FOM Hochschule

Datenbankmanagement

Prof. Dr. Gregor Hülsken

Theorie 5: Referenzen, Lookup etc. in der Praxis

Lernziele

- **Einsatz von Lookup in der Praxis und die verschiedenen Verfahren dazu**
- **Weitere Speicherverfahren**

Referenzen, Lookup etc. in der Praxis

Inhalte

✓ OTLT (On True Lookup Table)

Relationen

ENUM / Check Constrains

EAV (Entity Attribute Value)

Verteilte DB-Systeme, NoSQL

Beispiel:

Datensatz Herzschrittmacher-Implantation

09/1 (Spezifikation 15.0 SR 3)

BASIS

Genau ein Bogen muss ausgefüllt werden

1-7 Basisdokumentation

1 Institutionskennzeichen
http://www.arge-ik.de
☐☐☐☐☐☐☐☐☐

2 Betriebsstätten-Nummer
☐☐

3 Fachabteilung
§ 301-Vereinbarung
§ 301-Vereinbarung: http://www.dkgev.de
☐☐☐☐
Schlüssel 1

4 Identifikationsnummer des Patienten
☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐

5 Geburtsdatum
TT.MM.JJJJ

9 führendes Symptom
☐

0 = keines (asymptomatisch)
1 = Präsynkope/Schwindel
2 = Synkope einmalig
3 = Synkope rezidivierend
4 = Synkopenbedingte Verletzung
5 = Herzinsuffizienz NYHA II
6 = Herzinsuffizienz NYHA III oder IV
9 = sonstiges

10 führende Indikation zur Schrittmacherimplantation
☐☐

1 = AV-Block I
2 = AV-Block II Wenckebach
3 = AV-Block II Mobitz
4 = AV-Block III
5 = faszikuläre Leitungsstörung
6 = Sinusknotensyndrom (SSS) inklusive BTS (bei paroxysmalem/persistierendem Vorhofflimmern)
7 = Bradykardie bei permanentem Vorhofflimmern
8 = Karotis-Sinus-Syndrom (CSS)

13-23 Präoperative Diagnostik

13-17 Indikationsbegründende EKG-Befunde

13 Vorhofrhythmus
☐

1 = normofrequenter Sinusrhythmus
2 = Sinusbradykardie/SA-Blockierungen
3 = paroxysmales/ persistierendes Vorhofflimmern/-flattern
4 = permanentes Vorhofflimmern
5 = Wechsel zwischen Sinusbradykardie und Vorhofflimmern (BTS)
9 = sonstige

14 AV-Block
☐

0 = keiner
6 = nicht beurteilbar wegen Vorhofflimmerns
1 = AV-Block I. Grades, Überleitung <= 300 ms
2 = AV-Block I. Grades, Überleitung > 300 ms
3 = AV-Block II. Grades, Typ Wenckebach
4 = AV-Block II. Grades, Typ Mobitz
5 = AV-Block III. Grades

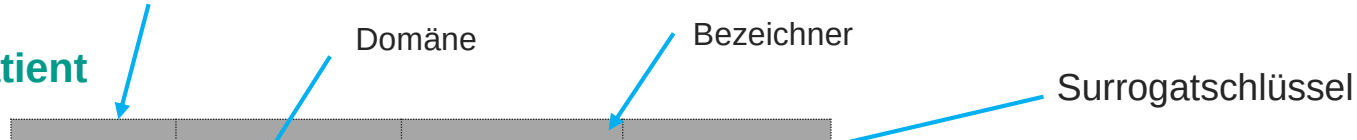
Referenzen, Lookup etc. in der Praxis

OTLT (One True Lookup Table)

- Eine Tabelle für statische Referenzen mit Key als Identifier
- OO Design in Relationales DB Design gepresst

Name	Vorname	Geschlecht	Geschlecht_ID	Blutgruppe
Walkes	Otto	Männlich	1	1
Hagen	Nina	Weiblich	2	2

R Patient



LookupTyp	LookupCode	LookupDesc	LookupID
Patient	Geschlecht	Männlich	1
Patient	Geschlecht	Weiblich	2
Patient	Geschlecht	Unbekannt	3
Patient	Blutgruppe	A	1
Patient	Blutgruppe	B	2

Referenzen, Lookup etc. in der Praxis

Vorteile:

- Aus Sicht des Programmierers einfache Handhabung,
- Schnelle Möglichkeit des Hinzufügens neuer Werte

Nachteile:

- Foreign keys sind nicht anwendbar, Datenintegrität kann nicht gewährleistet werden
- Datentypen können nicht benutzt werden, da alles in dem Datentyp String gespeichert werden muss => domain integrity kann nicht gewährleistet werden
- Daraus folgt, dass die String Column groß sein muss => große Datenmenge
- SQL Statements werden komplexer und Typ konvertierungen bei Joins müssen vorgenommen werden
- Sollte der Enduser die Möglichkeit zum hinzufügen bekommen

- → Rechtschreibfehler in den Attributen können die Funktionalität des Programms stören

