, soft state, eventually consistent)
- Eventually consistency
  - Synch nach Berlin, asynch nach New York
  - Kompromiss zugunsten Verfügbarkeit und Performance
- Cassandra unterstützt verschiedene Konsistenzstufen

Konsistenz

- Verschiedene Konsistenzstufen für Lese- und Schreiboperationen
- Wenn W + R > N, dann (fast) stark konsistent

Level	Write Consistency
ZERO	Asynchron, ohne Rückmeldung
ANY	1 Knoten (inkl. HintedHandoff Empfänger)
ONE	1 Knoten
QUORUM	(N/2)+1 Knoten
LOCAL_QUORUM	Quorum im lokalen DC
EACH_QUORUM	Quorum in allen DCs
ALL	N Knoten

R + W > N

Level	Read Consistency
ONE	1 Knoten
QUORUM	(N/2)+1 Knoten
LOCAL_QUORUM	Quorum im lokalen DC
EACH_QUORUM	Quorum in allen DCs
ALL	N Knoten

R + W > N

Konsistenz

- Hinted Handoff: Bei Knotenausfall wird auf einem anderen Knoten ein Hint hinterlassen, sodass die Schreiboperation nachgeholt werden kann
- Read Repair: Synchronisation aller Replikate (evtl. im Background)
- Anti Entropy: Vergleich der Replikate und Aktualisierung

Konsistenzprobleme beim Löschen

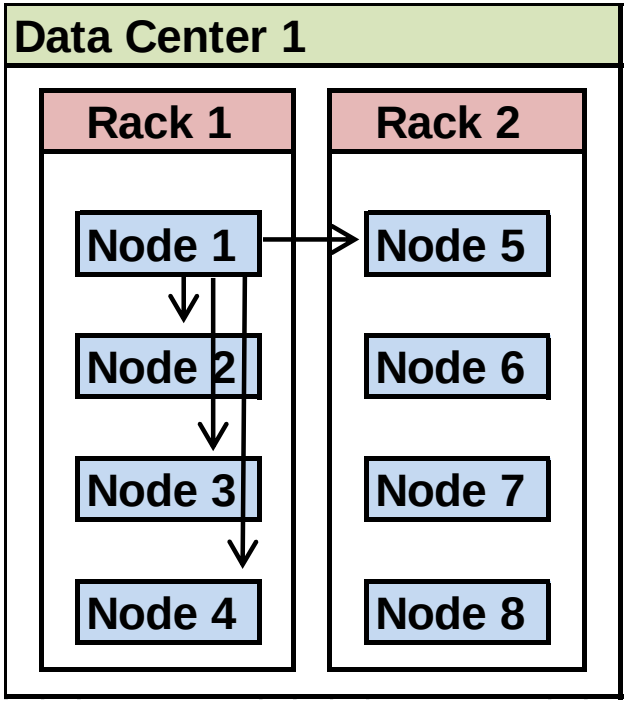
- Eventually consistent: Client liest möglicherweise ein Replikat, das noch nicht alle Updates erhalten hat (bei niedriger Konsistenzstufe)
- Gleiches Problem beim Löschen
- Lösung:
  - Nicht sofort Löschen, sondern mit Tombstones markieren
  - Wenn GCGraceSeconds < Tombstone Alter, dann Löschen
- Konsequenz: Knoten dürfen nicht länger als GCGraceSeconds down sein!

Replikationsverteilung

- Variabler Replikationsfaktor bestimmt die Anzahl der Kopien im Cluster
- Strategien zur Verteilung der Replikate im Cluster:
  - Simple Strategy
  - Old Network Topology Strategy
  - Network Topology Strategy

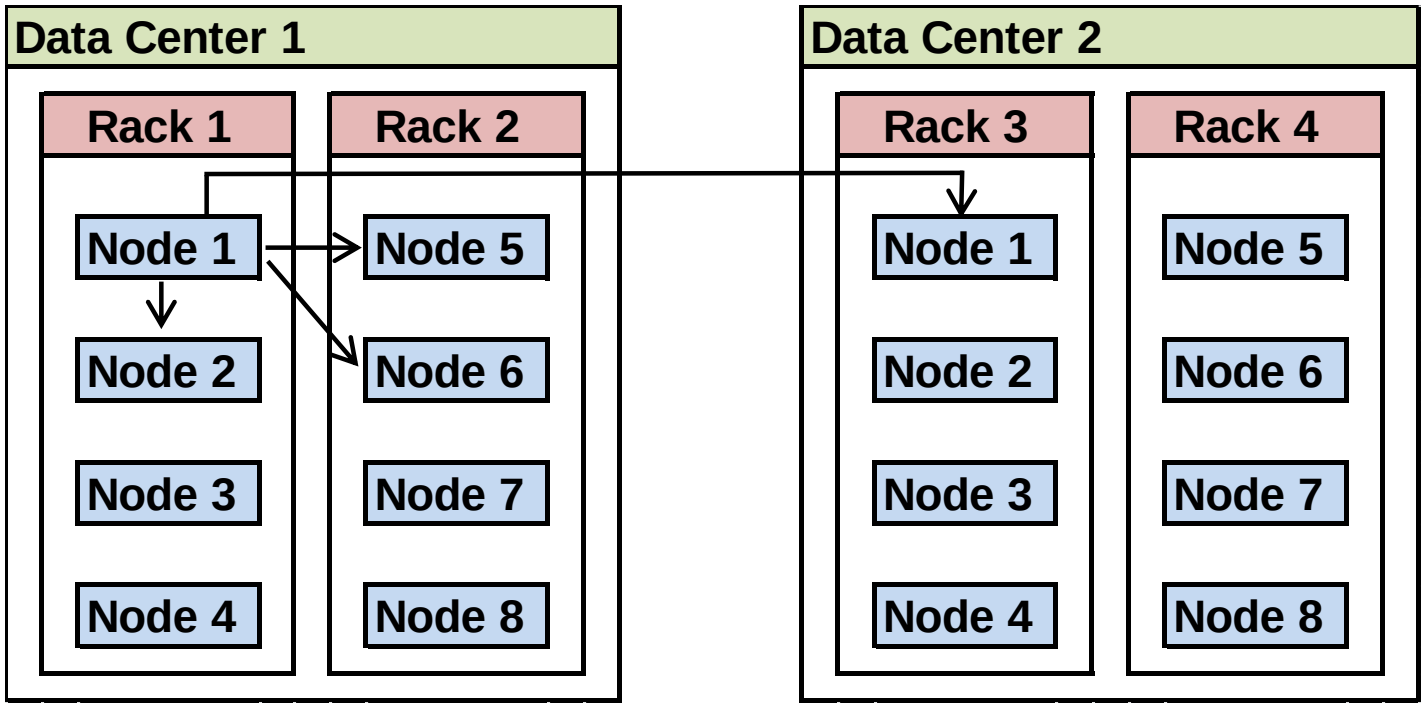
Replikationsverteilung

- **Simple Strategy** (Rack-unaware Strategy): Replikate werden auf nachfolgende Knoten im Ring verteilt
- Datacenters und Racks werden nicht beachtet
- Standardstrategie



