Wintersemester 2015/16

Prof. Dr.-Ing. Sebastian Michel
TU Kaiserslautern

smichel@cs.uni-kl.de

NoSQL

HOW TO WRITE A CV

Leverage the NoSQL boom

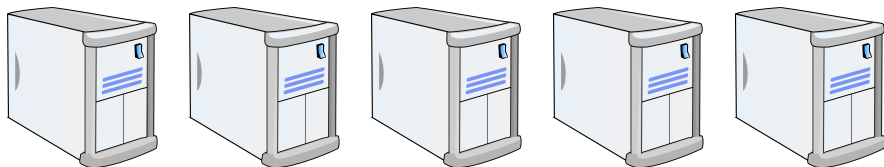
Was steckt hinter NoSQL?

- Beobachtung/Hypothese: Es gibt kein one-size-fits-all Datenbanksystem!
- NoSQL = Not Only SQL (nicht unbedingt “no” SQL)
- Steht als Bezeichner für eine Vielzahl von nicht traditionellen Datenmanagement-Systemen, die stark auf die Anwendung zugeschnitten sind:
  - Key-Value-Datenbanken
  - Graph-Datenbanken
  - Dokument-Datenbanken

Überblick gibt es unter: `http://nosql-database.org/`

Charakteristika von NoSQL Systemen

- Kein relationales Datenmodell
- System sind ausgelegt horizontal zu skalieren (scale out), also Daten und Datenverarbeitung über mehrere Maschinen zu verteilen.
- Kein Schema oder nur sehr lose beschrieben.
- Einfache API (normalerweise keine Unterstützung von SQL): CRUD (create, read, update, delete).
- Üblicherweise keine ACID Semantik. Stattdessen: BASE ;)
- Oftmals sind diese System open-source.



NoSQL: Key/Value Datenbanken

- Speichern von Key-Value Paaren
- Values können komplexe(re) Datentypen sein
- Beispiele von Systemen: Amazon Dynamo, Redis, Voldemort
- Zugriff via **CRUD**-Operationen: **C**reate, **R**ead, **U**pdate, **D**elete
- Einige Systeme unterstützen auch mächtigere/komplexere Anfragetypen.

Beispiel: Key-Value-Store: Redis

- Online Tutorial: `http://try.redis.io`

Get und Set

SET name "Datenbankanwendung"

GET name → Datenbankanwendung

Operationen auf Listen

LPUSH meineListe "a"

LPUSH meineListe "b"

LLENGTH → 2

LRANGE meineListe 0 1 → "b","a"

Dokumenten Datenbanken

- Speichern JSON (Javascript Object Notation), siehe Beispiel links, oder XML Dokumente
- Systeme: MongoDB oder CouchDB

```
{
 "firstName":"John",
 "lastName":"Smith",
 "age":25,
 "address":{
   "street":"21 2nd Street",
   "city":"New York",
   "state":"NY",
   "postalCode":10021
 },
 "phoneNumbers":[
   {
     "type":"home",
     "number":"212 555-1234"
   },
   {
    "type":"fax",
    "number":"646 555-4567"
   }
 ]
}
```

Beispiel: MongoDB

- Online Tutorial: `http://try.mongodb.org`

```
var student = {name:'Jim', scores:[75,99,87.2]};
db.lecture.store(student);

db.lecture.find();        --liefert alle Eintraege
db.lecture.find({name:'Jim'});   --spezielle Suche
db.users.update({name:'Johnny'},{name:'Cash',
            languages:['english']});
```

- MongoDB unterstützt MapReduce `http://docs.mongodb.org/manual/tutorial/map-reduce-examples/`

Übersicht VL

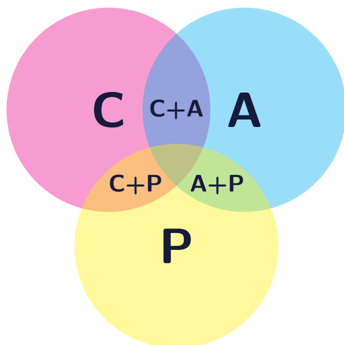
Im Folgenden schauen wir uns an

Konsistenz von Replikaten in Verteilten Systemen

Wie können Daten auf Server platziert/abgebildet werden

CAP Theorem

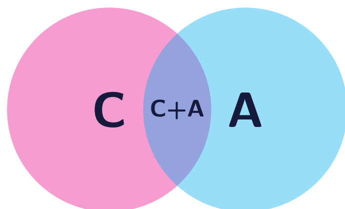
- **Ein System kann nicht gleichzeitig die folgenden drei Eigenschaften unterstützen:**
  - **Consistency** (Konsistenz)
  - **Availability** (Verfügbarkeit)
  - **Partition Tolerance** (Daten/verarbeitung verteilt auf mehrere Maschinen)