Referenzen, Lookup etc. in der Praxis

Inhalte

- ✓ OTLT (On True Lookup Table)
- ✓ Relationen
- ENUM / Check Constrains
- EAV (Entity Attribute Value)
- NoSQL

Referenzen, Lookup etc. in der Praxis

Relationen: Eine Tabelle für jeden Referenzwert

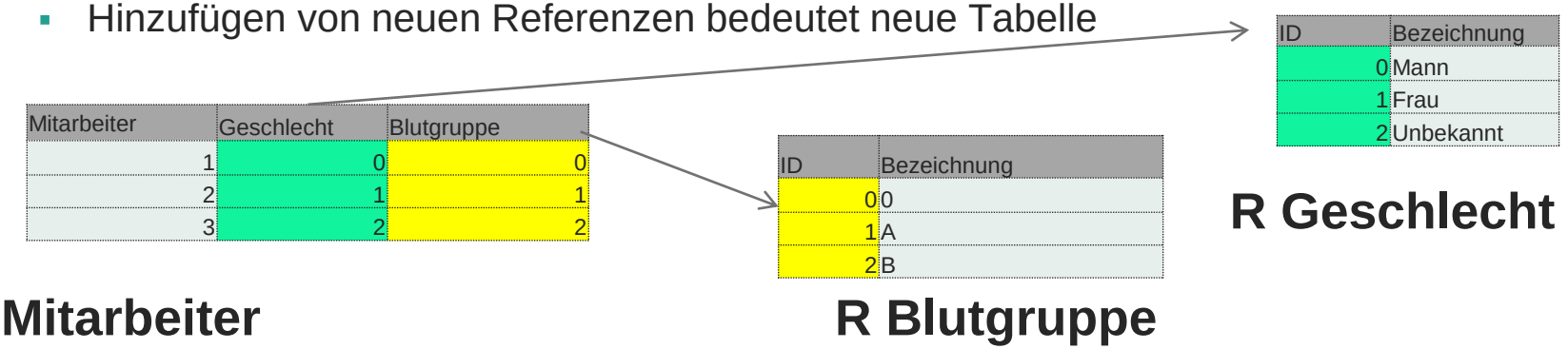
> ➢ Eine Tabelle für jeden Referenzwert

➢ **Vorteile:**

- Daten- / Domainintegrität können gewährleistet werden
- Normalform kann eingehalten werden
- Foreignkeys

➢ **Nachteile:**

- Bei großen Datenbanken müssen viele solcher Tabellen erstellt werden, kann aber durch passende Views vereinfacht werden
- Hinzufügen von neuen Referenzen bedeutet neue Tabelle



Referenzen, Lookup etc. in der Praxis

Inhalte

- ✓ OTLT (On True Lookup Table)
- ✓ Relationen
- ✓ ENUM / Check Constrains

EAV (Entity Attribute Value)

NoSQL

ENUM

- Festdefinierte Metadaten
- Datentyp ENUM ('Männlich', 'Weiblich', 'Unbekannt')

Value	Index
Männlich	1
Weiblich	2
Unbekannt	3

Vorteile

- Schnell da keine Joins etc.

Nachteile

- Änderung nur unter DDL
- Unflexibel
- Auslesen nur auf Metadatenbasis
- Wird nicht von allen DBMS, z.B. MYSQL(65,535 elements), PostgreSQL ab 8.3 unterstützt

Referenzen, Lookup etc. in der Praxis

Check Constraints

- Festdefinierte Metadaten, können aber mehr als reine Enums

z.B.

CHECK ([Geschlecht] IN ('Männlich','Weiblich', 'Unbekannt'))

CHECK ([Gehalt] >= 1500)

- **Vorteile**
  - Schnell da keine Joins etc. Man kann auch Werte ausgrenzen, die eine gewisse Größe überschreiten
- **Nachteile**
  - Änderung nur unter DDL
  - Unflexibel
  - Auslesen nur auf Metadatenbasis
  - Nicht in allen DBMS, z.B. MSSQL unterstützt

Referenzen, Lookup etc. in der Praxis

Inhalte

- ✓ OTLT (On True Lookup Table)
- ✓ Relationen
- ✓ ENUM / Check Constrains
- ✓ EAV (Entity Attribute Value)

NoSQL

Referenzen, Lookup etc. in der Praxis

Entity–Attribute–Value Model (EAV)

- Tabelle mit Entity-Attribute-Value Columns, um flexibel Daten zu speichern

Beispiel

R Mitarbeiter

Mitarbeiter
1
2
3

Auto
A1
A2

R EAV

E	A	V
1	Name	Meier
1	Einkommen	5000
1	Job	Angestellte
2	Name	Müller
2	Einkommen	1250
3	Job	Manager
3	Einkommen	50000
A1	Ps	120
A2	Farbe	Schwarz

R Auto

Referenzen, Lookup etc. in der Praxis

Vorteile

- Wissen über die Struktur der Daten muss nicht vorhanden sein
- Daten können absolut flexibel gespeichert werden

Nachteile

- Zum Großteil wie bei der OTLT
- Selbst kleine Abfragen werden sehr komplex
  - z.B. Wie kriegt man alle Mitarbeiter, die mehr als 10.000€ verdienen?

