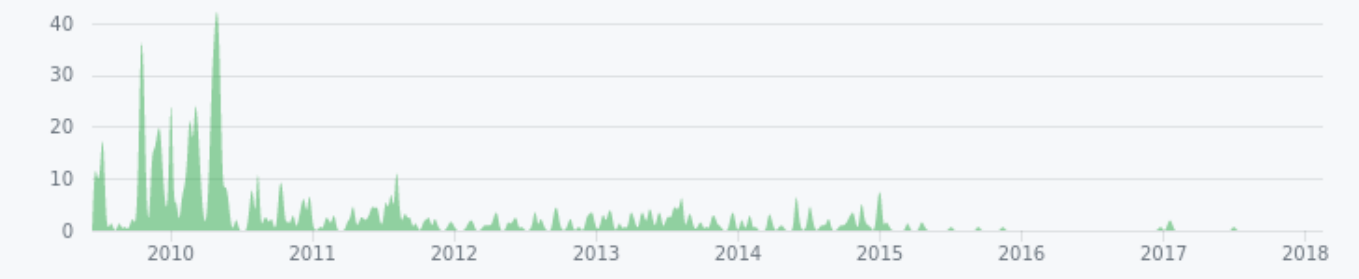
Inhaltsverzeichnis: Key-Value Stores

Riak

- Verteilter Key-Value Store mit Erweiterungen (Links, Suche, MapReduce)
- Open-Source Implementierung von Amazons Dynamo-Konzepten [Dynamo]: https://docs.riak.com/riak/kv/2.2.3/learn/dynamo/index.html
- Unterstützung beliebiger Datentypen: Text, JSON, XML, Bilder, Videos, PDF, ZIP, … (alle MIME-Types)
- Zugriff über HTTP REST oder Programmiersprache
- Geschrieben in Erlang, unterstützt JavaScript
- Zukunft ungewiss [Rich17]
  - 2017: Insolvenz von Basho (kommerzielles Unternehmen hinter Riak)
  - Entwicklung und Community-Support des Open-Source-Projekts stark rückgängig



https://github.com/basho/riak/graphs/contributors

Riak: Demo

- Dokumentation: [Riak]
- Mit Docker Compose : [RiakDocker]
  - Lokales Cluster aus 4 Knoten
  - http://localhost:8098/admin

```
cd docker-home/riak/
nano docker-compose.yml

docker pull basho/riak-kv
docker-compose up -d coordinator
docker-compose scale member=3
```

- Anfragen über HTTP REST: http://SERVER:PORT/riak/BUCKET/KEY

```
curl -X PUT http://localhost:8098/riak/lectures/nosql \
  -H "Content-Type: application/json" \
  -d '{"title" : "NoSQL-Datenbanken",
       "lecturer" : "Johannes Zschache"}'

curl http://localhost:8098/riak/lectures/nosql
```

Riak: Demo

```
curl -X PUT http://localhost:8098/riak/lectures/dwh \
  -H "Content-Type: application/json" \
  -d '{"title" : "Data Warehousing",
       "lecturer" : "Prof. Dr. E. Rahm" ,
       "start" : "20.04.2018"}'

curl -X PUT http://localhost:8098/riak/rooms/hs19 \
  -H "Content-Type: application/json" \
  -d '{"title": "HS 19",
       "type": "Hörsaal",
       "capacity" : 60}'

curl http://localhost:8098/riak?buckets=true
curl http://localhost:8098/riak/lectures?keys=true
```

Riak: Links

- Links werden als Meta-Daten gespeichert

```
curl -X PUT http://localhost:8098/riak/rooms/hs19 \
  -H "Content-Type: application/json" \
  -H "Link: </riak/lectures/nosql>; riaktag=\"donnerstags\", \
            </riak/lectures/dwh>; riaktag=\"freitags\"" \
  -d '{"title": "HS 19", "type": "Hörsaal", "capacity" : 60}'
```

- Link Walking:

```
curl http://localhost:8098/riak/rooms/hs19/_,_,_

curl http://localhost:8098/riak/rooms/hs19/lectures,_,_

curl http://localhost:8098/riak/rooms/hs19/_,donnerstags,_
```

Riak: Erweiterungen

1. Volltextsuche
   - Benötigt durchsuchbare Werte (z.B. JSON, XML, Text)
   - HTTP Solr interface:

```
curl http://localhost:8098/solr/lectures/select?q=lecturer:rahm
```

   - Mehrere Attribute:

```
curl http://localhost:8098/solr/lectures/select\
?q=lecturer:rahm%20title:data&q.op=and
```

   - Sortieren und Anfragen mit Gruppierung möglich

2. Komplexere Anfragen über MapReduce (nur JSON)

```
curl -X POST http://localhost:8098/mapred \
-H "content-type:application/json" -d '{ "inputs":"rooms",
  "query":[
    {"map":{ "language":"javascript", "bucket":"my_functions",
      "key":"map_capacity"}},
    {"reduce":{ "language":"javascript", "bucket":"my_functions",
      "key":"reduce_capacity"}}]}'
```

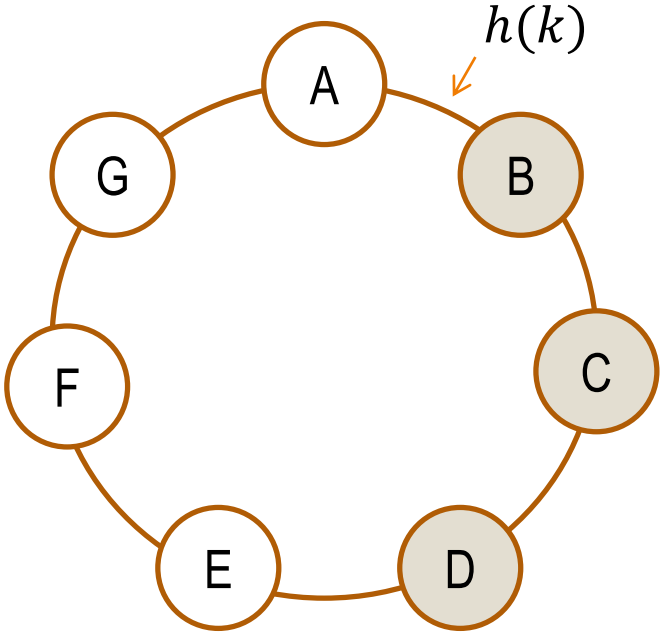
Riak: Architektur

- Verteiltes System
- Flexibel skalierbar: Datenpartitionierung über **Consistent Hashing**
- Multi-Master-Replikation
  - Alle Knoten haben gleiche Funktionalität: Keinen Single-Point-of-Failure
  - Alle Knoten akzeptieren Schreiboperationen
- *Eventually consistent und hochverfügbar*, aber höhere semantische Garantien konfigurierbar über **Read/Write-Quoren**
- Konfliktlösung:
  - Anti-Entropy-Mechanismen
  - Convergent Replicated Data Types
  - Version Vectors