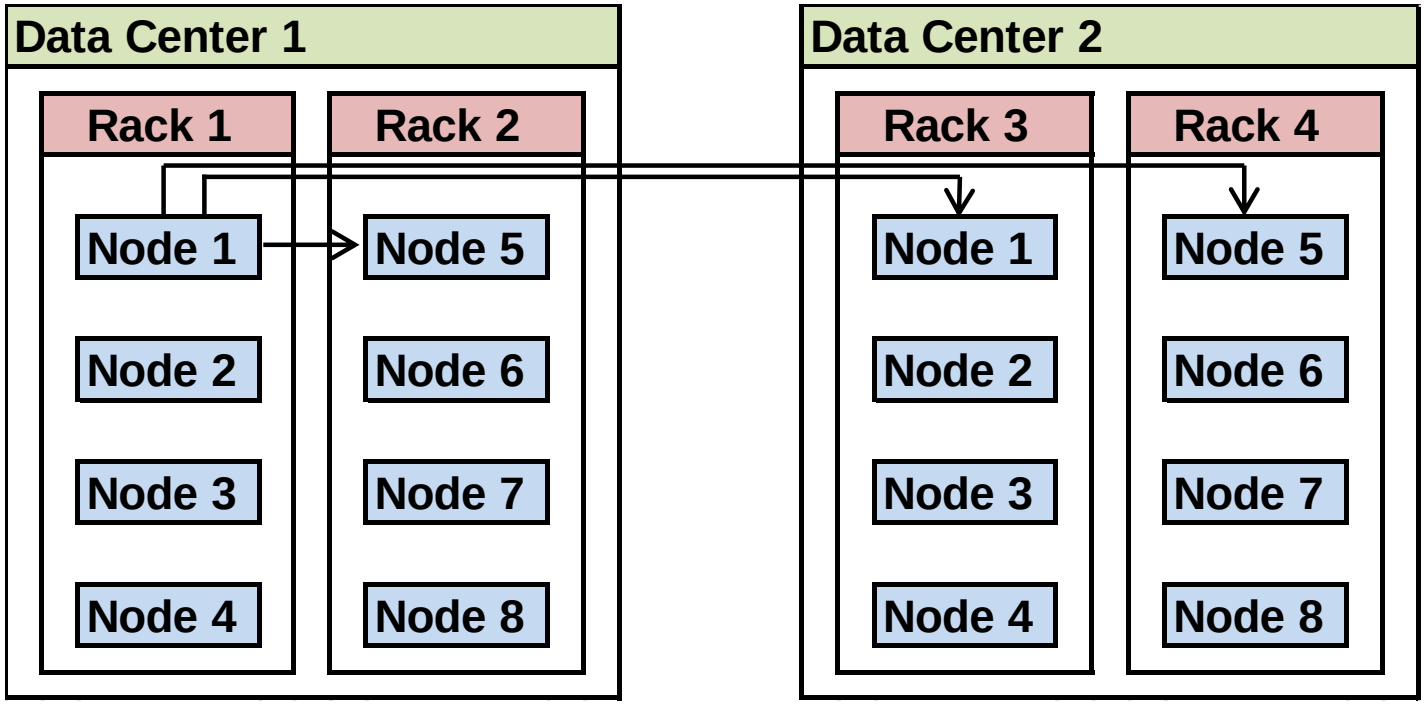
Replikationsverteilung

- **Old Network Topology Strategy** (Rack-aware Strategy): Heuristik zur Verteilung der Replikate auf verschiedenen Racks und Datacenters
- Rack-aware Snitch ist notwendig
- Höhere Verfügbarkeit
- Höhere Latenz



Replikationsverteilung

- **Network Topology Strategy** (Datacenter Shard Strategy): Für jedes DC kann pro Keyspace die Anzahl der Replikate angegeben werden
- Verteilung auf verschiedene Racks innerhalb eines DCs (wenn möglich)
- Rack-ware Snitch ist notwendig



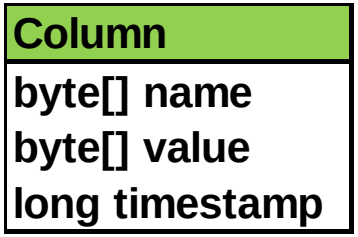
Konfiguration:
DC 1 hat 2 Replikate
DC 2 hat 2 Replikate

Datenmodell

- Strukturiertes Datenmodell ohne Schema
- Mehr als nur Key-Value-Modell
- Besteht aus den Konzepten:
  - Cluster
  - Keyspace
  - Column Family
  - Row
  - SuperColumn
  - Column