Referenzen, Lookup etc. in der Praxis

Inhalte

- ✓ OTLT (On True Lookup Table)
- ✓ Relationen
- ✓ ENUM / Check Constrains
- ✓ EAV (Entity Attribute Value)
- ✓ Verteilte DB-Systeme, NoSQL

Verteilte DB-Systeme

Es gibt verschiedene Ansätze zur Datenspeicherung

- **Schlüssel/Wert** (Key/Value)
  Beispiele: Riak, Redis

  Daten werden **ohne Schema** gespeichert. Keine Struktur regelt, was wie gespeichert wird. Ein Schlüssel kann auf ein Objekt oder einen Textwert oder eine Programmierfunktion zeigen.

  Datenbanken können einfach implementiert und Daten einfach hinzugefügt werden. Da die Daten mit Hilfe eines Keys abgelegt und ausgelesen werden, ist allerdings das Auffinden gespeicherter Werte erschwert.

Referenzen, Lookup etc. in der Praxis

- **Spaltenorientiert**
  Beispiele: HBase

  Ähnlich wie bei Key-Values werden die Daten in Spalten innerhalb eines Schlüsselbereiches abgelegt. Der Schlüsselbereich basiert auf einem eindeutigen Namen, einem Timestamp. Durch den Timestamp kann man alte von neuen Daten einem Wert und unterscheiden.

- **Dokumentenorientiert**
  Beispiel: MongoDB, CouchDB

  Daten für ein Objekt werden in Form eines Dokumentes gespeichert. Dokumente werden in Form von Collections zusammengefasst. Die Dokumente innerhalb einer Collection sind über einen eindeutigen Key auffindbar.

- **Graph**
  Beispiel: Neo4J, Polyglot

Geeignet für Daten, die leicht mit Hilfe eines Graphen repräsentiert werden können, beispielsweise Stammbäume, Airline-Routen, Straßenkarten.

Datenhaltung in verteilten Systemen

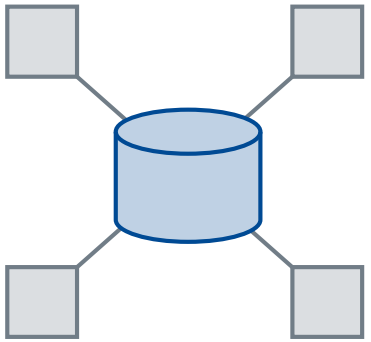
Wie kann ein verteiltes System auf einen gemeinsamen Datenbestand zugreifen?

Problemfelder

- Performance, speziell Latenz
- Verfügbarkeit
- Konsistenz

Einfachste Lösung: Zentrale Datenhaltung

- z.B. zentrale SQL-Datenbank
- Konsistenz: OK
- Verfügbarkeit: ?
- Performance: ?
  - Eingeschränkte Skalierbarkeit!

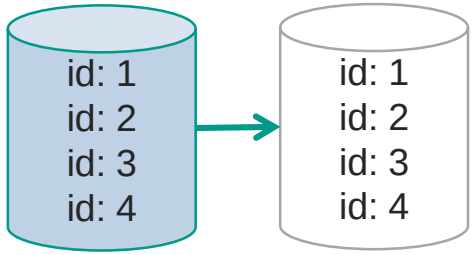


Replikation

Datenbestand liegt als **Kopie** (Replika) auf mehreren Systemen vor.

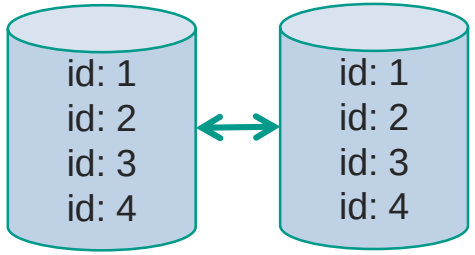
Änderungen werden auf die anderen Systeme **repliziert**.

Master/Slave-Replikation



- Ein Node ist Eigentümer der Daten (Master).
- Nur über den Master erfolgen Schreibzugriffe.
- Änderungen werden auf ein oder mehrere Slaves repliziert.
- Die Slaves können für Leseoperationen genutzt werden.

Peer-to-Peer-Replikation (Multi-Master)



- Alle Nodes sind gleichberechtigt.
- Schreibzugriffe erfolgen über alle Nodes.
- Vermeidung bzw. Management von Konflikten notwendig.

Replikation

Wann gilt Änderung als erfolgreich abgeschlossen?

→ Abhängig vom **Replikations-Modus**

Asynchrone Replikation

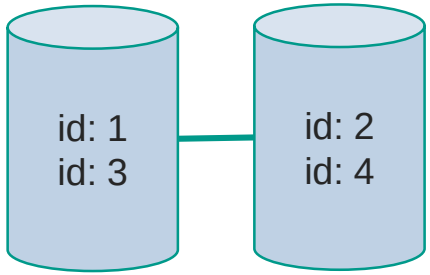
- Änderung sofort abgeschlossen, wird *später* repliziert.
- Performance von Schreiboperationen: gut
- Datenbestand vorübergehend inkonsistent.
  - Verschiedenen Knoten haben unterschiedliche Version der Daten.