Inhaltsverzeichnis: Key-Value Stores

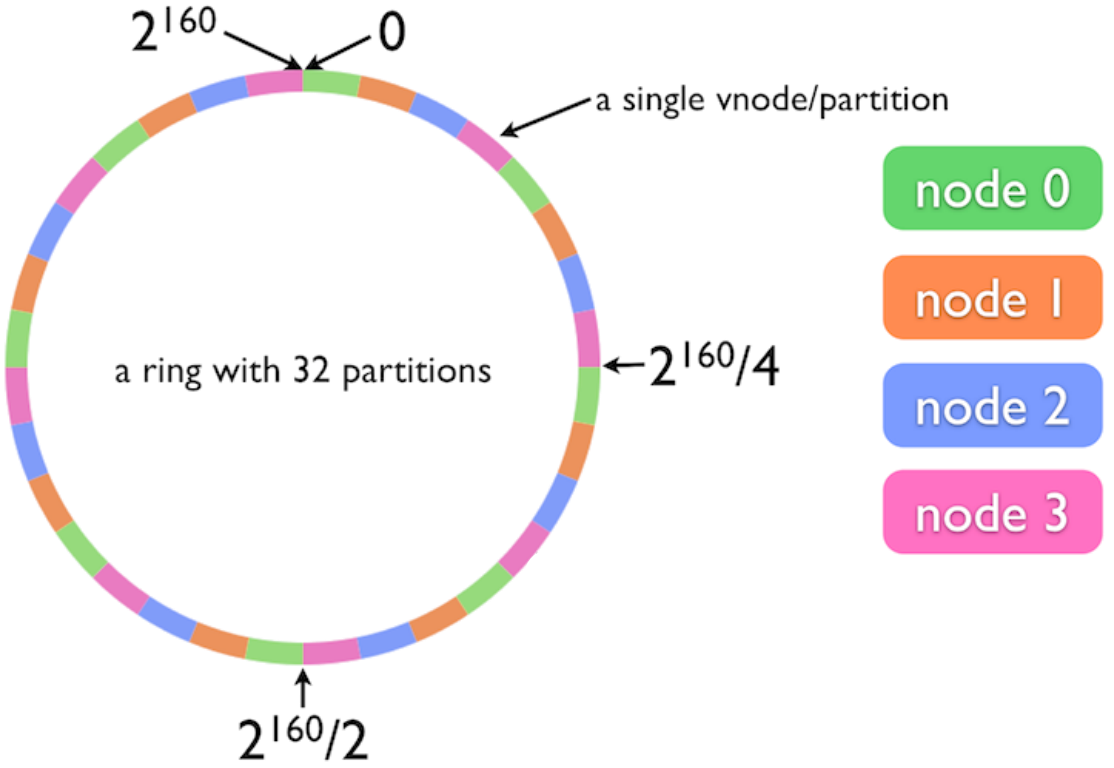
Partitionierung: Consistent Hashing

- Replikationsfaktor N
- Hash-Funktion h mit Wertebereich von 0 bis $2^m - 1$
- Anwendung von h auf Serverknoten-ID: logischer Ring
- Eintrag mit Key k wird auf den N Serverknoten gespeichert, deren Hash-Werte als nächstes $h(k)$ folgt
- Eigenschaften:
  - Gleichmäßige Verteilung der Daten über die Serverknoten
  - Effektive Datenumverteilung bei wechselnder Knotenzahl



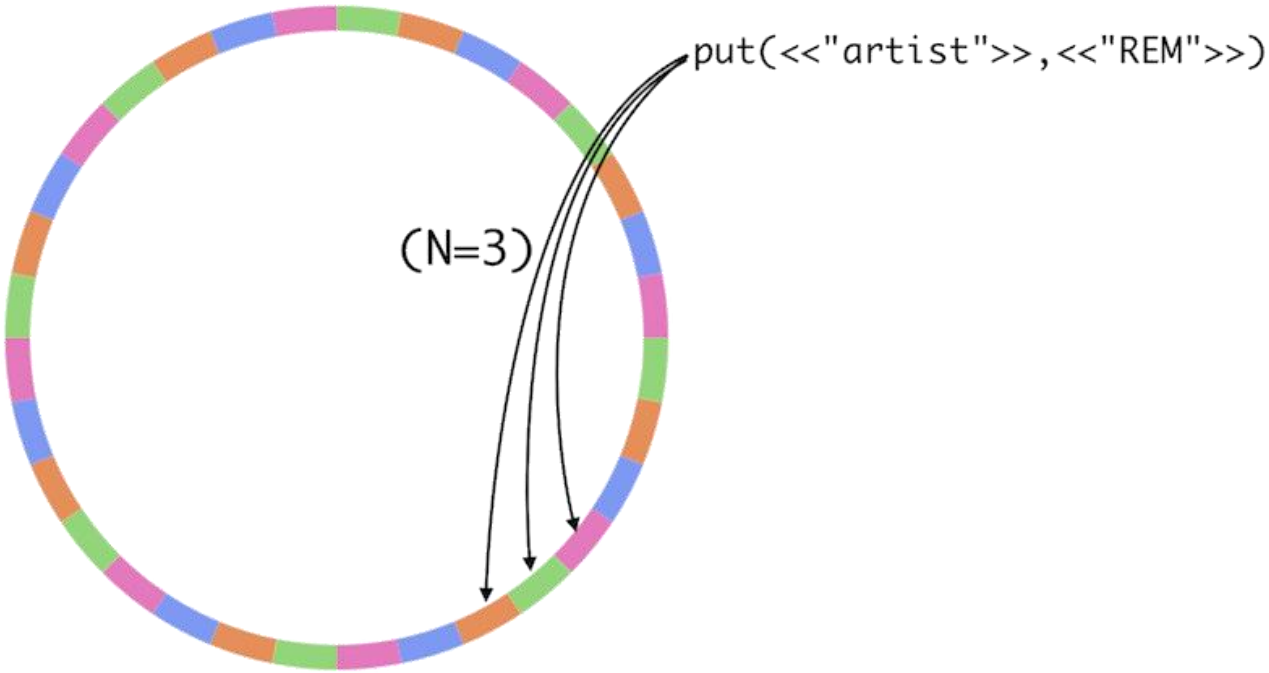
Riak-Ring

- Hash-Zuordnung auf 160-bit Zahl
- Zusätzlicher Lastenausgleich über *virtuelle Knoten*
- Gleichmäßige Aufteilung des Rings in z.B. 32 Bereiche (Vnodes)
- Feste Anzahl an Vnodes zu jedem Zeitpunkt
- Gleichmäßige Aufteilung der Vnodes über die funktionierenden physische Knoten (Nodes)



Riak-Ring

- Schreibanfragen können von jedem Server entgegen genommen werden
  - Weiterleitung an zuständige Bereiche (Vnodes)
  - Anzahl der zuständigen Bereiche über Replikationsfaktor N
- Zustand des Rings über Gossip-Protokoll geteilt



