Datenbanken und Informationssysteme

NoSQL

Burkhardt Renz

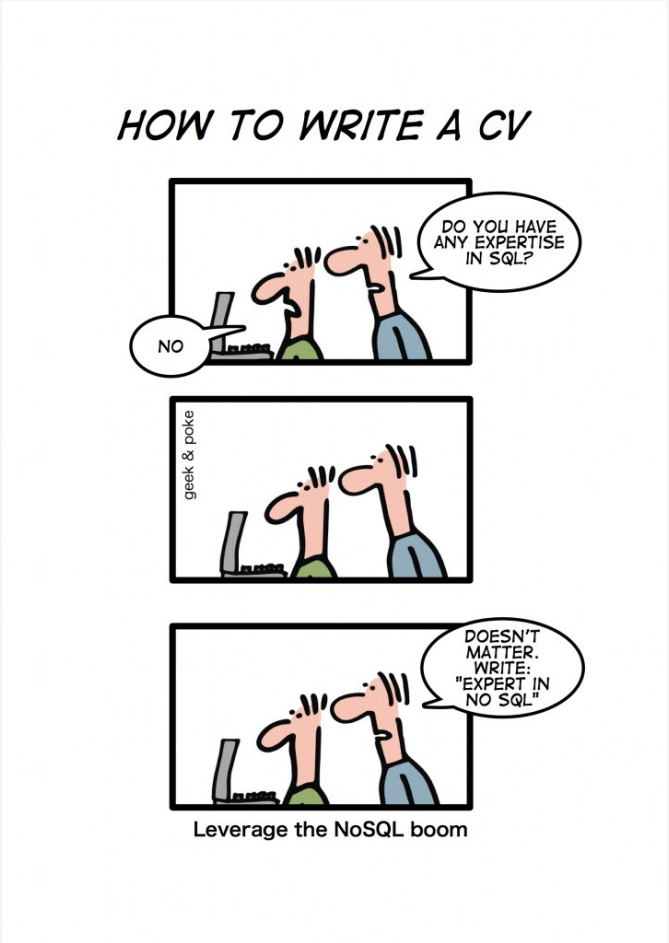
Fachbereich MNI
TH Mittelhessen

Sommersemester 2020

NoSQL

- Terminus: *Not SQL*, *Not Only SQL*
- Ironie: Strozzis NoSQL ist relational `http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page`
- Hashtag für ein Meeting am 11.6.2009 in San Francisco, initiiert von Johan Oskarsson

NoSQL



Quelle: `http://geekandpoke.typepad.com/geekandpoke/2011/01/nosql.html`

Charakteristik von NoSQL

- Datenmodell nicht relational
- Geeignet für Cluster (Rechnerverbünde), skalierbar
- i.A. Open Source
- kein „starres“ Schema , *schemaless*
- eventual consistency (BASE *basically available, soft state, eventually consistent*)

$\Rightarrow$ diskutable Merkmale $\rightarrow$ Teil 2 dieser Folien

Gründe

- Große Datenmengen
- Heterogene Daten
- Cluster von Linux-Boxen
- Vorrang von Verfügbarkeit gegenüber Konsistenz

Übersicht

Datenmodell

Erster Begriff von Datenmodell

> *A data model is an abstract, self-contained, logical definition of the objects, operators, and so forth, that together constitute the abstract machine with which the users interact.*
>
> *The objects allow us to model the structure of data. The operators allow us to model its behavior.*
>
> *– C.J. Date*

Beispiel: das relationale Datenmodell
konkret: Wir organisieren die interessierenden Informationen als *Werte* in *Tupeln* (=Datensätzen), die *Relationen* (=Tabellen) bilden und geben an, wie diese Relationen *zusammenhängen*.

Datenbankschema

Zweiter Begriff von Datenmodell (= Datenbankschema)

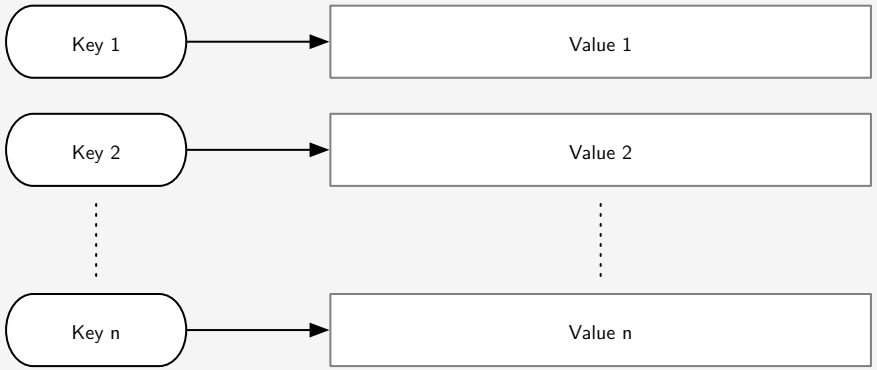
> *A data model is a model of the persistent data of some particular enterprise.*
>
> *– C.J. Date*

Beispiel:
Metadaten sind die Informationen *über* den Aufbau der Daten – sie werden im relationalen Modell selbst wieder in Relationen gespeichert – Systemkatalog

Datenmodelle in NoSQL – Übersicht

- Datenmodelle für Aggregate
  - *Key-Value Stores*
  - *Document Stores*
  - *Column-Family Stores*
- Datenmodell für *Graphen*
  - Graph-Datenbanken