Synchrone Replikation

- Änderung erst abgeschlossen, wenn Replikation abgeschlossen.
- Alle Knoten haben gleichen, konsistenten Datenbestand.
- Schreib-Performance?
- Verfügbarkeit?

Sharding

- Zusammengehörige Daten-Aggregate einem Node zugeordnet.
- Schreibzugriffe erfolgen nur über diesen Node.



- Lese- und Schreib-Operationen skalierbar
- Daten-Lokalität → bessere Latenz
- Konsistent: keine Konflikte, da Schreibzugriffe für einen Datensatz über einen definierten Node erfolgen.

Mittels Sharding können umfangreiche Datenmengen verwaltet werden, welche die Kapazitäten eines einzelnen Servers sprengen würden.

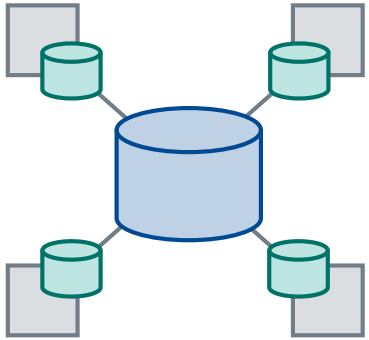
Der größte Nachteil des Shardings ist, dass ein Zugriff über andere Kriterien als das Aufteilungskriterium unverhältnismäßig aufwändig ist.

Abfragen über andere Kriterien oder JOINS müssen i.A. auf alle Server aufgeteilt werden. Das Datenmodell und die Zugriffspfade müssen so designed werden, dass derartige Zugriffe nur selten oder gar nicht vorkommen, sonst werden die Vorteile des Shardings zunichte gemacht.

Ideal für NoSQL

Caching

Clients verwalten eigene Kopie der Daten (oder Teilen davon).



Vorteile

- schneller, lokaler Zugriff
- Entlastung des Netzwerks und der Server

Problemfelder

- Konsistenz: Wie lange sind die gecachten Daten gültig?
- Bei Schreiboperationen im Cache sind Konflikte möglich.

Beispiele

- DNS
- Browser-Cache
- Offline-Webanwendungen
- Microsoft BranchCache, Windows Updates

https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching?hl=de

Map/Reduce

- Programmier-Modell zur **Parallelisierung**
  - Häufig in Software-Frameworks realisiert
  - z.B. Hadoop, MongoDB
- Bearbeitung von komplexen Aufgaben in **2 Phasen**: Map und Reduce

Map

- Gesamtaufgabe wird in einzelne Arbeitspakete aufgeteilt und auf verschiedene Nodes verteilt.
- Nodes bearbeiten jeweilige Teilaufgabe unabhängig voneinander
- Parallele Bearbeitung durch die Nodes möglich!

Reduce

- Zusammenfügen der Teilergebnisse der einzelnen Nodes

Map/Reduce: Beispiel

ID: 4711			
Kunde: Donald			
Positionen:			
Äpfel	5	0,20€	1,00€
Birnen	10	0,30€	3,00€
Lieferanschrift: …			
Payment details: …			

map

Äpfel:

Price: 0,20€

Quantity: 5

Birnen:

Price: 0,30€

Quantity: 10

Äpfel

Price: 1,00€

Quantity: 5

Price: 3,50€

Quantity: 20

Price: 7,00€

Quantity: 40

reduce

Äpfel:

Price: 11,5€

Quantity: 65

Beispiel: Map

Input:

- Jeweils ein einzelnes Dokument
- z.B. Rechnung mit verschiedenen Positionen

Produkt	Anzahl	Einzel-Preis	Preis
Äpfel	5	0,20	1,00
Birnen	10	0,30	3,00

Output:

- Key/Value Paare für jede einzelne Rechnungs-Position
- Äpfel: { anzahl=5, preis=1.00 }
- Birnen: { anzahl=10, preis=3.00 }

Jeder Map-Aufruf ist unabhängig von allen anderen.

→ **Parallelisierbar**!

Beispiel: Reduce

Input:

Map-Ausgaben für einen bestimmten Key

- Von vielen Map-Instanzen
- Hier: Key = Produkt, z.B. Äpfel
- Äpfel: [{ anzahl=5, preis=1.00 }, { anzahl=20, preis=3.50 }, …]

Output:

Berechnung aus den Werten für einen Key

- Hier: Berechnung der Summe (→ Gesamt-Anzahl, Umsatz) für ein Produkt.
- Äpfel: { anzahl=5320, preis=1010.80 }

Optimierung:

Verschiedene Reduce-Nodes berechnen parallel die Werte für verschiedene Keys.

Map/Reduce

Steuerung der Abläufe durch ein **Framework**.