Quelle: https://docs.riak.com/riak/kv/2.2.3/learn/concepts/clusters/index.html

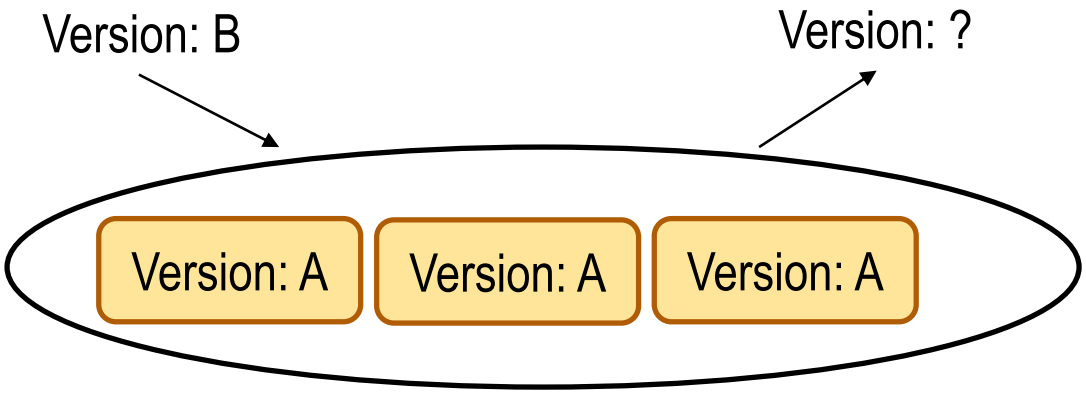
Inhaltsverzeichnis: Key-Value Stores

Read/Write-Quoren

- Riak ist AP (nach CAP-Theorem)
  - Stärkere Konsistenzgarantien im Austausch für geringere Verfügbarkeit möglich
  - Reduzierung der Gefahr alte Werte zu lesen oder Aktualisierungen zu verlieren
- Prinzip:
  - Replikationsfaktor N
  - Read-Quorum R = min. Anzahl der N (virtuellen) Knoten, die beim Lesen ein Ergebnis liefern müssen
    - Rückgabe der Antworten von R Knoten
    - Kann mehrere Versionen zurückliefern
  - Write-Quorum W = min. Anzahl der N (virtuellen) Knoten, die eine Schreib-Operation bestätigen müssen
    - Erfolg, wenn (mindestens) W Knoten antworten
    - Zusätzlich Parameter DW für sicheres Speichern (zunächst nur Buffer in Hauptspeicher)
- Riak: Anpassung von (N,R,W) pro Bucket oder (R,W) pro Anfrage

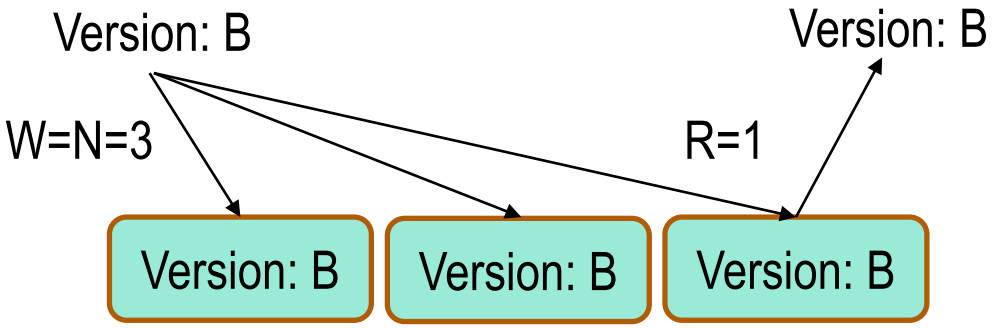
Quorum: Beispiel

- Replikationsfaktor N = 3
- Aktueller Zustand: Version A des Wertes
- Schreibzugriff: Version B
- Anschließend Lesen des Wertes

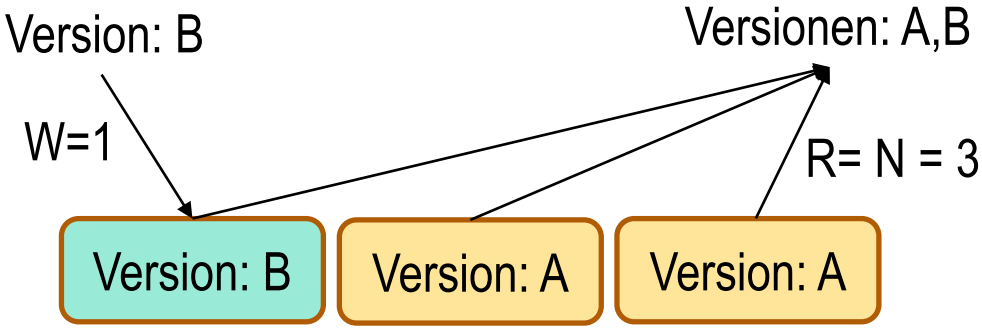


Quorum: Varianten

- Optimierung der Lesezugriffe: R=1, W=N

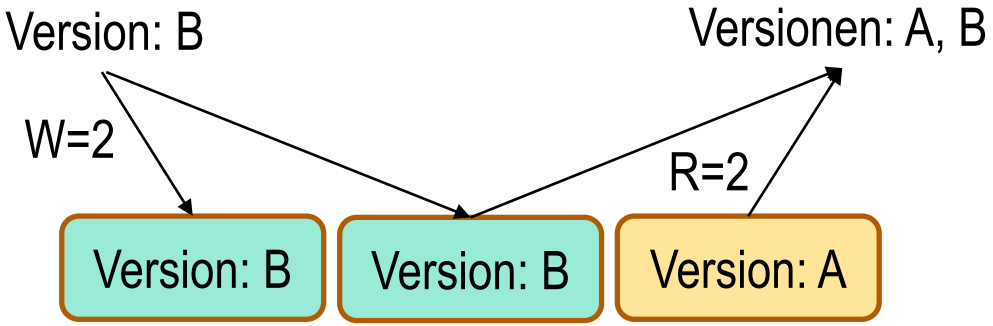


- Optimierung der Schreibzugriffe: R=N, W=1

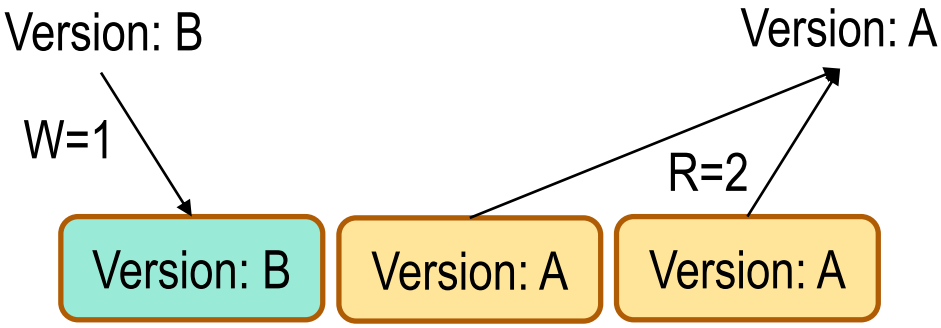


Quorum: Varianten

- **Overlap**: R + W > N
  - Kompromiss zwischen Lese- und Schreibaufwand
  - Mind. ein Knoten liest die aktuelle Version eine Wertes
  - z.B. Majority quorum: R > $\frac{N}{2}$ & W > $\frac{N}{2}$



- Eventually consistent: R + W ≤ N



Riak: Quorum

```
curl -X PUT http://localhost:8098/riak/lectures \
  -H "Content-Type: application/json" \
  -d '{"props":{"n_val":3, "r":2, "w":2 }}'
```

- Anzahl der Replikate: `n_val`
- Mögliche Werte für `r` und `w`:
  - Integer Value ≤ `n_val`
  - `"one"`
  - `"all"`
  - `"quorum"` (majority)
- Pro Anfrage

```
curl http://localhost:8098/riak/lectures/nosql?r=all
curl -X PUT http://localhost:8098/riak/lectures/dbs2?w=0 \
  -H "Content-Type: application/json" \
  -d '{"title" : "Datenbanksysteme 2"}'
```

Hinted Handoff / Sloppy Quorum:

- Erhöhung der Verfügbarkeit
- Hinted Handoff
  - Wenn ein Server nicht erreichbar ist, wird Anfrage an anderen Server weitergereicht
  - Dieser Server ist (noch) kein Replikat des relevanten Bereiches
  - Leseanfragen können von Hinted Server (vorerst) nicht beantwortet werden
  - Schreibanfragen werden an zuständigen Server geleitet, wenn wieder erreichbar
- Sloppy Quorum
  - Hinted Handoff wird auch für Erfüllung der Read-/Write-Quoren verwendet
  - R = 1: Hinted Server wird Anfrage am schnellsten beantworten (Key not found)
  - R = 2: Aktueller Wert trotz Ausfalls eines Servers
- Deaktivierung des Sloppy Quorum über *Primary* Read-/Write-Quoren
  - PR = 2: mind. 2 echte Replikate müssen Wert lesen (keine Hinted Server)
  - PW = 3: mind. 3 echte Replikate müssen Wert schreiben (höhere Gefahr, dass Schreibbefehl nicht ausführbar als bei W = 3)

Inhaltsverzeichnis: Key-Value Stores

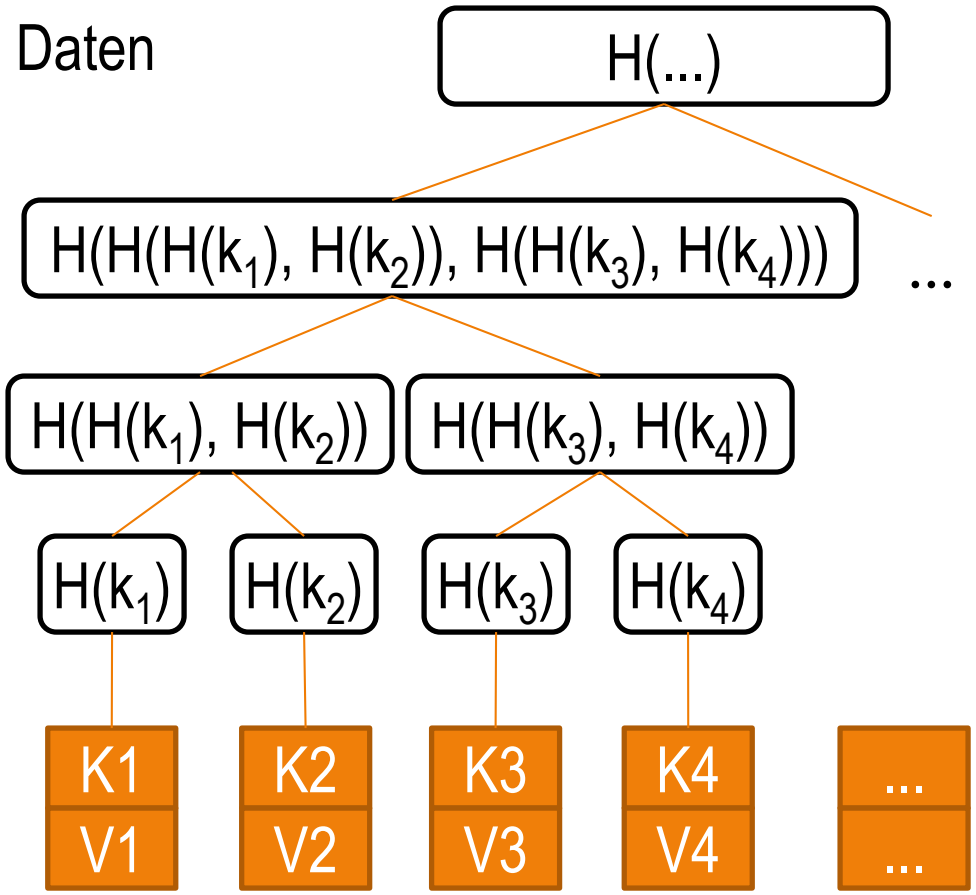
Erkennung von Konflikten

- Konflikte zwischen Replikaten sind Folgen von Serverausfällen oder dem gleichzeitigen Zugriff durch unterschiedliche Nutzer
- Riak unterstützt zwei Mechanismen zur Erkennung von Konflikten

1. Read Repair
   - Passiver Mechanismus
   - Starten von Reparaturprozessen bei Leseanfrage und erkannten Konflikten
   - Nachteile:
     - Konflikte bleiben unentdeckt bis erneutes Lesen des Wertes
     - Höhere Wahrscheinlichkeit des Lesens alter Werte
2. Active Anti-Entropy
   - Kontinuierlicher Prozess im Hintergrund vergleicht Zustand der Replikate
   - Vergleich über den Austausch von Hash-Bäumen (Merkle Tree)
   - Nachteil: Höhere CPU-Auslastung

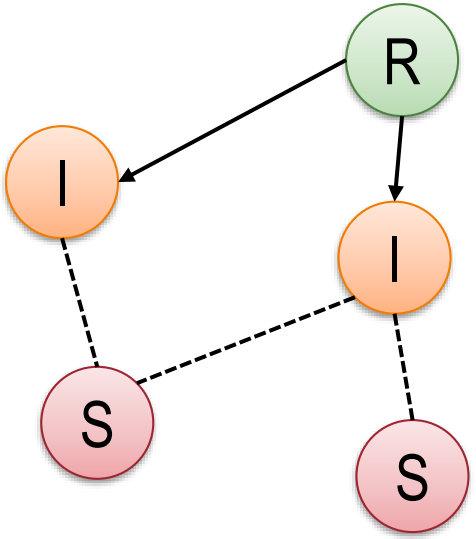
Synchronisation über Hash-Bäume

- Hash-Baum (Merkle Tree) für Bereich (Vnode)
  - Blätter sind Hash-Werte der Daten
  - Eltern-Knoten sind Hash-Werte der Kind-Knoten-Werte
- Vorteil: Minimaler Austausch von Daten
  - Überprüfung ob zwei Replikate synchron sind über Wurzel-Knoten
  - Identifikation nicht-synchroner Teile: Rekursive Analyse der Teilbäume
- Nachteil: Neuberechnung bei Datenänderungen
- Riak: Speicherung der Hash-Bäume auf Festplatte



Gossip/Epidemische Protokolle

- Peer-to-Peer Algorithmen um Nachrichten (z.B. Erreichbarkeit, Hash-Werte, Informationen über Partitionierung der Daten) effizient über das gesamte Netzwerk von Servern zu verbreiten
- Vollständige Kommunikation wächst mit der Anzahl der Server, benötigt globale Informationen und ist fehleranfällig
- **Gossip**: Senden der Nachricht an einige ausgewählte Server und diese leiten die Nachricht an andere Server weiter
- SIR Modell: Server ist in einem von 3 Zuständen
  - Susceptible: Nachricht noch nicht erhalten
  - Infected: Weiterleitungen
  - Removed: keine Weiterleitungen
- Übergang zu Removed: probabilistisch, Zähler, Feedback



Konfliktlösung

- Riak: verschiedene Möglichkeiten, Konflikte zu lösen

1. Automatische Konfliktlösung

- Verwendung von Timestamps (default)
  - Jüngster Eintrag gewinnt
  - Problem: Uhren der Replikate sind nicht notwendig synchron
- „Last-write-wins“

```
curl -X PUT http://localhost:8098/riak/lectures \
  -H "Content-Type: application/json" \
  -d '{"props":{"last_write_wins":true}}'
```

  - Jeder Schreibbefehl wird direkt ausgeführt (ohne Beachtung des Timestamp)
  - Nur nützlich, wenn keine gleichzeitigen Zugriffe