Datenmodell von Key-Value Stores



- Ein „Datensatz“ besteht aus Schlüssel und Wert
- Wert hat aus Sicht der Datenbank keine Struktur
- Anwendungsseitige Interpretation der Werte
- Keine Beziehungen zwischen Daten — wenn, dann anwendungsseitig
- Erweiterungen des Datenmodells je nach Produkt

Zugriff auf Key-Value Stores

- `put` schreibt Schlüssel/Wert-Paare
- `get` liest Schlüssel/Wert-Paar zu einem Schlüssel
- `delete` löscht Schlüssel/Wert-Paar zu einem Schlüssel

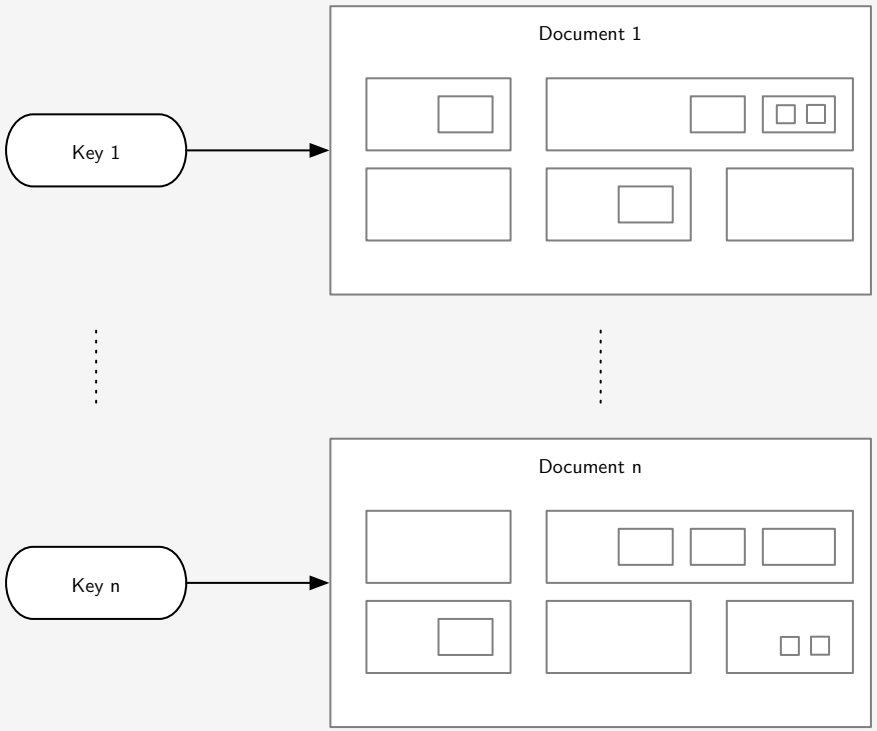
Beispiel Riak KV

- Basiert auf dem Design von *Amazon Dynamo*
- Implementiert in Erlang
- Datensatz/Objekt: *Bucket*/*Key* + *Value*
- *Bucket Type*: für gemeinsame Parameter von Buckets
- *Riak Data Types*: Datentypen von Werten z.B. Maps
- *Riak Search*: Volltextsuche mittels Solr/Lucene

Beispiel Riak KV

- put, get, delete, post
- Liste aller Buckets, aller Schlüssel in einem Bucket
- Suche über Secondary Index (2i)
- MapReduce/Link Walking
- Riak Search/Yokozuna über Tags oder Volltext (via Lucene)

Datenmodell von Document Stores



Datenmodell von Document Stores, 2

- Ein „Datensatz“ besteht aus Schlüssel und Dokument
- Dokument: JSON bzw. JavaScript-Objekt mit eindeutiger `id`, andere Implementierungen: XML
- Eingebettete Daten: JSON in JSON
- Referenzen: Werte sind Ids anderer Dokumente

Zugriff auf Document Store

- Insert: `insert(doc)`
- Query: `find(id)`, Suche mit Vergleichsoperatoren, logischen Operatoren im Inhalt der Dokumente per dot-Notation
- Update: `update(id, set(...)`
- Delete: `delete(id)`

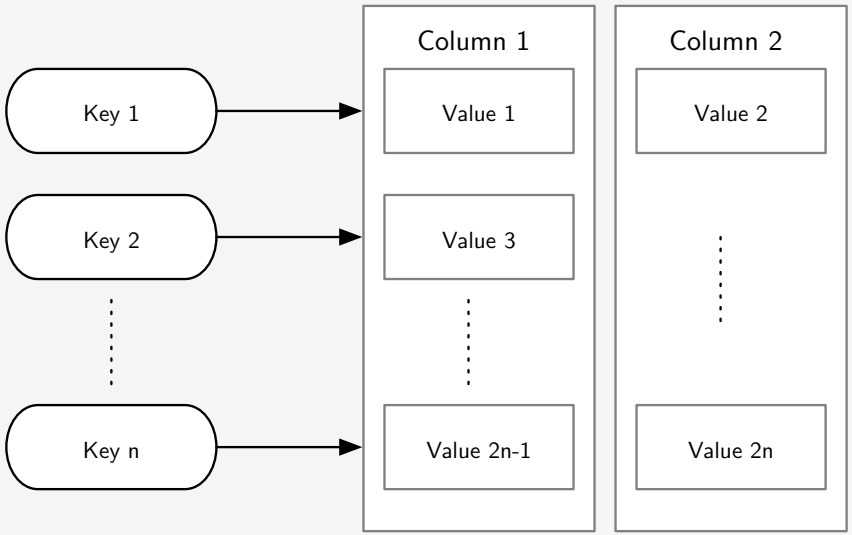
Beispiel Mongo DB

- Implementiert in C++
- Internes Format der Dokumente: BSON (binary JSON)
- Schema-Validierung ist möglich
- Sekundäre Indexe möglich