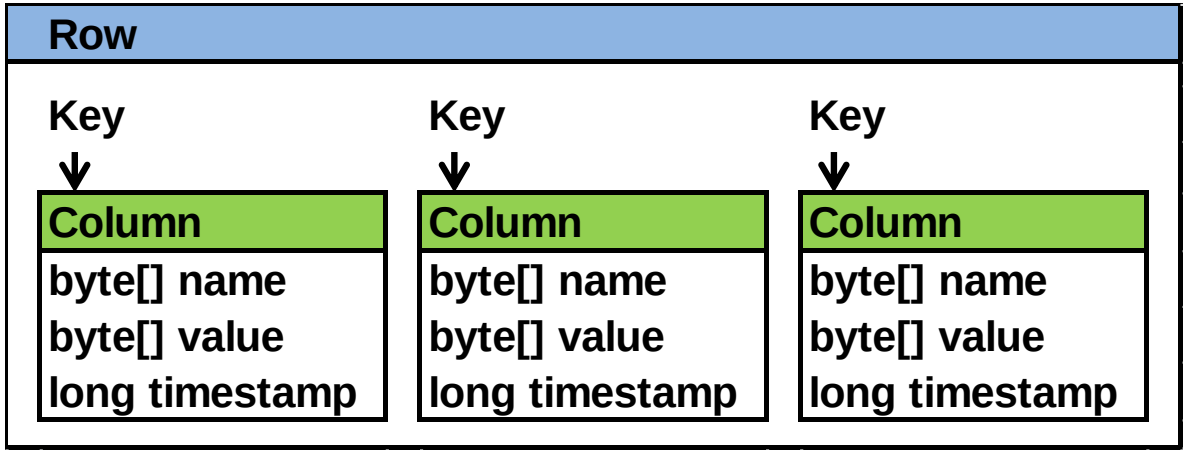
Datenmodell

- Cluster > Keyspace > Column Family > Row > **Column**



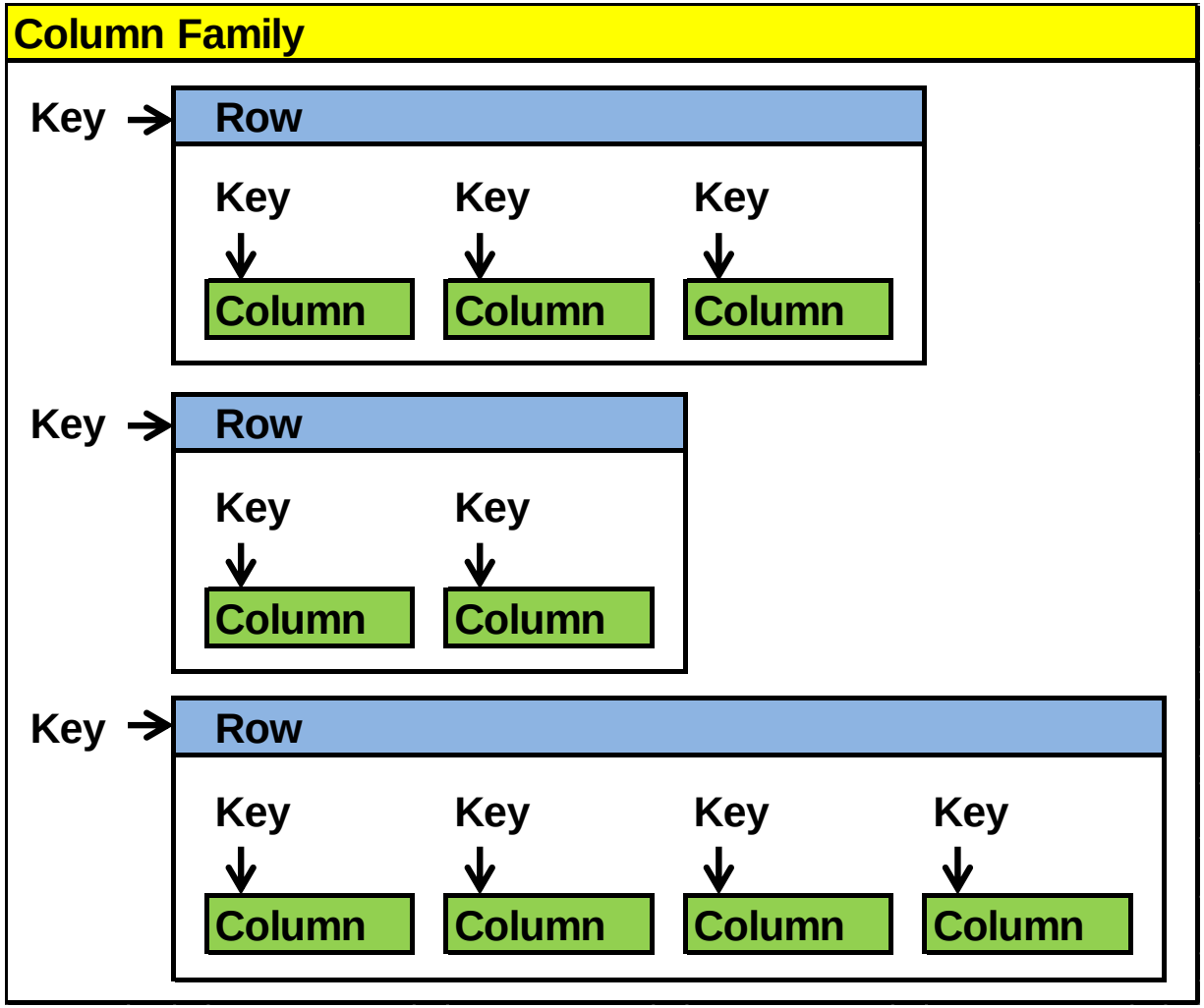
Datenmodell

- Cluster > Keyspace > Column Family > **Row**



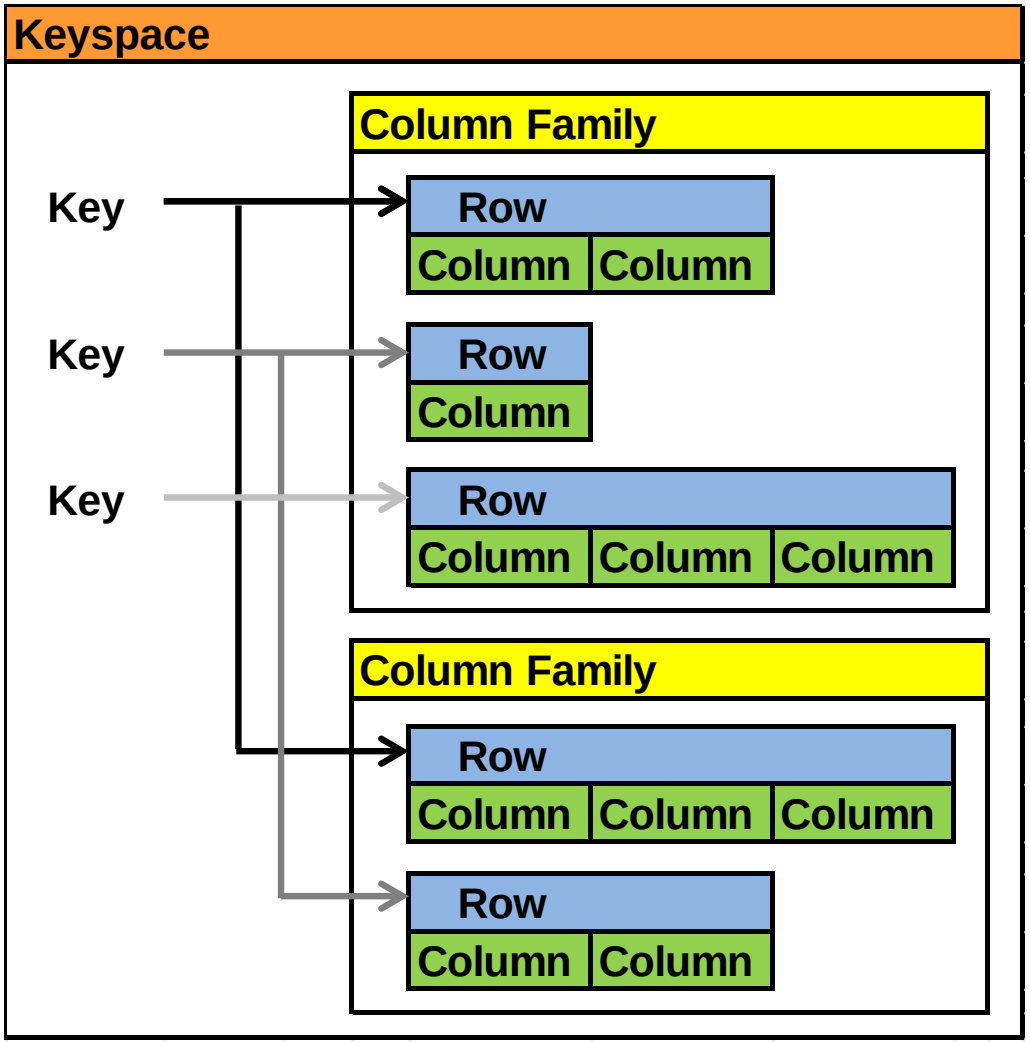
Datenmodell