Verbleibende Aufgaben:

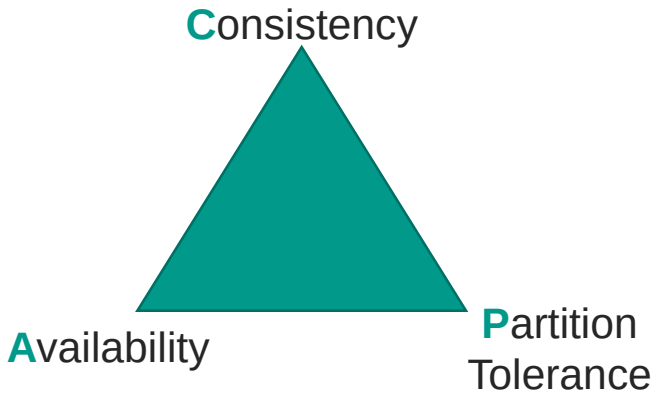
- Programmierung der map- und reduce-Funktionen
  - Werden vom Framework aufgerufen.
  - Keine Komplikationen durch Parallelisierung → einfach zu implementieren
- Registrierung der verfügbaren Nodes am Framework.

Ablauf Map/Reduce:

1. Aufteilung aller Dokumente und Zuweisung zu Map-Nodes.
2. Weiterleitung der Map-Ergebnisse an die Reduce-Nodes.
3. Abholen des aggregierten Gesamt-Ergebnisses von den Reduce-Nodes.

CAP-Theorem

- Eric Brewer (2000)
- Lynch+Gilbert (2002): Formaler Beweis



- Aussage:
  Nicht alle drei Eigenschaften sind in einem verteilten System vollständig erreichbar!

- **Consistency**: Konsistenz (Lese-/Schreib-Konsistenz)
- **Partition Tolerance**:
  Ausfall von Kommunikationsverbindungen im Cluster wird toleriert.
- **Availability**:
  Nicht-ausgefallene Nodes antworten (!)

CAP-Theorem

- SQL: Fokus auf Konsistenz
  → bei Partitionierung Verlust der Verfügbarkeit

- NoSQL: Aufweichung der starken Konsistenz
  - **BASE** – Basically Available, Soft state, Eventual consistency
  - Erlaubt vorübergehende Inkonsistenzen
- Je nach Anforderung unterschiedliche Konsistenzregeln konfigurierbar:
  z.B. **Quorum-Regeln** (“Schreibvorgang muss auf mehr als der Hälfte der Nodes bestätigt sein.”)

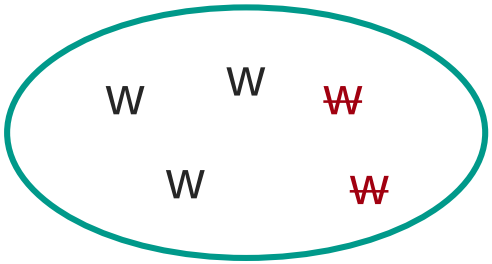
CAP-Theorem und Konsistenz

Wird das eine besser, wird zugleich das andere schlechter

- Tradeoff: **Konsistenz vs. Geschwindigkeit + Verfügbarkeit**
- Vorteil vieler NoSQL-Implementierungen:
  - Konsistenz-Anforderungen sind konfigurierbar.
  - Teilweise pro Anfrage.

- Parameter:
  - N Anzahl der Replikate (Knoten mit Kopien der Daten)
  - R Anzahl der Knoten, die READ bestätigen müssen.
  - W Anzahl der Knoten, die WRITE bestätigen müssen.

- Forderung für **starke Schreib-Konsistenz**:
  $W > \frac{N}{2}$



N=5, W=3

NoSQL

NoSQL

„*Not Only SQL“*

NoSQL steht für „Not only SQL“ für Datenbanken, die nicht zu den traditionellen relationalen Datenbankmanagementsystemen gehören.
Ziele sind ein einfacheres Design, einfachere Skalierung, und Kontrolle über die Daten.
Bei diesen Datenbanksystemen werden die traditionellen relationalen Datenbankstrukturen aufgehoben, so dass Modle implementiert werden können, die näher an den Datenanforderungen sind.

No SQL Systeme = 150 Produkte!

Gruppen:

- Document Stores für Dokumente, ohne Schema, kein Modell
- Key Values Based (Schlüssel -> Wert, ein wenig wie Tupel)
- Column based Stores – Big Table (hier ist Google sehr aktiv)
- Graph wie ein White Board (Weg-Suche, Beziehungsgeflechte)

Document Stores:

- MongoDB ähnlich wie SQL,
- CouchDB – MapReduce,
- Riak (auch Key Value, Jason)
- Lotus Notes (ähnelt CouchDB)

- → Sharding

Kes Value Stores

- Redis: legt auch Datentypen ab, bietet Listenfunktion
- memcache, memcachdb (facebook):
- Riak (eigentlich Document Store)
- BerkeleyDB (oft auf Unix)
- Tokyo Cabinet
- Project Voldemort