Beispiel Mongo DB

- Ausgefeilte Abfragesprache
- Aktionen sind atomar pro Dokument, Isolations-Garantien ähnlich zu Isolationsleveln in SQL
- Seit Mongo DB 4.0 auch Transaktionen, die mehrere Dokumente betreffen, also verteilte Transaktionen
- Volltextsuche: Atlas Search basierend auf Apache Lucene

Datenmodell von Column-Family Stores



Datenmodell von Column-Family Stores, 2

- Ein „Datensatz“ hat als Schlüssel die Kombination aus (*row key*, *column name*, *timestamp*) und als Wert ein *Byte array*
- Die *column name*s sind gruppiert in *column families* – sie bilden die Basis für die Zugriffssteuerung
- Werte sind multi-versioniert, dazu dienen die *timestamps*
- Die Daten werden nicht in einer Haufendatei (wie bei relationalen Systemen typisch) gespeichert, sondern pro *column family*

Zugriff auf Column-Family Store

- CRUD-Operationen, je nach Implementierung sehr unterschiedlich ausgeprägt
- MapReduce

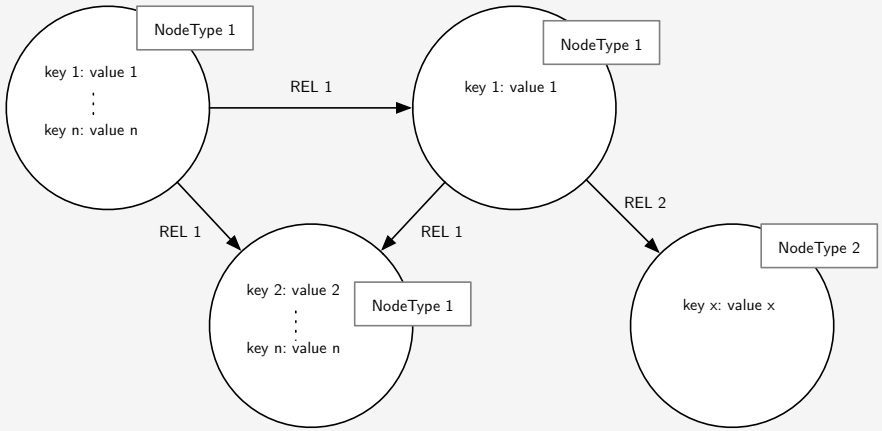
Beispiel Apache Cassandra

- Ursprünglich initiiert von Facebook basierend auf Ideen von Amazon Dynamo und Google BigTable
- Implementiert in Java
- Persistenz der Daten in *SSTables*
- *SSTable* = „Sorted Strings Table“ = Speicherstruktur für *Key-Value Store*, hier pro *Column Family*
- *SSTable* besteht im Prinzip aus einem Index, der die Daten einer *Column Family* über den *Row Key* indiziert

Beispiel Apache Cassandra

- Cassandra Query Language, SQL-ähnlich
- Redundanz statt Joins
- Keine referenzielle Integrität
- Modellierung: *Query-first design*

Datenmodell von Graph-Datenbanken



Datenmodell von Graph-Datenbanken, 2

- *Entitäten* werden repräsentiert durch Knoten
- *Beziehungen* von Entitäten werden dargestellt durch Links (Gerichtete Kanten im Graph)
- Entitäten (Knoten) und Beziehungen (Kanten) haben *Eigenschaften*
- Entitäten und Beziehungen können in *Typen* kategorisiert werden

Literatur

Ian Robinson, Jim Webber, Emil Eifrem: *Graph Databases*, O'Reilly 2015.

Zugriff auf Graph-Datenbank

- Knoten erzeugen: `graphDB.createNode()`
- Kanten erzeugen: `node1.createRelationshipTo(node2, ...)`
- Eigenschaften hinzufügen: `node.setProperty()` , `rel.setProperty()`
- Abfragen auf Graphen:
  - Startknoten suchen per Id oder Eigenschaft
  - Traversieren des Graphen entlang der Beziehungen
  - Verwenden der Knoten die man dabei findet

Beispiel neo4j

- Implementiert in Java
- *Native* Graph-Datenbank: Direkte Verknüpfung von Knoten mit Kanten
- *Native* Graph-Navigation: *index-free adjacency*
- Optional Indexe und Integritätsbedingungen (also ein Schema)
- Transaktionen (ACID)

Beispiel neo4j

- Sprache: Cypher (ganz entfernt von SQL inspiriert)
- `CREATE`: Knoten und Kanten erzeugen
- `MATCH`: Knoten/deren Eigenschaften, die Kriterien entsprechen
- `MERGE`: Update von Knoten
- Aggregat-Funktionen, Sortierung, Paginierung

Übersicht

Aus dem Internet

Properites of NoSQL databases

- There are no complex relationships, such as the ones between tables in an RDBMS.
- They have higher scalability. They use distributed computing.
- They support flexible schema. They're able to process both unstructured and semi-structured data.
- In NoSQL databases, the principles of ACID (atomicity, consistency, isolation, and durability) are reduced.