- Cluster > Keyspace > **Column Family**



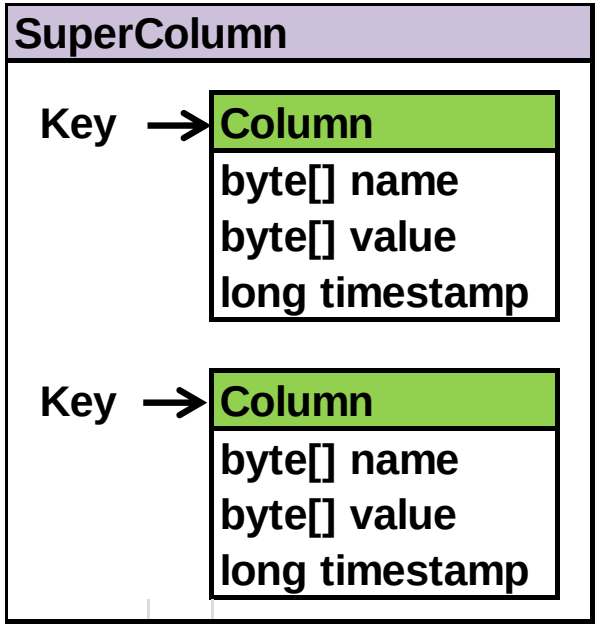
Datenmodell

- Cluster > **Keyspace**



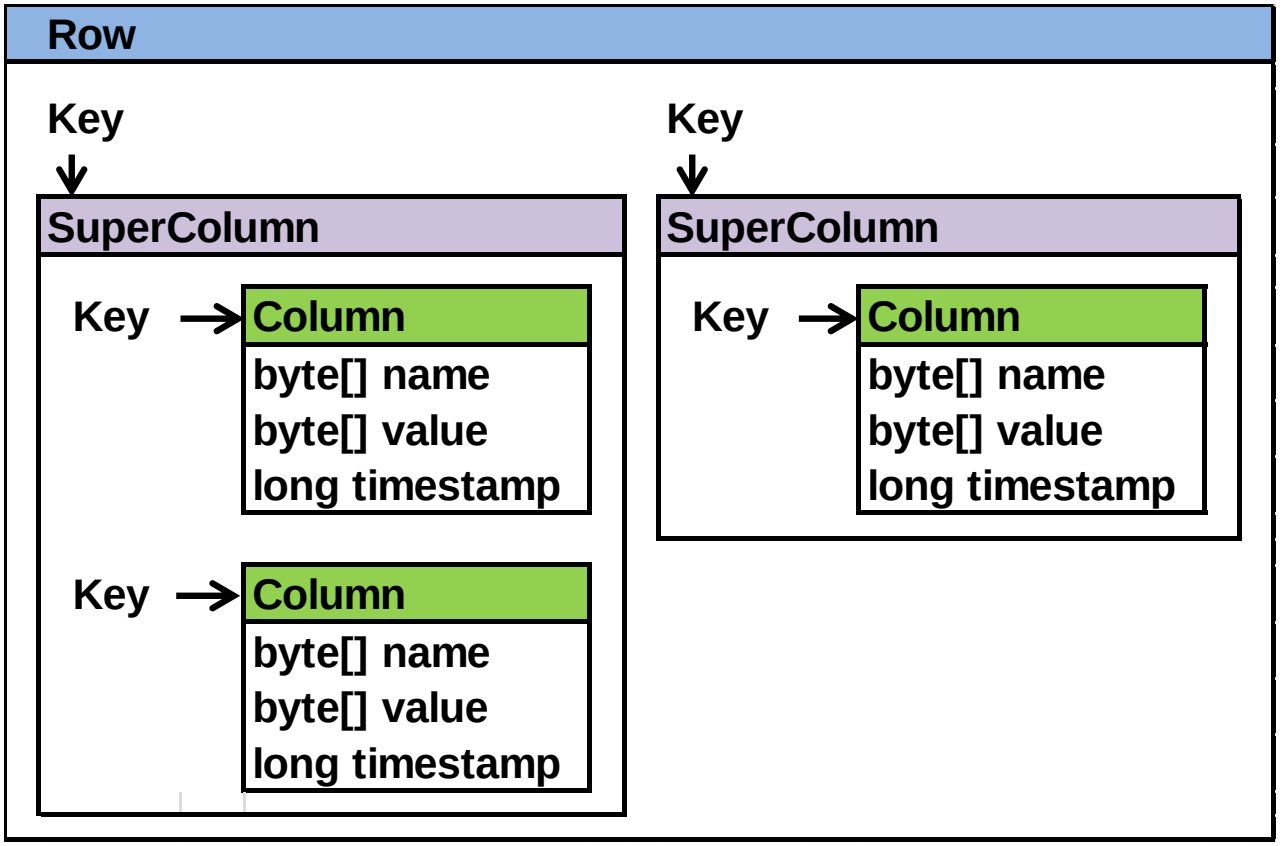
Datenmodell

- Cluster > Keyspace > Column Family > Row > **SuperColumn**



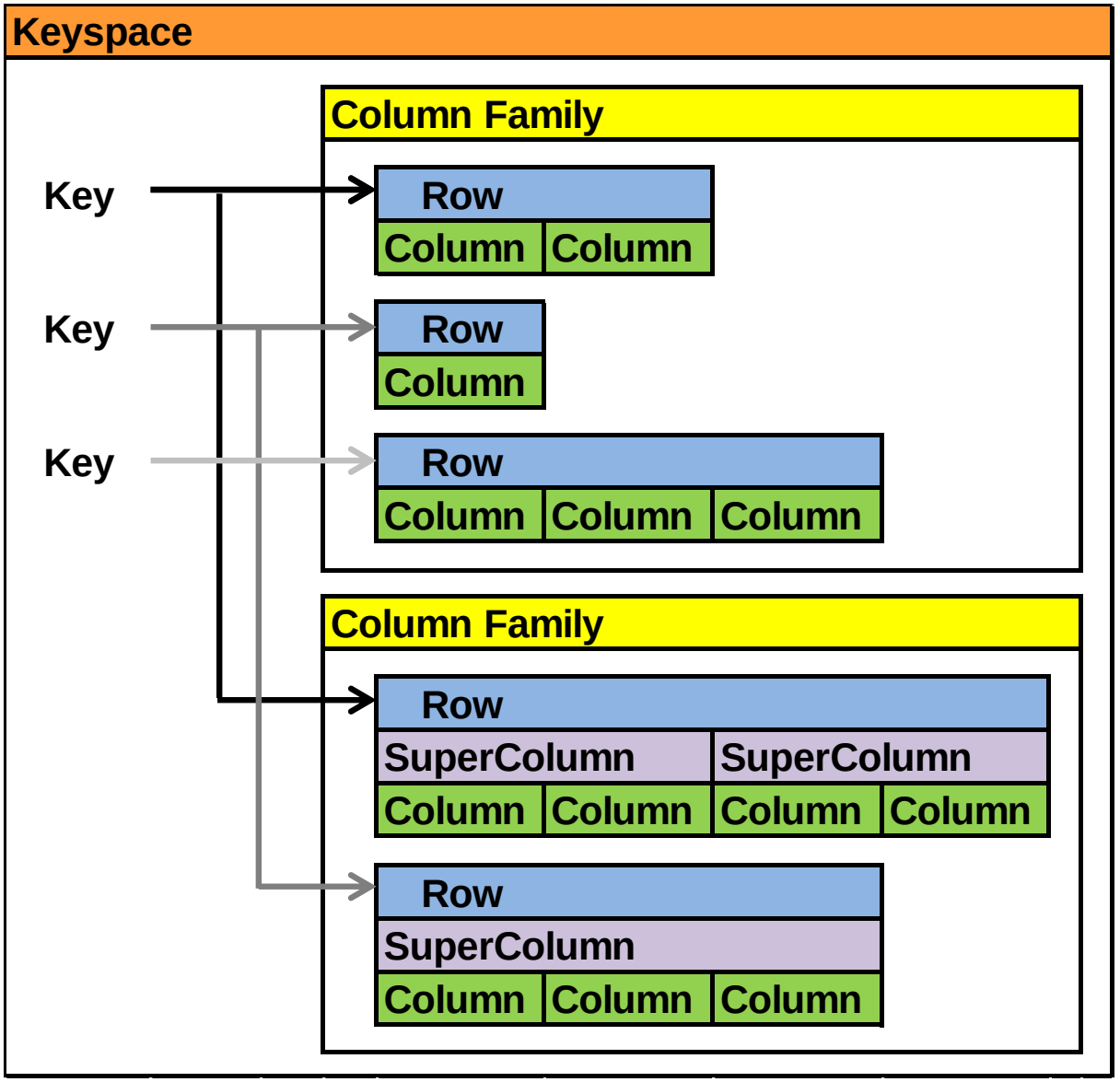
Datenmodell