Basis der KV ist Performance, Basis für Cches

Wide Column (Column based)

- Big Table (von Google)
- Simple DB (Amazon Webservices AWS)
- Cassandra (Hadoop)
- Hbase (Hadoop)
- Sybase IQ

Infomationen per MapReduce über verschiedenen Rechner durch legen der Spalten auf verschiedenen Rechnern

Graphen Datenbanken

- Neo4J (enthält Wegsuchen-Allgorithmen)
- FlockDB
- InfoGrid
- Trinity (in Csharp)

DBs berücksichtigen verschieden Beziehungsformen

Infomationen per MapReduce über verschiedenen Rechner durch legen der Spalten auf verschiedenen Rechnern

Schwer horizontal skalierbar

Weitere

- OrientDB (kombination)
- Dateisysteme (ähnlich wie WebDAV)
- Integrationen in MySQL (Storage Engines, aber auch memory tables)
- Lucene Search

- Hadoop [hädop] (Basis für NoSQL, aus Apache)
  - Map Reduce Framework
  - Yarn Management (Management, statet Prozesse, monitoring etc)
  - HDFS (Hadoop Distributed File System)
  - Hadoop Core: Hbase, Cassandra,

NoSQL

- Kein definierter Begriff
  … außer anders als SQL-Datenbanken zu sein.
- Fokus auf Performance und Skalierbarkeit
  … auf Kosten der Konsistenz
- An Anforderung optimierte, unterschiedliche Datenmodelle
  Polyglot Persistence

Datenmodelle

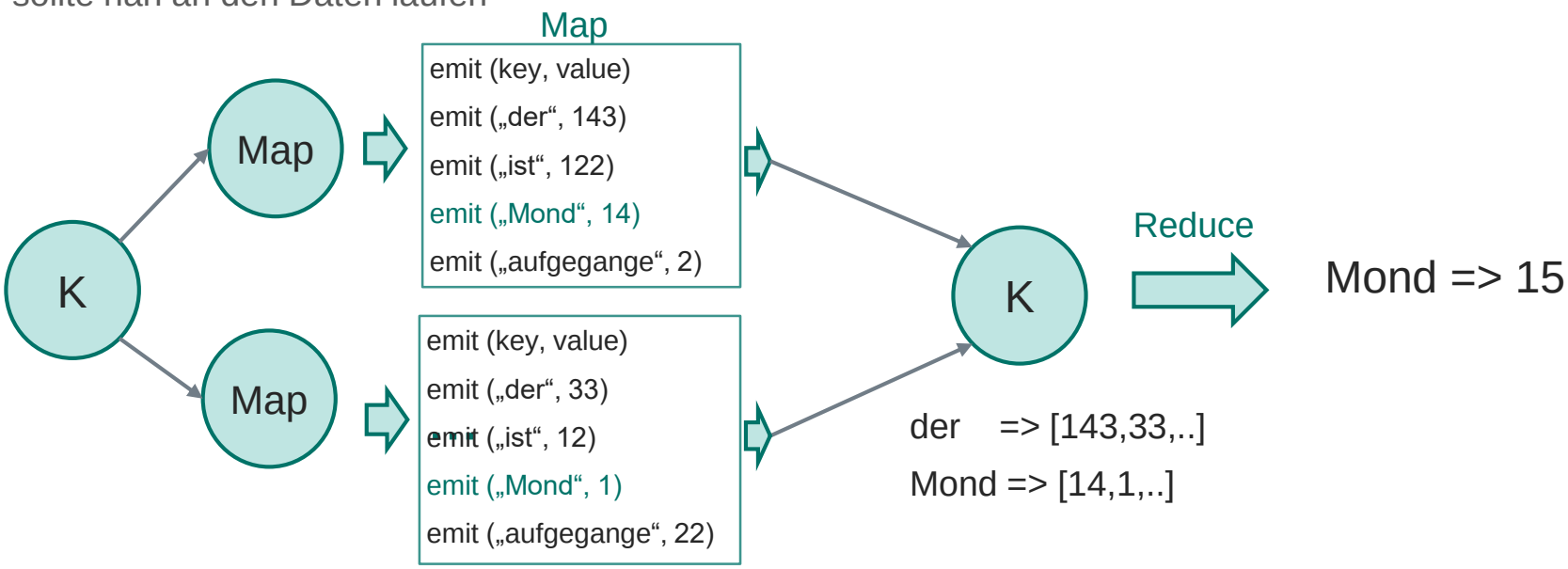
Key-Value-Stores	z.B. Riak
Dokumenten-orientiert	z.B. Mongo (oder Lotus)
Column-Family-Stores	z.B. HBase oder Cassandra
Graph-Datenbanken	z.B. Neo4J

Beispiel: Wörter zählen:

Texte aus einem Projekt werden eingelesen. Es soll ermittelt werden, welche Wörter wie oft verwendet werden.