Konfliktlösung: CRDT

2. Automatische Lösung über CRDT (Convergent Replicated Data Types)

- Spezielle Datentypen mit eingeschränkten Operationen und speziellen Regeln der Konfliktlösung

CRDT	Beschreibung	Operationen	Konfliktlösung
Counter	Ganze Zahl	Inkrement/Dekrement	-
HyperLog Log	Zählen eindeutiger Elemente	Element hinzufügen Zähler abfragen	-
Set	Menge	Hinzufügen/Entfernen von Elementen	„Hinzufügen“ gewinnt über „Entfernen“
Map	Assoziatives Datenfeld	Hinzufügen/Entfernen von Feldern	„Hinzufügen“ gewinnt über „Entfernen“
Flag	Zweiwertig (nur innerhalb einer Map verwendbar)	Ein-/Ausschalten	„Ein“ gewinnt über „Aus“
Register	Binärer Datentyp (nur innerhalb einer Map verwendbar)	Setzen	Verwendung des Timestamp

Konfliktlösung: Manuell

3. Manuelle Lösung über Anwendung

```
curl -X PUT http://localhost:8098/riak/lectures \
  -H "Content-Type: application/json" \
  -d '{"props":{"allow_mult":true}}'
```

- Erneutes Beschreiben eines Schlüssels erzeugt *Siblings;* Beispiel:

```
curl -X PUT http://localhost:8098/riak/lectures/bdprak \
  -H "Content-Type: application/json" \
  -d '{"title" : "Big Data Praktikum",
       "lecturers" : ["Rahm"]}'
curl -X PUT http://localhost:8098/riak/lectures/bdprak \
  -H "Content-Type: application/json" \
  -d '{"title" : "Big Data Praktikum",
       "lecturers" : ["Rahm", "Zschache"]}'
curl http://localhost:8098/riak/lectures/bdprak
```

Konfliktlösung: Manuell

- Versionen:

```
curl -i http://localhost:8098/riak/lectures/bdprak \
  -H "Accept: multipart/mixed"
```

- Manuelle Lösung über kausalen Kontext (Vektoruhr)

```
curl -X PUT http://localhost:8098/riak/lectures/bdprak \
  -H "X-Riak-Vclock: a85hYGBgzGDKBVI8tgFmdg0/vx2FCCUy5bE" \
  -H "Content-Type: application/json" \
  -d '{"title" : "Big Data Praktikum",
       "lecturers" : ["Rahm", "Zschache"]}'
```

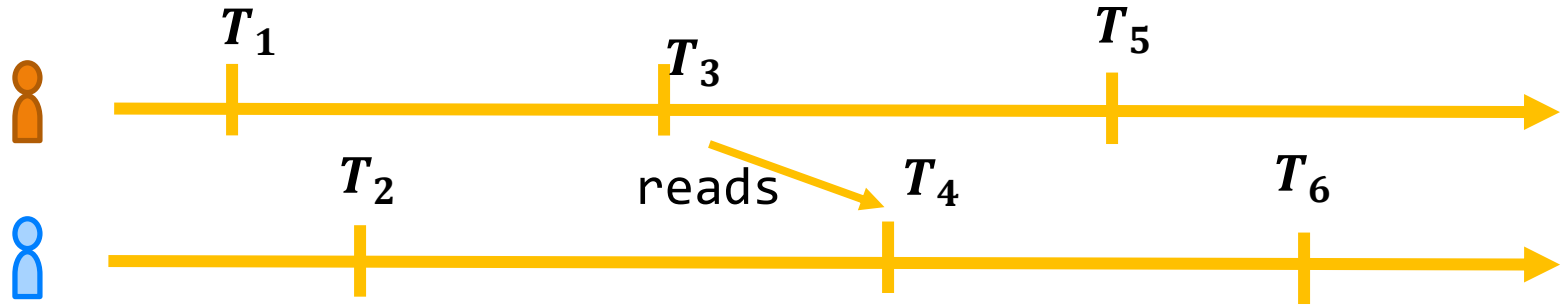
- Gefahr des *Sibling Explosion*
  - Häufiges gleichzeitiges Beschreiben eines Schlüssels ohne manuelle Lösung
  - Probleme: Hohe Latenzzeiten, Out-of-Memory-Fehler oder Ausfall des Servers

Konfliktlösung: Vektoruhren

- Vektoruhren helfen bei der Bestimmung der kausalen Reihenfolge von Ereignissen (z.B. Transaktionen) in verteilten Systemen

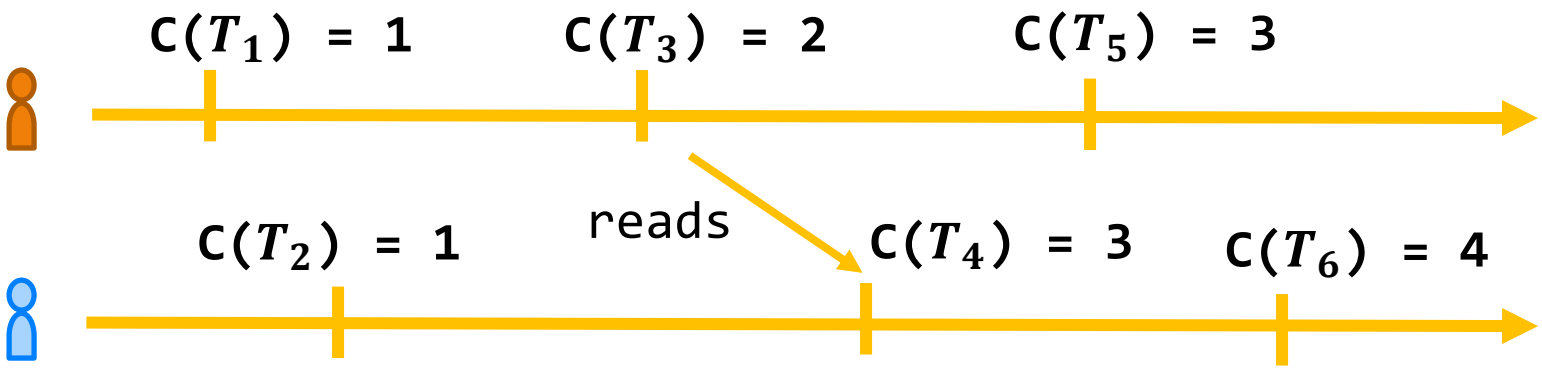
Wiederholung: Eine Operation T_2 ist ***kausal abhängig*** von T_1 ($T_1 \rightarrow T_2$), falls
- der selbe Nutzer führte T_1 vor T_2 aus,
- T_2 liest Werte, die durch T_1 geschrieben wurden, oder
- es gibt eine Operation T_3 mit $T_1 \rightarrow T_3$ und $T_3 \rightarrow T_2$ (Transitivität).