- Cluster > Keyspace > Column Family > **Row**



Datenmodell

- Cluster > **Keyspace**

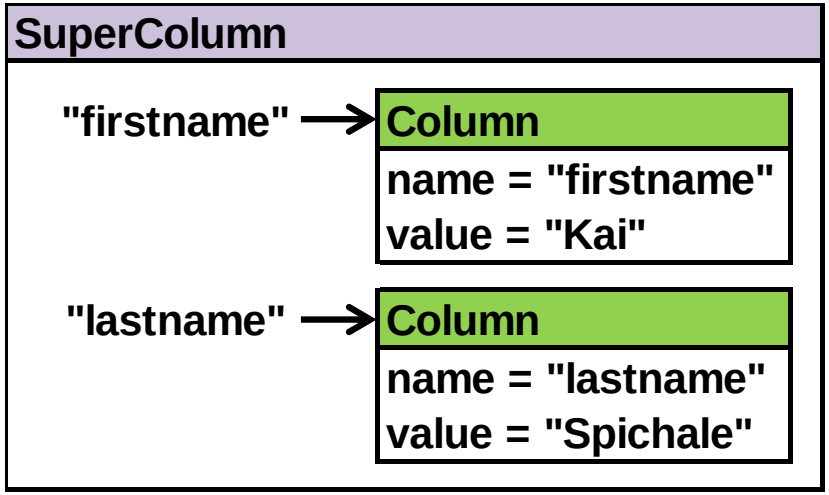


Datenmodellbeispiel

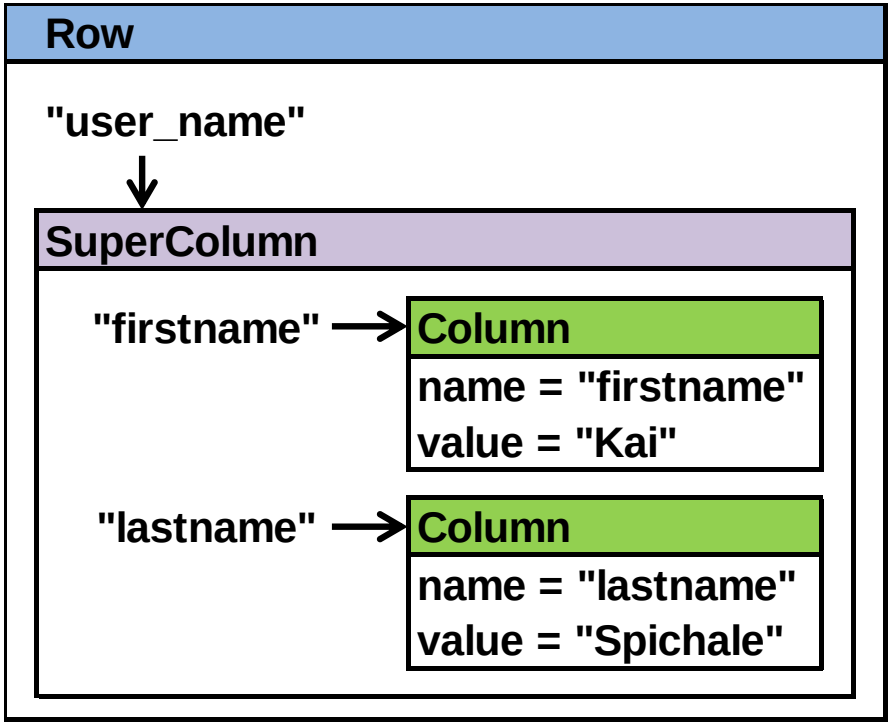
Column
name = "firstname"
value = "Kai"

Column
name = "lastname"
value = "Spichale"