Koordinator:

Für jedes Kapitel wird ein Prozess angestossen. -> Map Prozess sollte nah an den Daten laufen



NoSQL

- Kein definierter Begriff
  … außer anders als SQL-Datenbanken zu sein.
- Fokus auf Performance und Skalierbarkeit
  … auf Kosten der Konsistenz
- An Anforderung optimierte, unterschiedliche Datenmodelle
  Polyglot Persistence

Datenmodelle

Key-Value-Stores	z.B. Riak
Dokumenten-orientiert	z.B. Mongo (oder Lotus)
Column-Family-Stores	z.B. HBase oder Cassandra
Graph-Datenbanken	z.B. Neo4J

Aggregat-Orientierung

Aggregat

- Hier: Sammlung von zusammengehörigen Objekten, die als Einheit behandelt werden.
- Begriff aus dem *Domain Driven Design*.

- Aggregate werden durch die Anwendung bestimmt.
- Aggregate bei NoSQL: Key/Value, Document, Column
- Denormalisiert

Vorteile

- Aggregate leicht auf Cluster verteilbar.
- Transaktionalität in Bezug auf Aggregate.
  - Atomare Manipulation von Aggregaten.

(Sadalage/Fowler 2013)

Datenmodell: Map (key → value)

- Der value ist für die DB ein Blob
  - *Binary Large Object*
  - DB kümmert sich nicht um die Struktur des Wertes
- Operationen nur anhand des keys:
  - get / put / delete (→ REST)
  - Atomar
- Sharding anhand des keys möglich.

Produkte (Auswahl)

- Riak, Redis, Memcached, Berkeley DB, Amazon Dynamo DB

Use Cases

- Web-Sessions
  - Key = sessionID
- Benutzer-Profile
  - Key = userID

Weniger geeignet für

- Suche nach Eigenschaften der Values (Daten)
- Beziehungen zwischen Daten
- Bearbeitung mehrerer Werte auf einmal

Problem: Diese Anforderungen können sich *später* ergeben.

Beispiel: RIAK KV

- http://basho.com/products/#riak
  http://basho.com/products/riak-kv/
- Distributed Data Store
- Fokus: Verfügbarkeit, Skalierbarkeit
- Multi-Datacenter Replication
- REST-API über http

- **Buckets**: Namensräume für KV-Paare, z.B. für session, user
  - /buckets/session/keys/a7e508154711
  - /buckets/user/keys/

Beispiel: RIAK KV

Zu jedem gespeicherten Wert gehören Metadaten

- Content-Type (z.B. text/plain, text/json)
- Last-Modified

Riak nach Eigenaussage gut anwendbar für:

- Immutable data
- Small objects (< 1MB)
- Independent objects
- Objects with “natural” keys
  - Key muss durch Anwendung bestimmt werden.
- Data compatible with Riak Data Types
  - counters, sets, maps

Dokumenten-orientierte DB

DBMS versteht Struktur der Daten („Dokumente“)

→ Abfragen möglich

Beispiel **MongoDB** https://www.mongodb.org/

- Dokument-Format: **BSON** (Binary JSON)
  - JSON + date + byte-array
  - Jedes Dok hat eindeutig _id
- Strukturierung: Databases → Collections → Documents
- **Schemalos**: Jedes Dokument kann andere Struktur haben
  - Implizites Schema i.d.R. durch Applikation festgelegt.
- **Auto-Sharding** anhand eines Sharding Keys
  - z.B. Location

Beziehungen: Einbetten oder Referenzieren

Einbetten

```
// Document Order:
{
  orderID: 42;
  customer: {
      id: 112; name:
"Hans"
  }
}
```

Vorteile

- Daten-Lokalität des Aggregates
  → schneller Zugriff Order→ Customer
- Atomarer, isolierter Zugriff

Nachteil

- Größe der Dokumente?
  Wichtig bei 1:n

Referenzieren

```
// Document Order:
{
  orderId: 42;
  customerID: 112
}
// Document Customer
{
  customerID: 112;
  name: "Hans";
  orders: [ 13, 42 ]
}
```

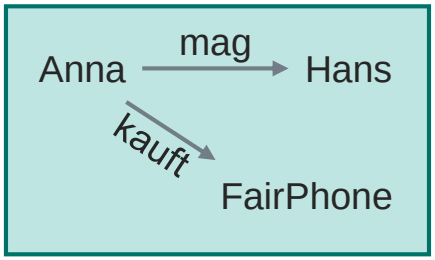
Vorteil

- Flexiblität bei Anfragen
  - Finde Bestellungen des Customers

Nachteil

- 2 Abfragen für „Finde Customer zur Bestellung 42“
- Bidirektionalität aufwändig

Graphen-DB / Neo4J



- Entitäten (Knoten)
- Gerichtete Beziehungen (Kanten)
- Fokus: Abfragen entlang der Beziehungen
  - Alle Personen, die mit Freunden von „Alice“ befreundet sind.
  - Alle Produkte, die von Personen gekauft wurden, die Produkt X gekauft haben.