- Keine globale Ordnung: zwei Operationen ohne kausale Beziehung heißen „*concurrent*“
- z.B. $\boldsymbol{T_1}$ und $\boldsymbol{T_2}$ oder $\boldsymbol{T_5}$ und $\boldsymbol{T_6}$ sind concurrent, aber $\boldsymbol{T_6}$ ist kausal abhängig von $\boldsymbol{T_1}$ und $\boldsymbol{T_2}$:



Lamport-Uhr

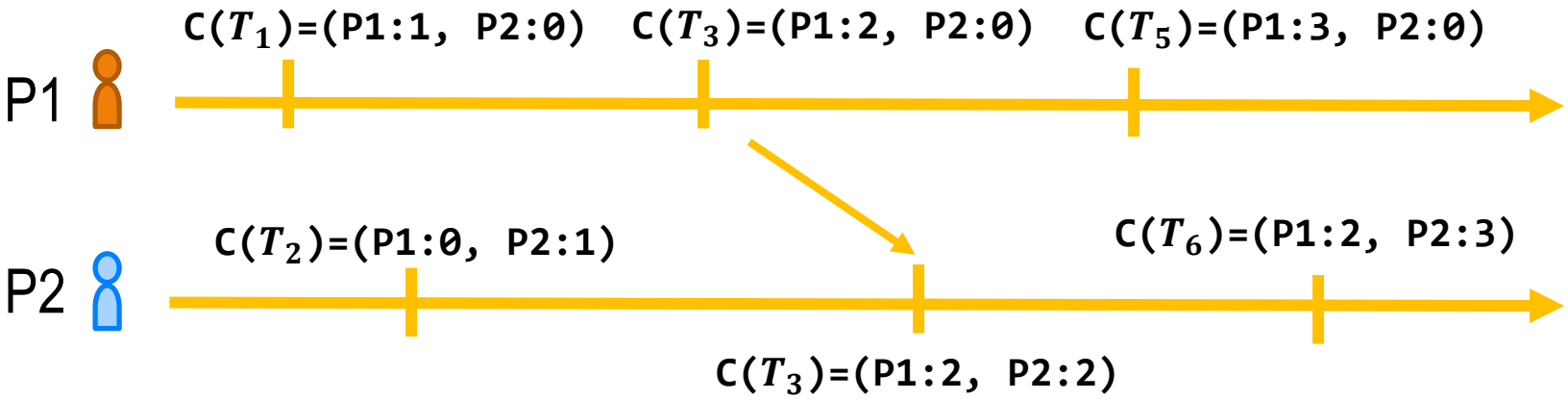
- Jeder Prozess zählt seine Transaktionen
  - Dieser Zähler (C) wird jeder Transaktion beigefügt
  - Beim Empfang eines anderen Zählers durch das Lesen eines Eintrags, wird der eigene Zähler auf das Maximum der beiden Zähler gesetzt (plus Eins)
- Beispiel:



- *Weak Clock Property*: Falls $T_1 \rightarrow T_2$, dann $C(T_1) < C(T_2)$

Vektoruhr

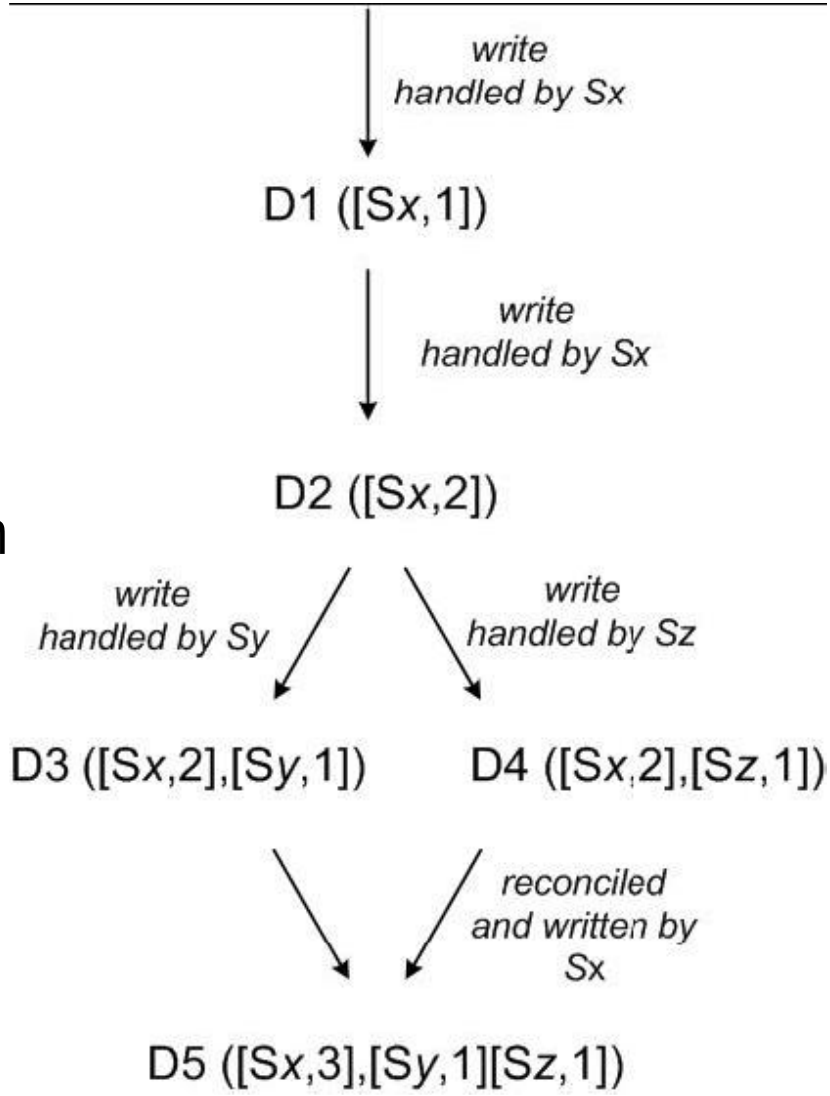
- Vektor von Zählern, ein Eintrag pro Prozess
  - Der Vektor (C) wird jeder Transaktion beigefügt
  - Beim Lesens eines Wertes, wird der eigene Vektor auf das elementweise Maximum der beiden Zähler gesetzt (plus Eins im eigenen Eintrag)
- Beispiel:



- Sei $C(T_1) < C(T_2)$ genau dann, wenn
  - $C(T_1)_i \leq C(T_2)_i$ für alle i und
  - $C(T_1)_i < C(T_2)_i$ für mind. ein i
- Strong Clock Property: $T_1 \rightarrow T_2$ genau dann, wenn $C(T_1) < C(T_2)$

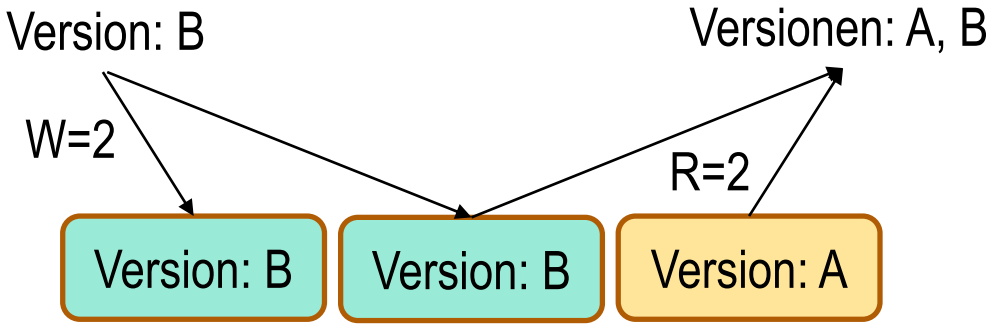
Konfliktlösung über Vektoruhren

- Verwendung von Vektoruhren zur Synchronisation zwischen Replikaten
  - Jeder Schlüssel hat eigene Vektoruhr
  - Jeder schreibende Nutzer bekommt einen Vektoreintrag, der dessen Operationen zählt
- Alle Befehle beinhalten Vektoruhr
- Bei Synchronisation zwischen Replikaten und beim Schreiben durch Nutzer: Vergleich zweier Versionen v_1 und v_2 mit Vektoruhren c_1 und c_2
  - $c_1 < c_2$: v_2 ist aktuell
  - $c_2 < c_1$: v_1 ist aktuell
  - Sonst: „concurrent“ bzw. Konflikt
- Konfliktlösung, z.B. über Timestamps oder Anwendung