http://webpages.cs.luc.edu/~pld/353/gilbert_lynch_brewer_proof.pdf

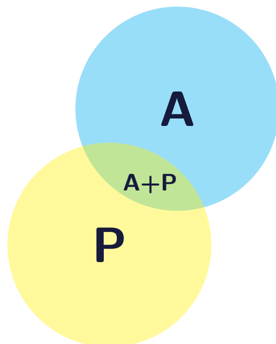
Consistent + Available

- Beispiel: Traditionelle (zentralisierte) Datenbanksysteme



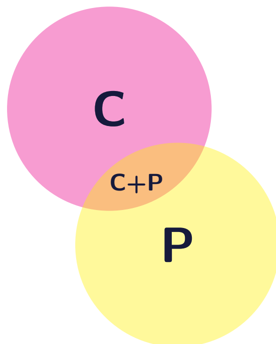
Partition Tolerant + Available

- Beispiel: Domain Name Service (DNS)



Consistent + Partition Tolerant

- Beispiel: Verteilte Datenbanken mit verteiltem Locking/Commit



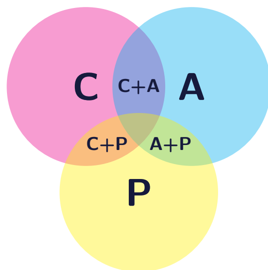
Und nun?

Ohne "P" geht es nicht

- Es sind große Datenmengen zu verarbeiten

  ⇒ Horizontale Skalierung

  D.h. das System muss Partitionierung der Daten/Verarbeitung auf verschiedene Maschinen unterstützen.
- Also ist "P" gegeben. Was ist nun zu tun?



Und nun?

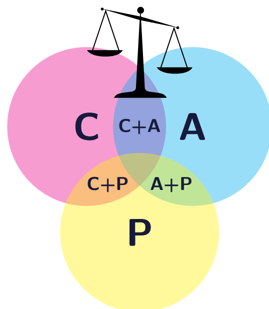
Ohne "P" geht es nicht

- Es sind große Datenmengen zu verarbeiten

  ⇒ Horizontale Skalierung

  D.h. das System muss Partitionierung der Daten/Verarbeitung auf verschiedene Maschinen unterstützen.
- Also ist "P" gegeben. Was ist nun zu tun?

Abwägung (Tradeoff) zwischen Konsistenz und Verfügbarkeit.



BASE

Basically Available

Soft State

Eventual Consistency

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

Idee

Tradeoff zwischen Consistency und Availability.

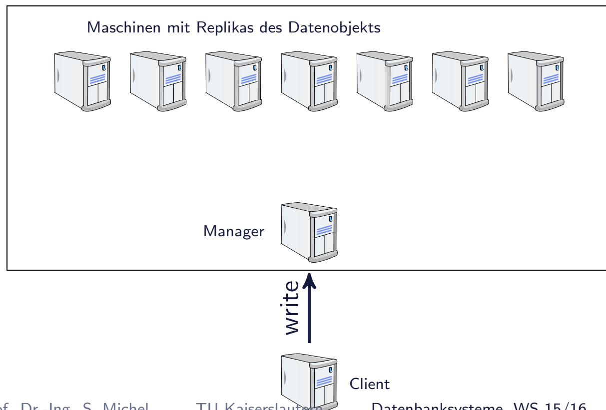
- Repliziere Daten, d.h. mehrere Versionen pro Datensatz
- Replikate werden auf Maschinen verteilt
- Sende Updates an alle Replikate aber warte nicht auf Acknowledgement.
- Lesen Daten von Teilmenge der Replikate.
- D.h. sehr effizient aber nicht unbedingt garantiert konsistent (man kann alte Antworten erhalten)
- Erst nach einiger Zeit konsistent (wenn alle Replikate aktualisiert wurden): Eventual Consistency

Bemerkung zu Consistency

- Consistency hier anders definiert als im DB-Kontext (in ACID).
- Hier, generell um Konsistenz von Replikaten (Kopien) einzelner Datenobjekte; dem Erzwingen bzw. nicht Erzwingen von garantierter Synchronisation.