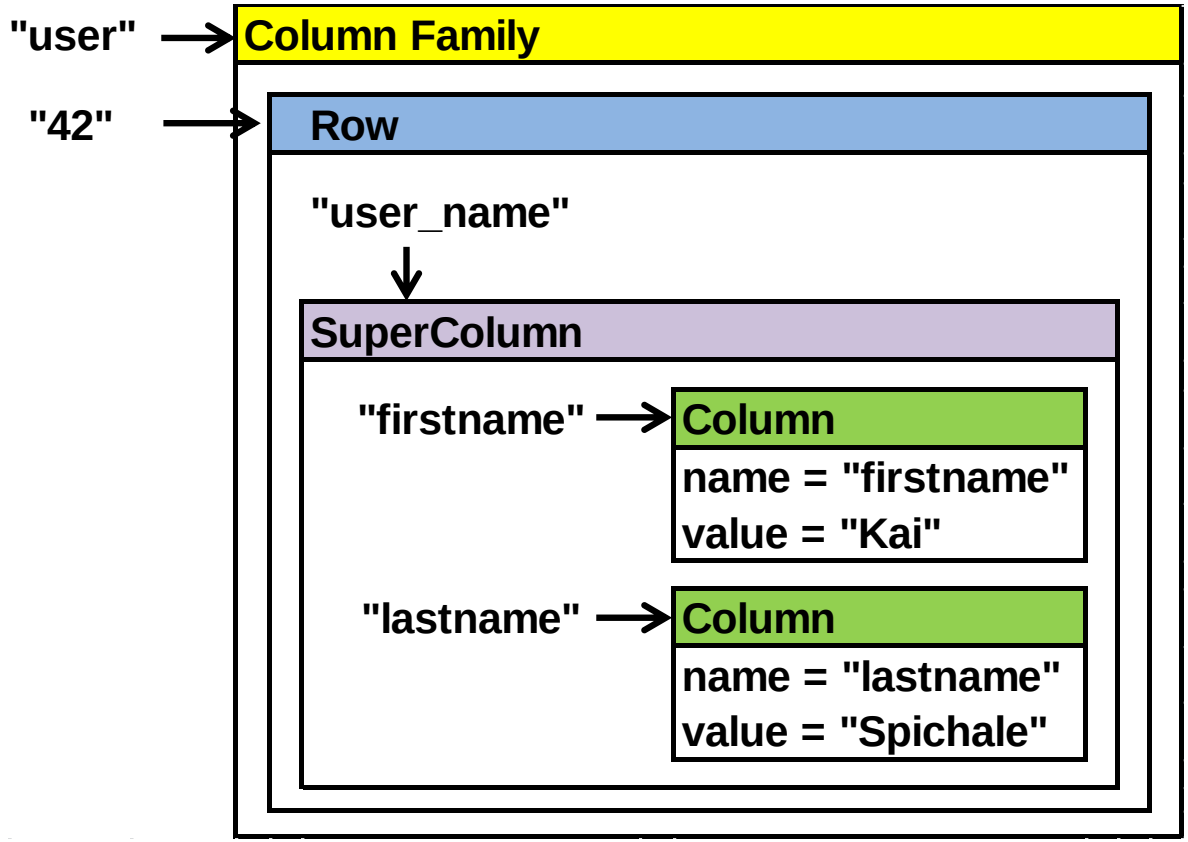
Datenmodellbeispiel



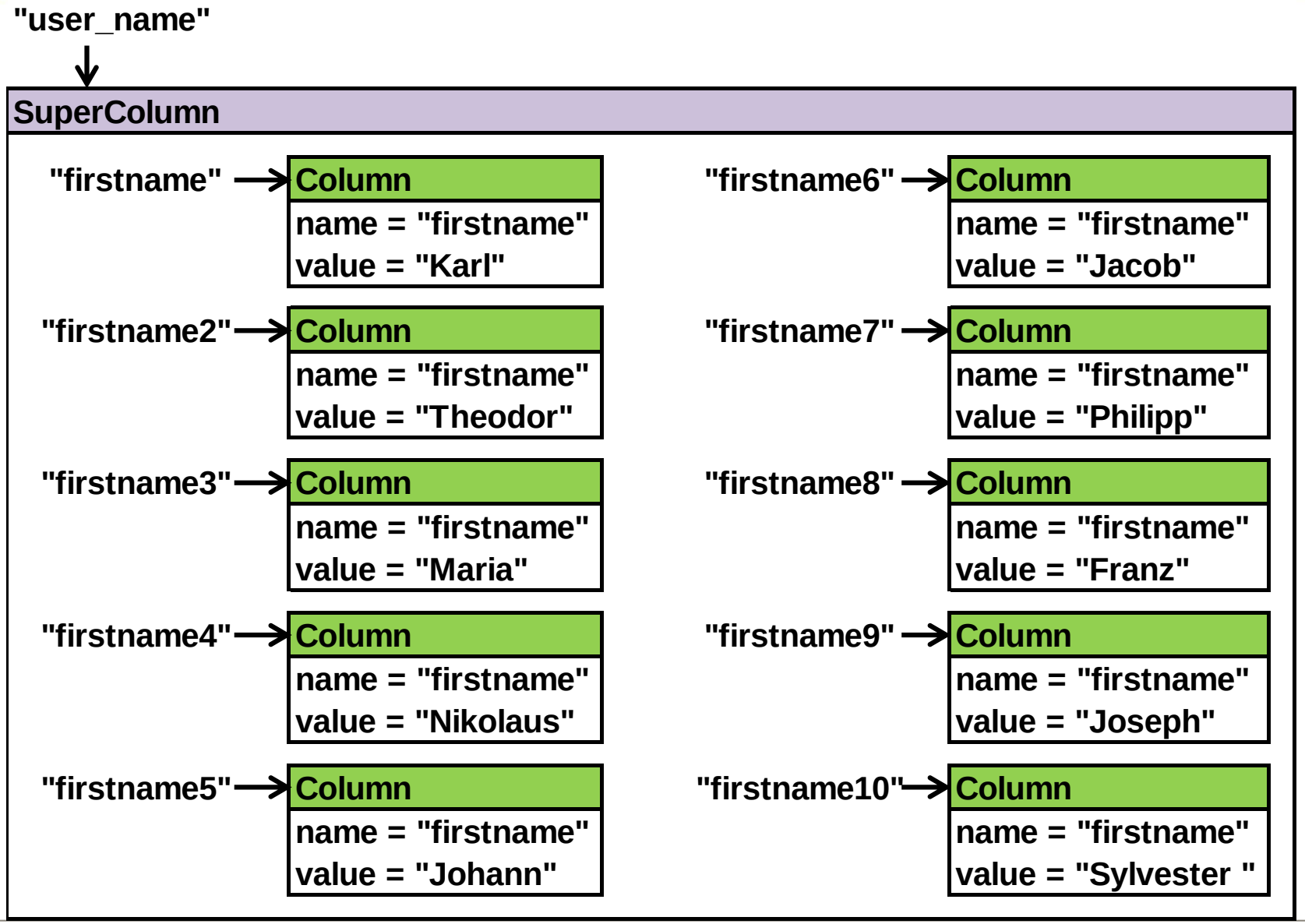
Datenmodellbeispiel



Datenmodellbeispiel



Datenmodellbeispiel – flexible Struktur