Vektoruhren und Konsistenz

- Vektoruhren erlauben die Bestimmung der kausalen Reihenfolge aller Transaktionen und ermöglichen somit *kausale Konsistenz*
- In Verbindung mit Read-/Write-Quoren: sogar Linearisierbarkeit möglich
  - R + W > N und 2W > N
  - Verwendung der Vektoruhren für die Wahl der aktuelleren Version bei Leseoperationen und Ablehnung von inkompatiblen Schreiboperationen



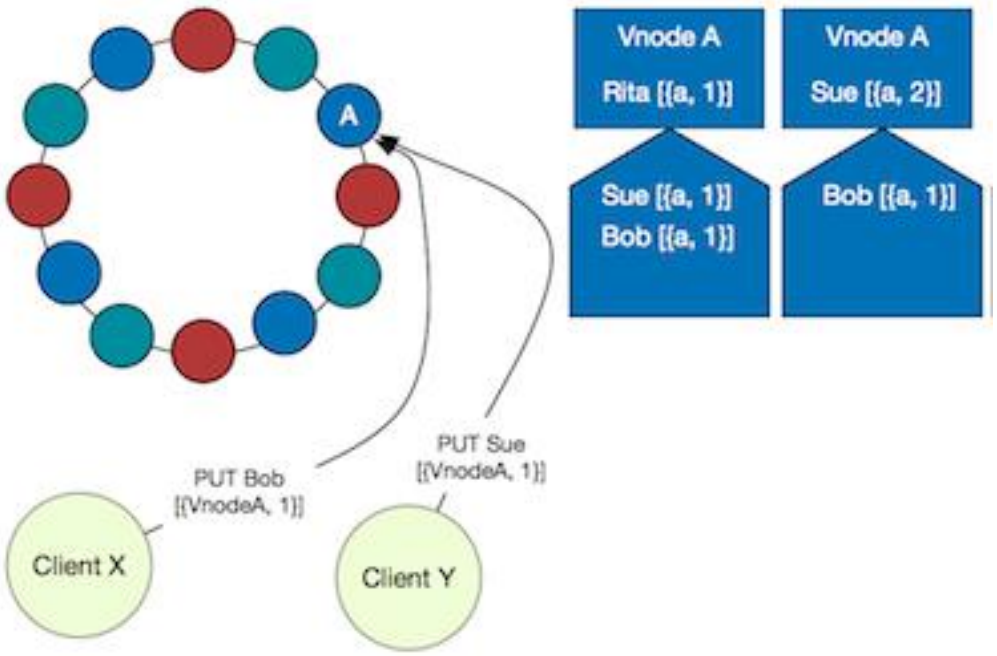
- Riak: Verwendung der Vektoruhren zur Synchronisation einzelner Objekte
  - Keine Umsetzung der Sitzungsgarantien → Keine kausale Konsistenz (ohne geeignete Read-/Write-Quoren)
  - Außerdem: *Approximative Vektoruhren*

Probleme mit Vektoruhren

- Schlechte Skalierbarkeit
  - Vektordimension wächst mit Anzahl der schreibenden Prozesse
  - Vektorgröße wächst mit Anzahl der Schreibbefehle
  - Nachrichtengröße wächst mit Vektorgröße
- Verschiedene Strategien um Skalierbarkeit zu verbessern
  - Pruning: Löschen von Einträge, falls Vektor zu viele Dimensionen und/oder diese lange nicht aktualisiert wurden
  - Resetting: Zurücksetzen auf Null nach Erreichen eines Schwellenwerts
  - Incremental: Nur Unterschiede seit letzten Kommunikation werden gesendet
  - **Approximative Vektoruhren**
  - **Dotted Version Vectors**

Riak: Approximative Vektoruhren

- Verwendung der Vnodes anstatt Nutzer
- Größe des Vektors = N (Replikationsfaktor)



Quelle: http://basho.com/posts/technical/vector-clocks-revisited/

- Falls lokale Vektoruhr (eines Replikats) größer als eingehende Vektoruhr (oder „concurrent“), Hinzufügen des eingehenden Wertes als *Sibling*

Riak: Vektoruhren

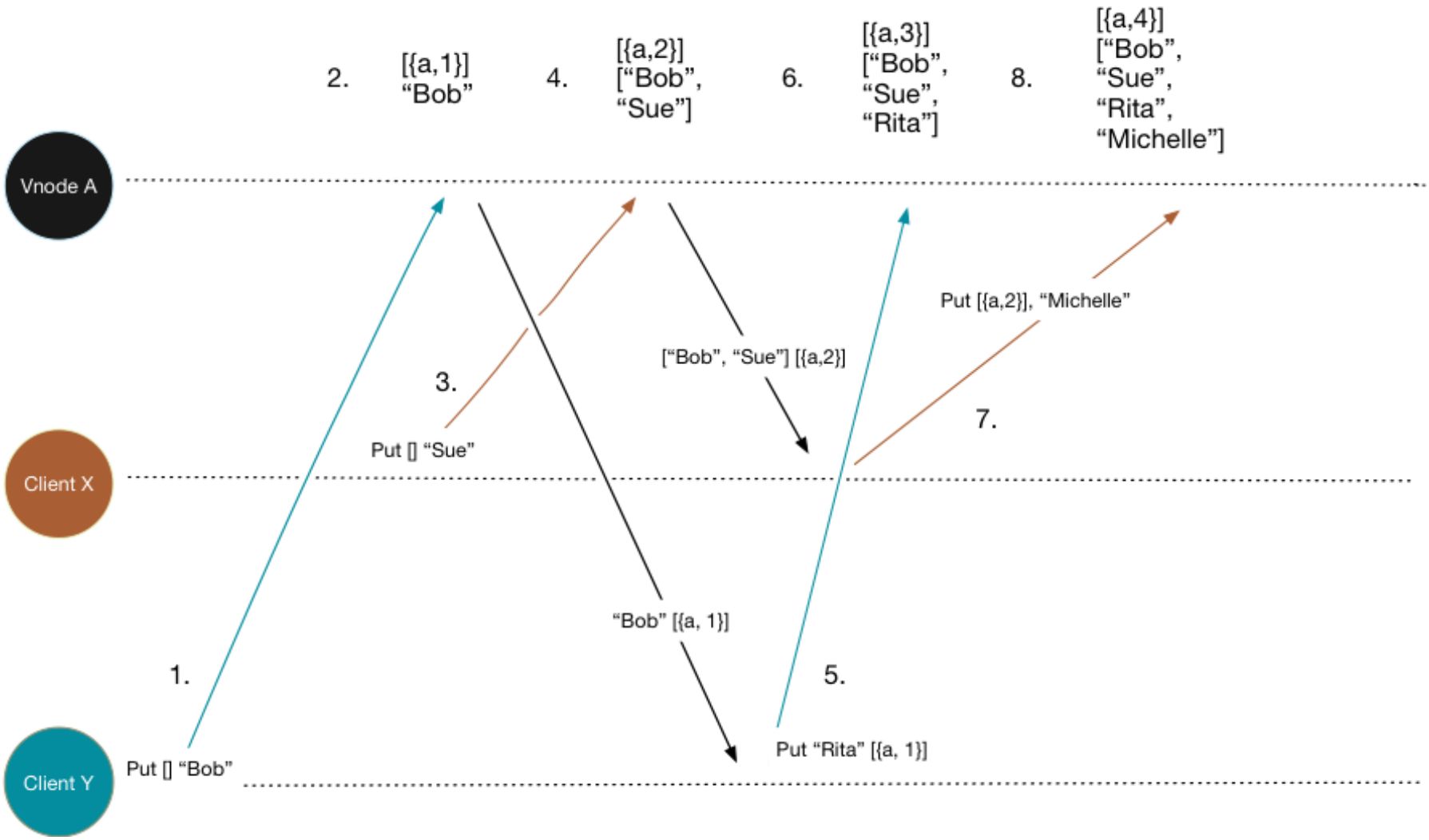
- Keine Siblings bei Verwendung der aktuellen Vektoruhr

```
curl -X PUT http://localhost:8098/riak/lectures/bdprak \
  -H "X-Riak-Vclock: a85hYGBgzGDKBVI8tgF" \
  -H "Content-Type: application/json" \
  -d '{"title" : "Big Data Praktikum",
       "lecturers" : ["Rahm", "Zschache", "Franke"]}'
```