Quelle: Satyakam Mishra: *Properties of RDBMSs and NoSQL databases*, `https://support.rackspace.com/how-to/properties-of-rdbmss-and-nosql-databases/`, 2019

Nicht-relationales Datenmodell

Charakteristik des relationalen Datenmodells

- *Ein* Konzept: Werte in Tupeln in Relationen
- Beziehungen der Daten ergeben sich *nur* durch die Gleichheit von Werten, d.h. keine „eingebaute“ Navigation
- Keine Redundanz von Daten
- Deklarative Abfragesprache unabhängig von internen Zugriffsmechanismen

Charakteristik des aggregat-orientierten Datenmodells

- *Aggregate* sind Kollektionen von Daten, die in einer Anwendung als Einheit verwendet werden
- Der Zugriff erfolgt in der Regel über eine „eingebaute“ Strategie: den *Schlüssel* des Aggregats

Diskussion des aggregat-orientierten Datenmodells, 1

Beispiel Auftragsdaten

Aggregation:
Kunde (mit eventuell mehreren Adressdaten)
Aufträge mit Auftragspositionen und Artikeln
Daten der Bezahlung
zugreifbar über einen Key (Kundennummer)

Vor- und Nachteile

- sehr praktisch, wenn man alle Daten eines Kunden benötigt
- sehr unhandlich, wenn man sich für den Umsatz eines bestimmten Artikels interessiert
- zwingt zu Redundanz oder Verwendung von Referenzen auf die Keys anderer Dokumente

Diskussion des aggregat-orientierten Datenmodells, 2

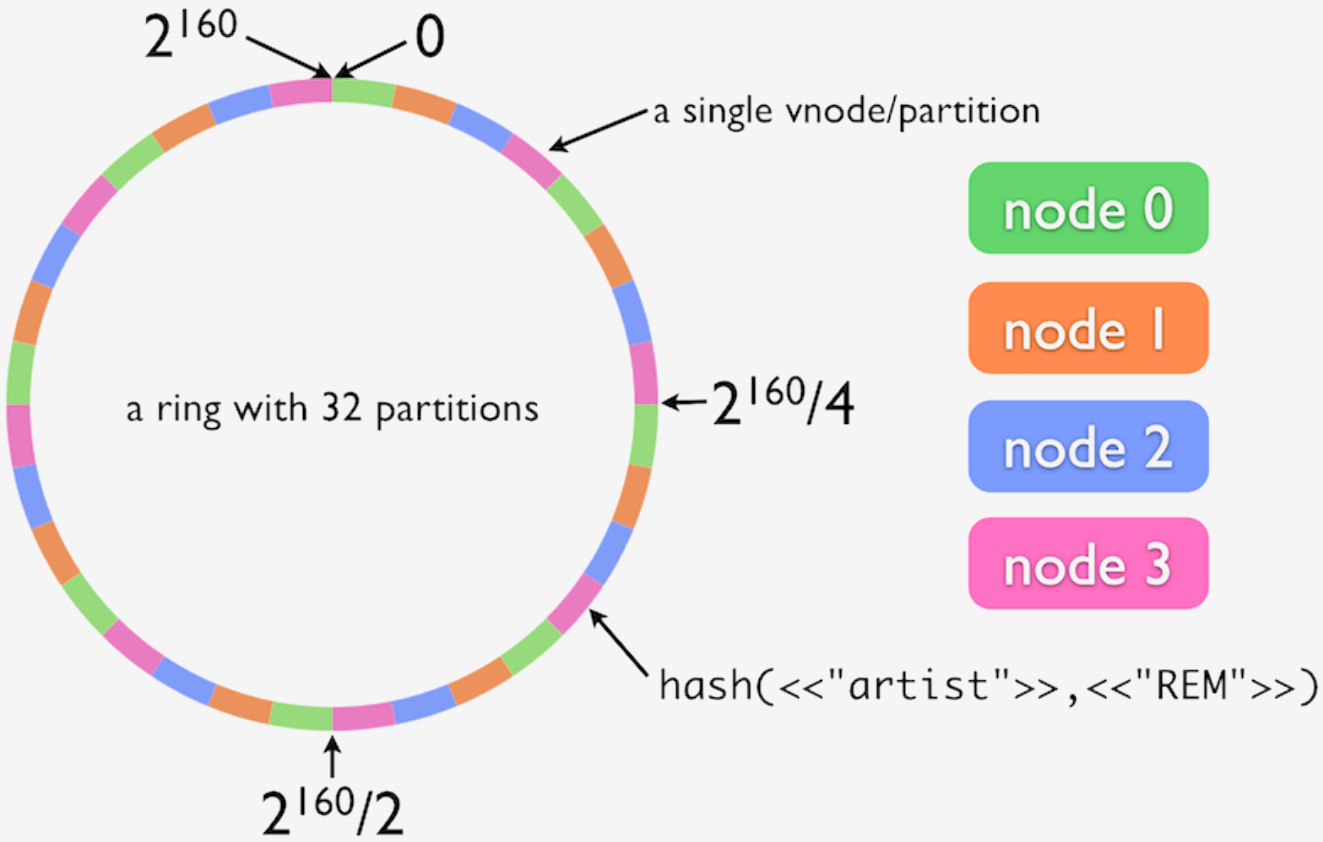
Fazit zum Datenmodell

- Das aggregat-orientierte Datenmodell ist nicht generisch, sondern muss in der Verwendung auf eine bestimmte Anwendung hin konzipiert werden.
- Faktoren des „Schneidens“ der Aggregate:
  - Objektmodell der Anwendung
  - Absehbare Zugriffspfade der Anwendung
  - Folgen für die Skalierbarkeit und die Konsistenz
- erlaubt automatisches Zuordnen zu Knoten eines verteilten Systems $\Rightarrow$ horizontale Skalierbarkeit (*scale out*)