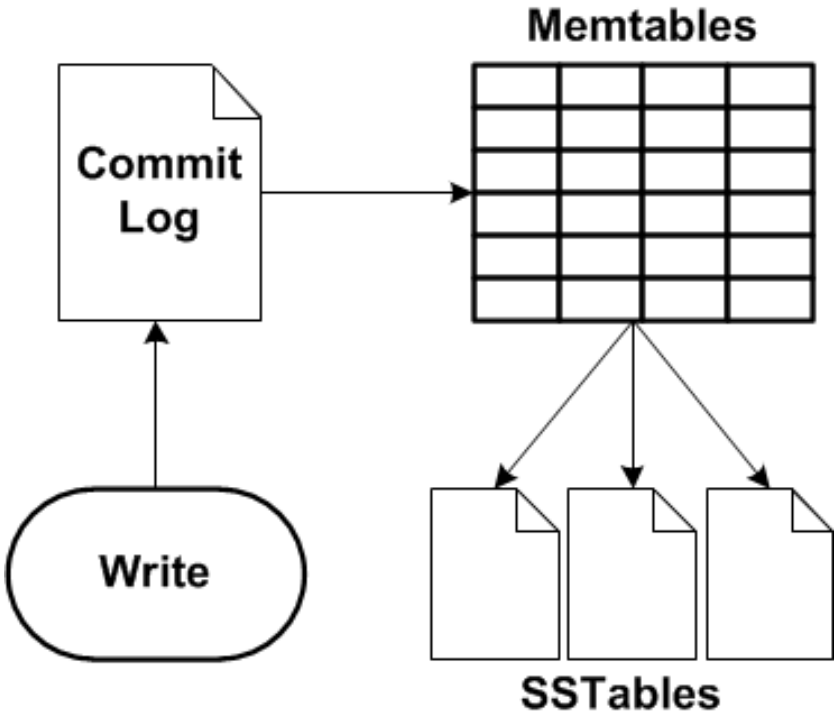


Cassandra vs. MySQL (50GB Daten)

- MySQL
  - 300ms write
  - 350ms read
- Cassandra
  - 0.12ms write
  - 15ms read