Use Cases

- Soziale Netzwerke
- Empfehlungssysteme
- Knowledge Engineering

Beispiel

- Neo4J: https://neo4j.com/

Polyglot Persistence

Beobachtung

- Anforderungen an Data Stores differieren stark.
  - Auch innerhalb einer Anwendung!
  - SQL nicht für alle Anforderungen geeignet, v.a. Skalierbarkeit.

→ Idee: Verwendung unterschiedlicher Data Stores.

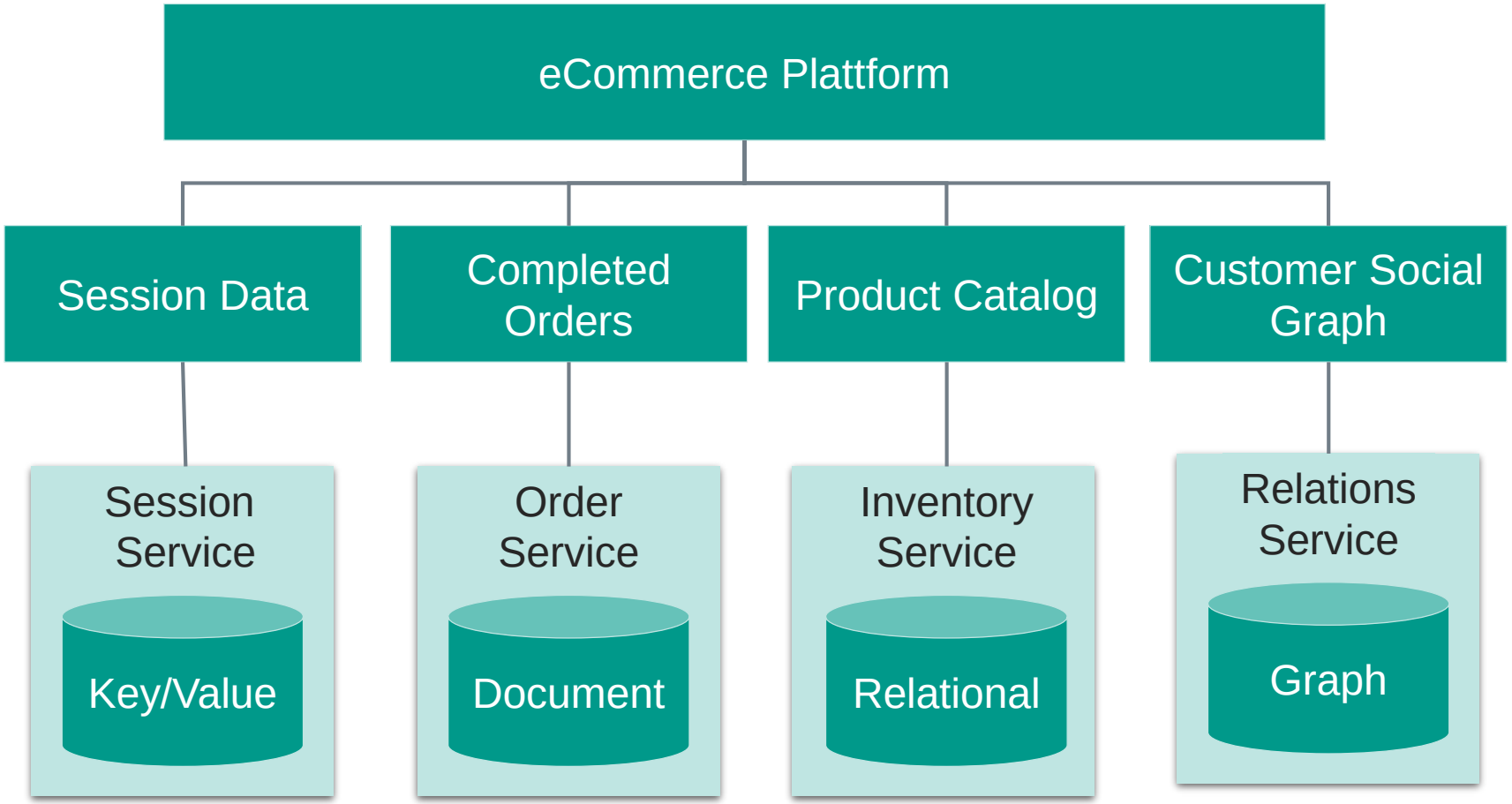
Beispiel: eCommerce-Plattform

- Session Data (Einkaufswagen): Key-Value-Store
- Bestellungen: Document Store
- Produktkatalog: Relational
- Kunden-Netzwerk: Graph

Rückfall in Vor-DBMS-Zeiten???

Polyglot Persistence

Für verschiedene Anforderungen das jeweils beste DBMS nehmen:



Nach [Sadalage/Fowler 2013]

Zusammenfassung

- Map/Reduce
- Sharding
- CAP-Theorem, BASE

- NoSQL
- Key-Value Store
  Beispiel RIAK KV
- Document Store
  Beispiel MongoDB
- Graph DB
  Beispiel Neo4J

- Polyglot Persistence, Aggregat-Orientierung