Consistent Hashing

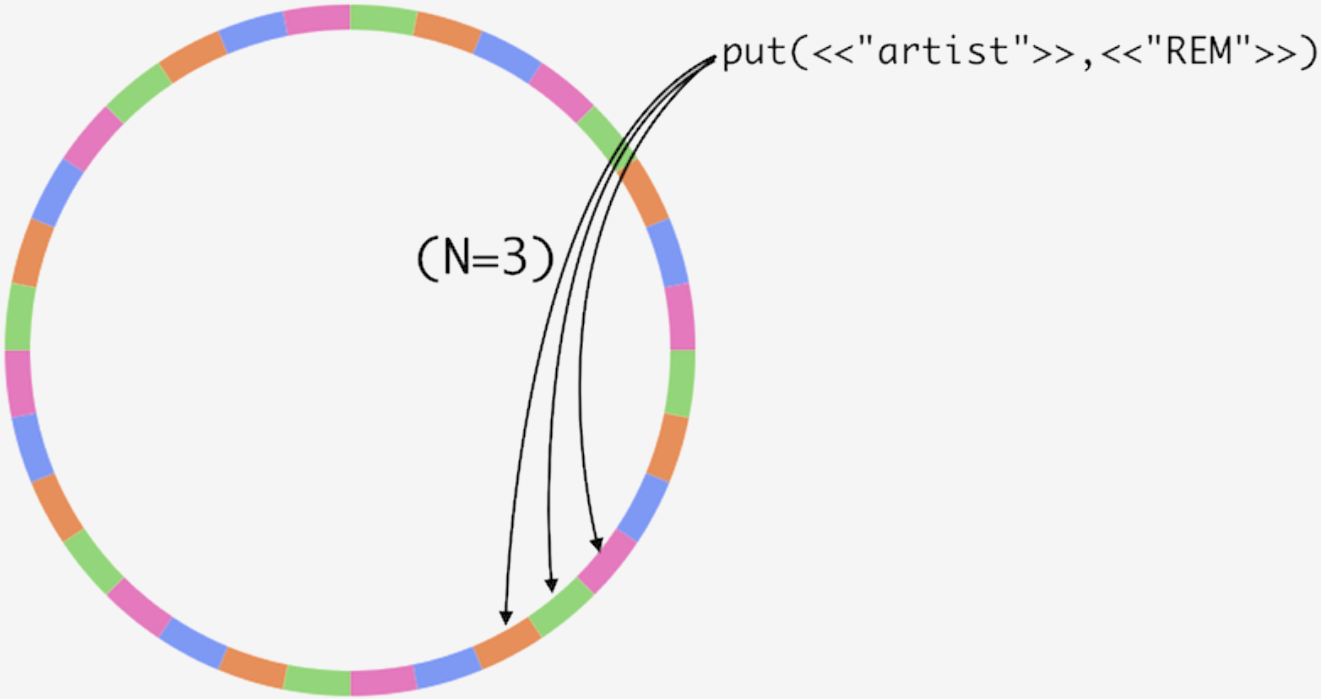
- Verwenden einer Hash-Funktioni auf dem Key, um einem Aggregat einen Speicherort in einem verteilten System zuzuweisen
- Was passiert, wenn sich die Zahl der Speicherorte (Knoten) im verteilen System ändert?
- Hashing möglichst so, dass nicht zu viele Aggregate kopiert werden müssen
- Lösung: Consistent Hashing, wie etwa bei RIAK KV

Der Riak-Ring



Quelle: Riak KV Documentation *Learning* > *Clusters*

Replikation in Riak KV



Quelle: Riak KV Documentation *Learning* > *Clusters*

Diskussion Skalierbarkeit/Verteilte Datenbank

- Verteilte Datenbanken in der relationalen Welt erlauben *keine* automatische Verteilung der Daten auf Knoten
- Aggregat-orientierte Datenmodelle kombiniert mit *Consistent Hashing* erlauben automatische Verteilung und sogar Umverteilung bei Ausfall oder Hinzukommen von Knoten
- Dadurch ist NoSQL effizient skalierbar und kann dadurch mit kostengünstiger Hardware hochverfügbar betrieben werden
- Der Preis $\Rightarrow$ Verzicht auf *Datenunabhängigkeit*

Ein (kritisches) Beispiel, 1

Eine Person mit ihrer Adresse

```
// patron document
{
   _id: "joe",
   name: "Joe Bookreader"
}

// address document
{
   patron_id: "joe", // reference to patron document
   street: "123 Fake Street",
   city: "Faketon",
   state: "MA",
   zip: "12345"
}
```

Aus der Dokumentation von Mongo DB

Ein (kritisches) Beispiel, 2

Noch eine Person mit ihrer Adresse

```
// patron document
{
   _id: "eve",
   name: "Eve Photograph",
   address: {
              street: "123 Fake Street",
              city: "Faketon",
              state: "MA",
              zip: "12345"
            }
}
```

Aus der Dokumentation von Mongo DB (leicht abgewandelt)

Diskussion des Beispiels

Angenommen nur eine der beiden Formen kommt in einer Mongo DB vor $\Rightarrow$

- Damit die Anwendung die Daten verarbeiten kann, muss sie den Aufbau, das Schema der Daten kennen, d.h. die Datenbank hat ein *implizites Schema*.
- Flexibiltät besteht nur insofern, als es auch Personenobjekte geben kann, die zusätzliche, für unsere Anwendung uninteressante Daten enthalten

Angenommen beide Arten eine Person als JSON-Objekt darzustellen kommt in ein und derselben Mongo DB vor $\Rightarrow$

- Die Anwendung muss *beide* Schemata kennen.
- Anwendungsseitig muss eine *Schemavalidierung* durchgeführt werden, um erlaubte, gültige Daten von invaliden Daten unterscheiden zu können.