Quelle: http://www.odbms.org/download/cassandra.pdf, Zugriff 7.4.2011

Schreibvorgang

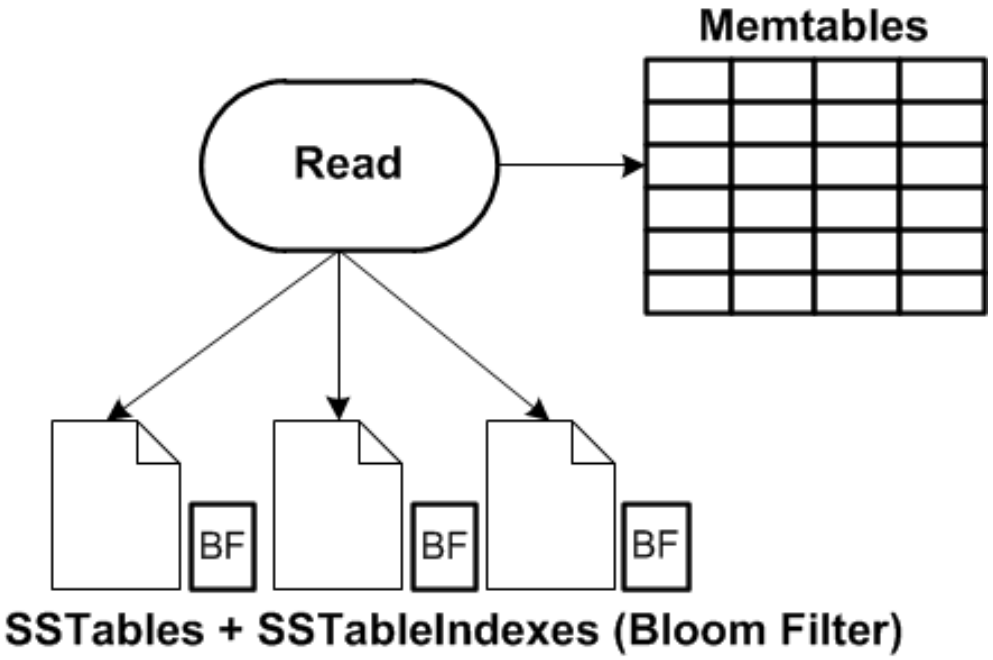


Schreibvorgang

- Client schickt Write Request zu beliebigen Knoten im Cluster
- Write Request wird sequentiell ins lokale Disk Commit Log geschrieben
- Partitioner bestimmt verantwortliche Knoten
- Verantwortliche Knoten erhalten Write Request und schreiben in lokales Logdatei
- Memtables (Write-back Cache) werden aktualisiert
- Flush auf Festplatte in SSTable und SSTableIndex

- Eigenschaften:
  - Kein Lesen, keine Suche, keine Locks
  - Sequentieller Festplattenzugriff
  - Atomare Vorgang für eine ColumnFamily
  - „Always Writable“ (d.h. akzeptiert auch Write Requests bei partiellen Failure)

Lesevorgang



Lesevorgang

- Client schickt Read Request zu beliebigen Knoten im Cluster
- Partitioner bestimmt verantwortliche Knoten
- Warten auf R Antworten
- Warten auf N – R Antworten für Read Repair im Hintergrund

- Eigenschaften:
  - Liest mehrere SSTables
  - Langsamer als Schreiben
  - Skaliert sehr gut

Client API

- „The API is horrible and it produces pointless verbose code in addition to being utterly confusing.“

- „The RCP interface is baroque, and too tightly coupled to Cassandra‘s internals.“

Client API

- Cassandra:
  - Thrift-Interface bietet für viele Sprachen (Python, Ruby, Java, PHP) die API
  - RPC framework
  - CQL ab Version 0.8 (SQL-like)

- Aber besser sind high-level Client Libraries:
  - Java: Pelops, Hector
  - Python: Pycassa
  - Ruby: Cassandra

- Object Mapper:
  - Kundera: **JPA 1.0** Implementierung für Cassandra
  - **Hecotor**: nicht JPA-complaint, kein Entity Relationship Support

Hector

- High-level Cassandra Client
- Open Source: https://github.com/rantav/hector
- Queries:
  - Ein Request bezieht sich auf einen Keyspace und eine ColumnFamily
  - Einfache Requests mit ColumnQuery / SuperColumnQuery
  - *SliceQuery zur Abfrage von Columns, SuperColumns und Sub-Columns
    - Column Range
    - Row Range
- Sekundärer Index:
  - Abfragen mit IndexedSlicesQuery

Hector – ein Beispiel

```
Cluster myCluster = HFactory.getOrCreateCluster("MyCluster",
  "192.168.178.37:9160");

Keyspace keyspace = HFactory.createKeyspace("MyKeyspace",
  myCluster);

StringSerializer se = StringSerializer.get();
```

Hector – ein Beispiel

```
List<HColumn<String, String>> subColumns = new
  ArrayList<HColumn<String, String>>();

subColumns.add(HFactory.createColumn("firstname", "Kai", se, se));
subColumns.add(HFactory.createColumn("lastname", "Spichale", se, se));

HSuperColumn<String, String, String> superColumn = HFactory
  .createSuperColumn("UserName", subColumns, se, se, se);

Mutator<String> mutator = HFactory.createMutator(keyspace, se);
mutator.insert("rowKey1", "User", superColumn);
```

Hector – ein Beispiel

```
SuperColumnQuery<String, String, String, String> query = HFactory
   .createSuperColumnQuery(keyspace, se, se, se, se);

QueryResult<HSuperColumn<String, String, String>> queryResult = query
   .setColumnFamily("User").setKey("rowKey1")
   .setSuperName("UserName");

queryResult.execute();

HSuperColumn<String, String, String> hSuperColumn = queryResult.get();
```

Einschränkungen

- Keine Joins
- Kein ORDERED BY, GROUP BY, LIKE
- Kein kaskadierendes Löschen
- Keine referenzielle Integrität

- Konsequenzen für Datenmodellentwurf
  - Bereits beim Entwurf des Datenmodells müssen alle Abfragen identifiziert sein
  - Zusätzliche Abfragepfade können schwer hinzugefügt werden
  - Denormalisierung, sodass jeder Request mit einer oder mehreren Zeilen einer ColumnFamily beantwortet werden kann
  - Client macht Joins

Fazit

- **Hochskalierbare Datenbank, kann riesige Datenmengen speichern**
- **Einfache Administration**

- **Gewöhnungsbedürftiges Modell**
- **Eventually Consistent**

- **Keine Standard-Client-Library**
- **API-Veränderungen**
- **Upgrade**

Vielen Dank für Ihre Aufmerksamkeit!

Fragen?

adesso business. people. technology.

Wir suchen Sie als

- Software-Architekt (m/w)
- Projektleiter (m/w)
- Senior Software Engineer (m/w)

jobs@adesso.de
www.AAAjobs.de