- Eine weitere Aktualisierungen mit gleicher (nun echt kleineren) Vektoruhr würde zu *Siblings* führen und eine manuelle Auflösung erfordern

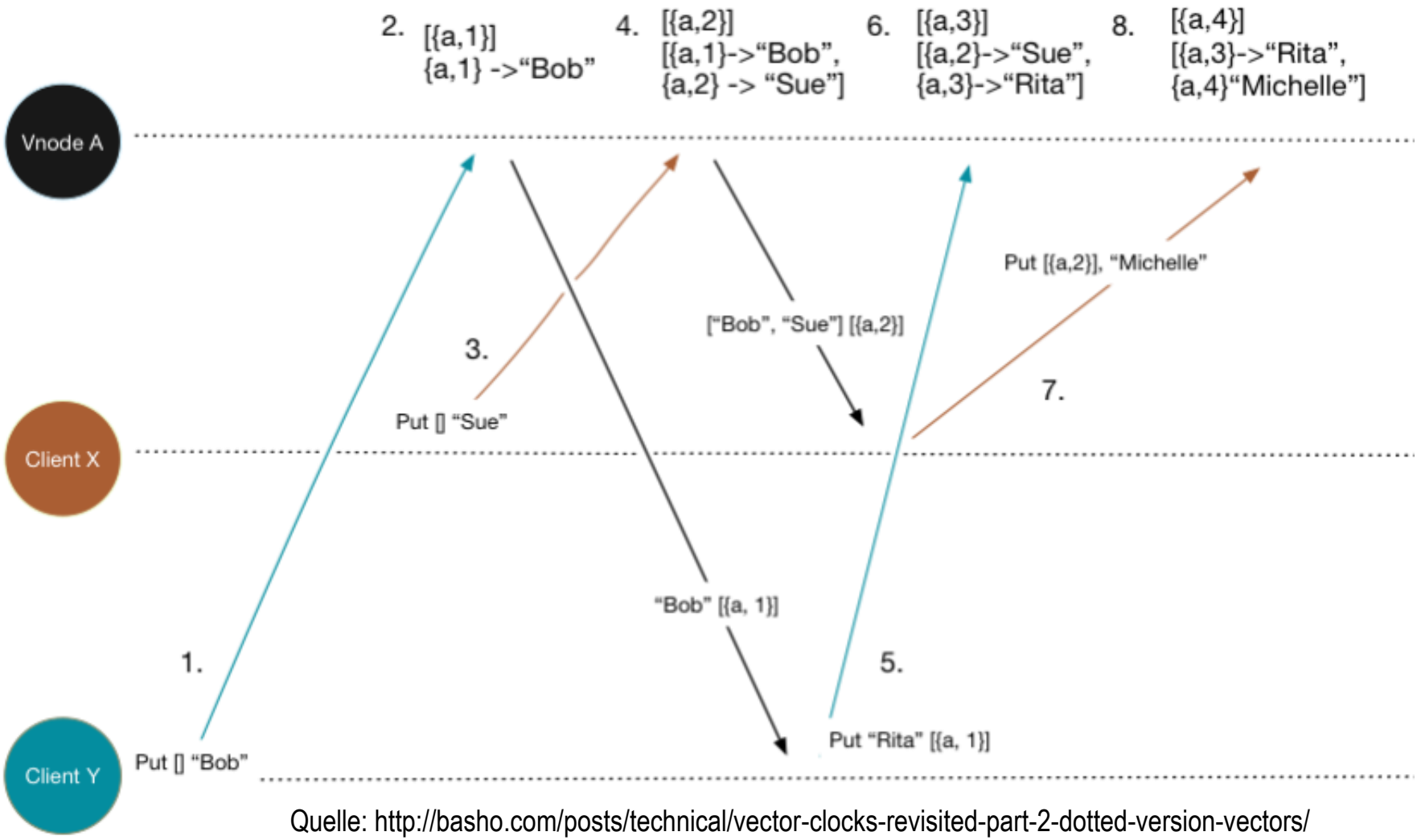
```
curl -X PUT http://localhost:8098/riak/lectures/bdprak \
  -H "X-Riak-Vclock: a85hYGBgzGDKBVI8tgF" \
  -H "Content-Type: application/json" \
  -d '{"title" : "Big Data Praktikum",
       "lecturers" : ["Rahm", "Zschache", "Christen"]}'
```

Problem: Sibling Explosion



Quelle: http://basho.com/posts/technical/vector-clocks-revisited-part-2-dotted-version-vectors/

Lösung : Dotted Version Vectors



Quelle: http://basho.com/posts/technical/vector-clocks-revisited-part-2-dotted-version-vectors/

- Parameter für `"props":{"dvv_enabled": true}`

Inhaltsverzeichnis: Key-Value Stores

Zusammenfassung: Key-Value Stores

- Einfache, sehr flexible Form der Datenspeicherung
  - Einfache Anfragen: put(key, value), get(key), remove(key)
  - Hohe Lese-/Schreibraten für große, unstrukturierte Datensätze
  - Fortgeschrittene Funktionen in speziellen Datenbanken

	Redis	Riak
Stärken	- Geschwindigkeit - Linearisierbar (Single-Threaded) - Komplexe Datentypen - Master-Slave Replikation für schnelles Lesen	- Hohe Verfügbarkeit - Keinen Single-Point-of-Failure - Flexible Skalierbarkeit - HTTP Anfragen - Dynamische Anpassung von CAP
Schwächen	- Begrenzte Datensicherheit - Beschränkung der Datengröße durch Hauptspeicher eines einzelnen Rechners - Gefahr der Wiederherstellung alter Werte	- Einfache Datenstruktur - Keine Transaktionen - Implementierung in Erlang - Ungewisse Zukunft

Literatur

- [Redis]: http://redis.io/documentation
- [Riak]: http://docs.basho.com/riak/kv/latest

- [Dynamo]: DeCandia et al.: „Dynamo: Amazon's Highly Available Key-value Store“, https://docs.riak.com/riak/kv/2.2.3/learn/dynamo/index.html
- [Rich17]: Richardson: „Riak, the Dynamo paper and life beyond Basho.”, https://opencredo.com/riak-the-dynamo-paper-and-life-beyond-basho/
- [RiakDocker] https://hub.docker.com/r/basho/riak-kv/