Diskussion Schemafreiheit

- Die Aussage *schemaless data* als Eigenschaft von NoSQL ist unzutreffend:

  „A common statement about NoSQL databases is that since they have no schema, there is no difficulty in changing the structure of data during the life of an application. We disagree—a schemaless database still has an implicit schema that needs change discipline when you implement it. . . “ [Sadalage, Fowler: *NoSQL Distilled*, Preface]

- Anwendungsseitig muss ein implizites Schema der Daten explizit verwendet werden
- Andererseits: Gewisse Flexibiltät bei Erweiterung oder Änderung von Strukturen
- Und: Gut geeignet für Aggregate semistrukturierter Daten (baumartig, viele optionale Komponenten)

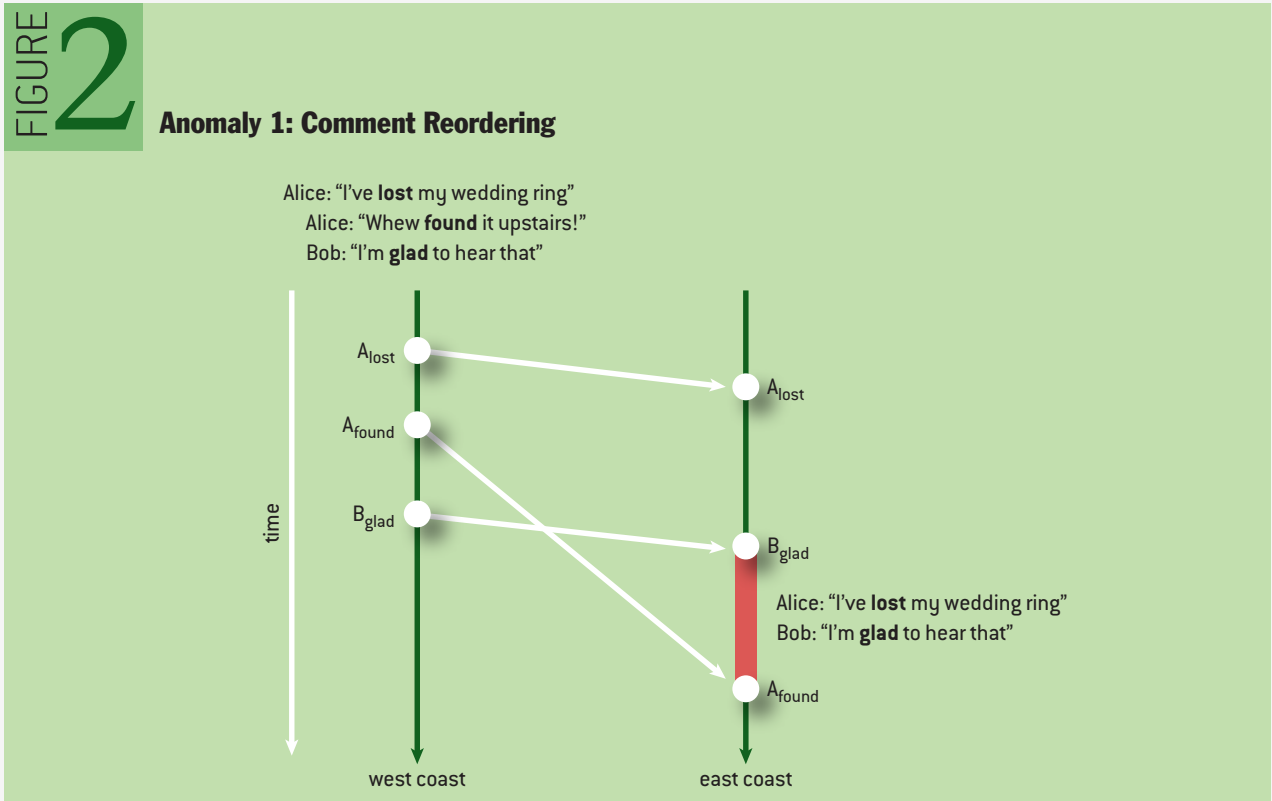
ACID und das CAP-Theorem

- ACID benennt die Eigenschaften von Transaktionen, darunter C = *Consistency*
- Das sogenannte CAP-Theorem besagt, dass in einem verteilten System nicht alle drei Eigenschaften erfüllt sein können: *Consistency*, *Availability* und *Partition Tolerance*
- *Consistency* in ACID bedeutet: Nach einer Transaktion erfüllt der Datenbankzustand alle Integritätsbedingungen, die im Datenbankschema definiert sind.
- *Consistency* in Brewers CAP-Theorem bedeutet: Jedes Lesen von Daten liefert immer die aktuellsten Daten (oder eine Fehler) — auch bei replizierten Daten.
- Die beiden Begriffe sind *nicht* identisch!