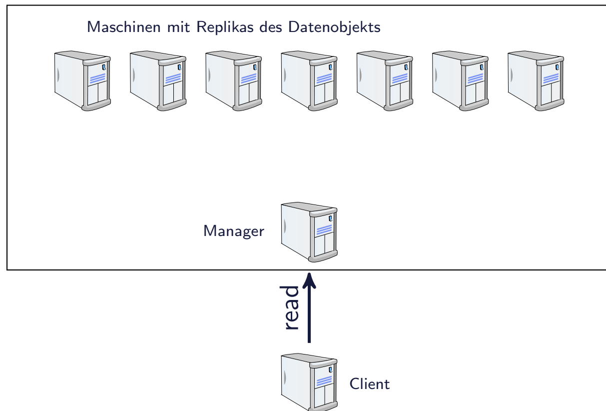
Veranschaulichung Inkonsistenz

- Client schickt **Schreibanweisung** an Manager
- Dieser schickt Schreibanweisung **an alle** Replikate.
- Zeitstempel (im einfachsten Fall) beschreibt Zeitpunkt des Schreibens.
- Und schickt Acknowledgement zurück an Client **sobald garantiert** W **Replikas aktualisiert wurden**.



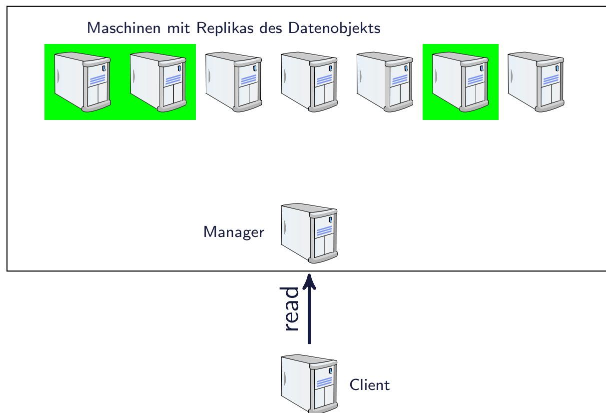
Veranschaulichung Inkonsistenz (2)

- Client schickt **Leseanweisung** an Manager.
- Dieser leitet Anweisung an R Replikas. D.h. **von N existierenden Replikaten werden R gelesen.**
- Antworten werden an Client weiter geleitet.



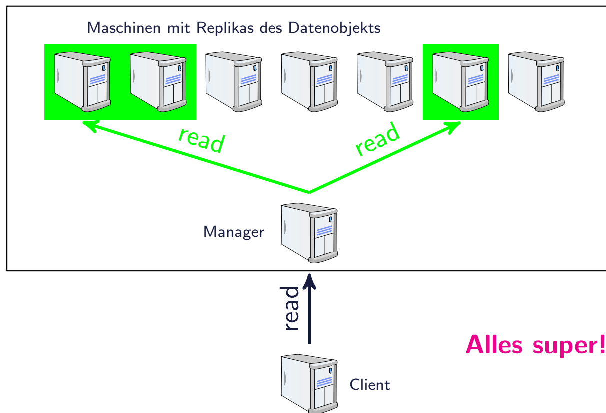
Veranschaulichung Inkonsistenz: Read

- $N = 7$ Replikate
- $W = 3$ und $R = 2$
- Grün markiert sind Replikate, die die neue Version des Objekts besitzen



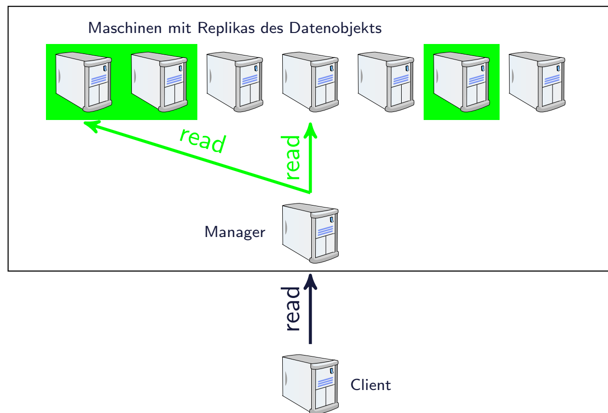
Veranschaulichung Inkonsistenz: Read - OK