BASE

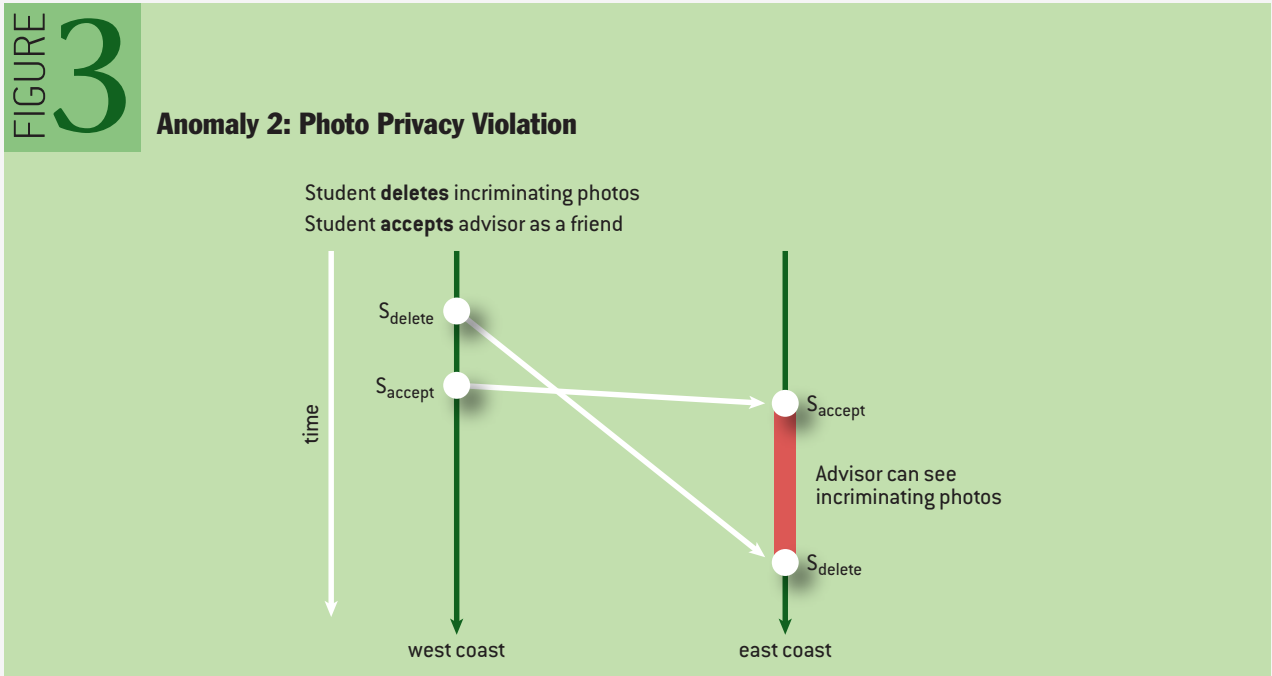
- Aggregat-orientierte NoSQL-Datenbank sind verteilte Systeme. CAP bedeutet für sie: Bevorzugung von *Availabiltity* gegenüber *Consistency*
- BASE = *Basically Available, Soft State, Eventual consistency*
- Schwache Konsistenz aka *Eventual consistency*: Alle Replika haben schließlich denselben Wert, vorausgesetzt er wird nicht mehr verändert
- Gerne getätigte Gegenüberstellung ACID — BASE nicht sinnvoll

Phänomene bei schwacher Konsistenz, 1



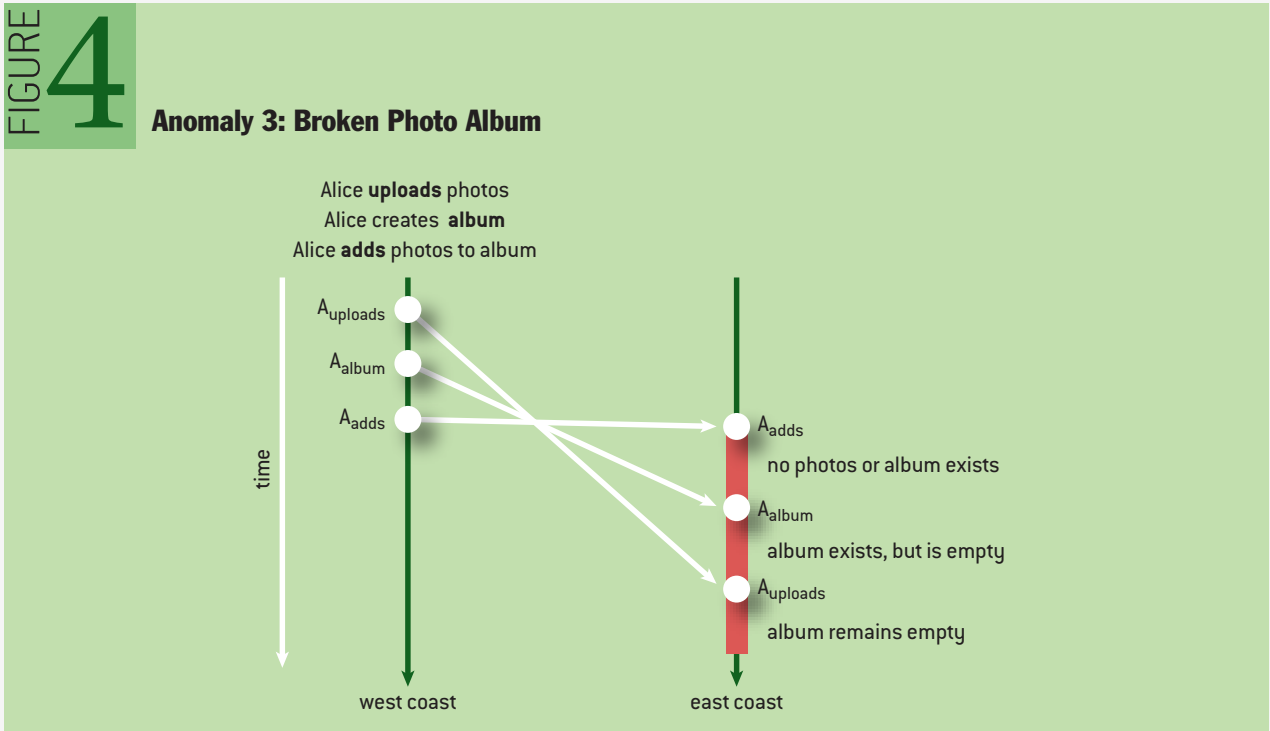
Quelle: W. Floyd et al: *Don't Settle for Eventual Consistency*, ACM Queue, Vol 12(3), April 2014

Phänomene bei schwacher Konsistenz, 2



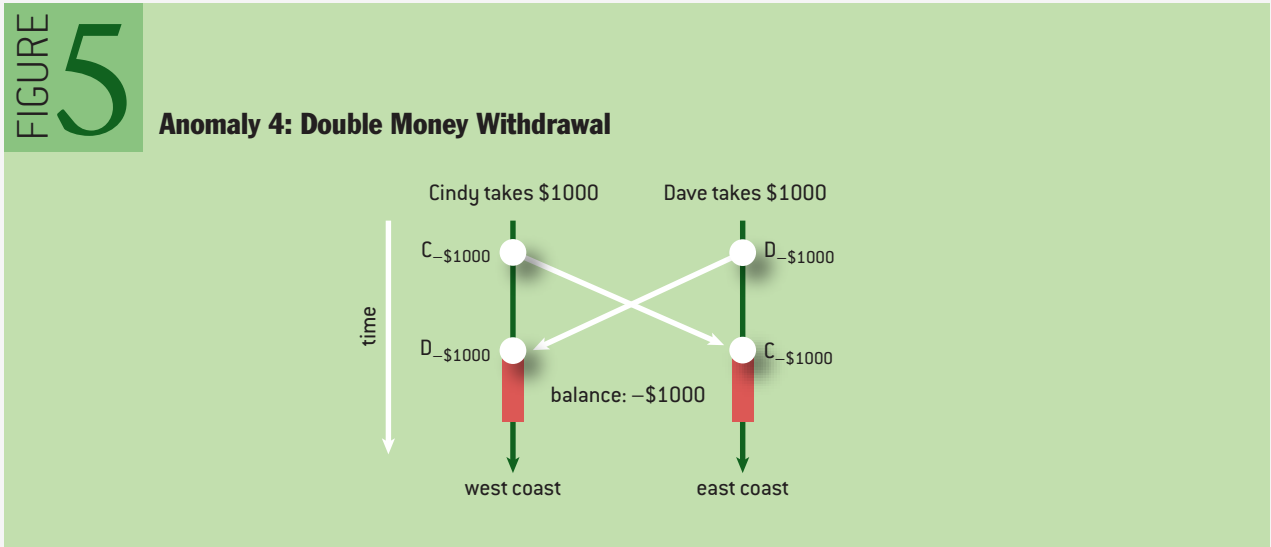
Quelle: W. Floyd et al: *Don't Settle for Eventual Consistency*, ACM Queue, Vol 12(3), April 2014

Phänomene bei schwacher Konsistenz, 3



Quelle: W. Floyd et al: *Don't Settle for Eventual Consistency*, ACM Queue, Vol 12(3), April 2014

Phänomene bei schwacher Konsistenz, 4



Quelle: W. Floyd et al: *Don't Settle for Eventual Consistency*, ACM Queue, Vol 12(3), April 2014

Diskussion Schwache Konsistenz

- Mögliche Phänomene bei schwacher Konsistenz präzise analysieren — sind im laufenden Betrieb nur schwer nachzuvollziehen
- Genaue Analyse, welches Level an Konsistenz benötigt wird — Phänomene 1 - 3 sind durch *Causal consistency* ausgeschlossen
- Phänomen 4 — benötigt verteilte Transaktion mit 2PC – oder:
- Google Spanner: „Spanner is Google's scalable, multi-version, globally- distributed, and synchronously-replicated database“ [Corbett et al: *Spanner: Google's Globally-Distributed Database*, Proceedings OSDI, 2012]

Übersicht

Fazit, 1

> *In a system that cannot count on distributed transactions, the management of uncertainty must be implemented in the business logic.*
>
> *Pat Helland*

Lektüre

Pat Helland: *Life Beyond Distributed Transactions*
Comm. of the ACM, Febr 2017

Fazit, 2

NOSQL was born out of the necessity to build highly available systems that operate at scale at a comparatively low cost.

Transactions are essential for mission-critical applications

...

... embracing the relational model and SQL natively, in a tightly coupled fashion has been the latest big leap...

Lektüre

David F Bacon et al: *Spanner: Becoming a SQL System*
SIGMOD'17, Mai 2017