- $N = 7$ Replikate
- $W = 3$ und $R = 2$
- Grün markiert sind Replikate, die die neue Version des Objekts besitzen



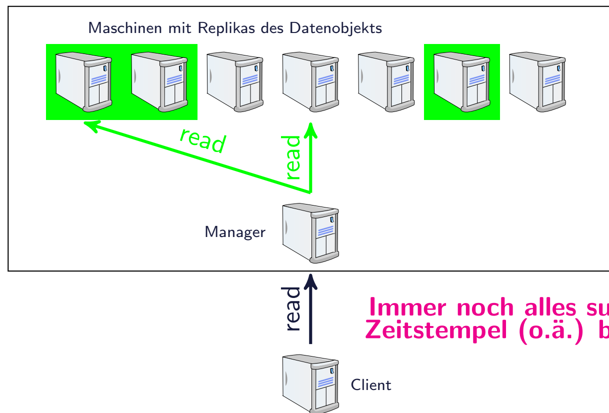
Veranschaulichung Inkonsistenz: Read - OK

- $N = 7$ Replikate
- $W = 3$ und $R = 2$
- Grün markiert sind Replikate, die die neue Version des Objekts besitzen



Veranschaulichung Inkonsistenz: Read - OK

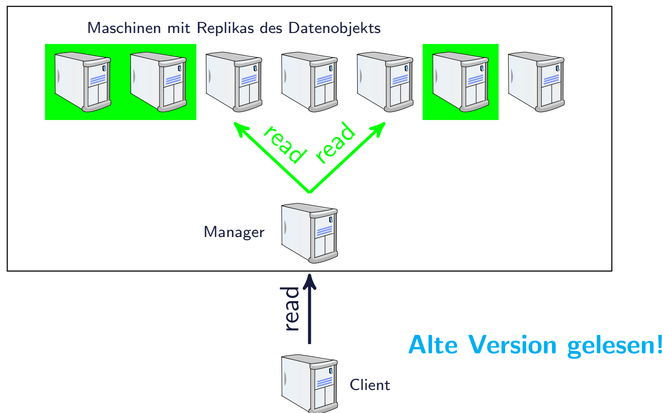
- $N = 7$ Replikate
- $W = 3$ und $R = 2$
- Grün markiert sind Replikate, die die neue Version des Objekts besitzen



Immer noch alles super!
Zeitstempel (o.ä.) bestimmt neueste Version.

Veranschaulichung Inkonsistenz: Read - Nicht Korrekt

- $N = 7$ Replikate
- $W = 3$ und $R = 2$
- Grün markiert sind Replikate, die die neue Version des Objekts besitzen



Write- bzw. Read-Optimized Strong Consistency

Wir möchten ein System, wie zuvor beschrieben, so konfigurieren, dass es jederzeit konsistent ist.

Write-Optimized

Read-Optimized

Write- bzw. Read-Optimized Strong Consistency

Wir möchten ein System, wie zuvor beschrieben, so konfigurieren, dass es jederzeit konsistent ist.

Write-Optimized

- W=1, R=N
- Beim Lesen finden wir garantiert die zuletzt geschriebene Version.

Read-Optimized

- R=1, W=N
- Da alle Knoten die aktuelle Version haben, reicht es von einem Knoten zu lesen.

Eventual Consistency und Konfigurationen

Konfiguration: $R + W > N$

- In diesem Fall kann garantiert werden, dass immer die aktuelle Version gelesen wird.
- Da sich die Mengen der aktualisierten Replikate und die der angefragten Replikate **überlappen müssen!**

Konfiguration: $R + W \leq N$

- In diesem Fall liegt Eventual Consistency vor.

Eventual Consistency

- Eventual (auf Deutsch: letztendlich) Consistency **beschreibt, dass nach einer gewissen Zeit alle Replikate aktualisiert sind.** Aber ab dem Schreibvorgang bis zu diesem Zeitpunkt ist nicht garantiert, dass Lesevorgänge die zuvor geschriebene Version sehen.

Abbildung von Daten auf Knoten

Gegeben Daten mit Schlüsseln 13, 34, 11, 9
und Hashfunktion $f(k) := 17 \times k \textbf{ mod } m$

Für $m = 4$ Knoten

Knoten	Daten
0	
1	13, 9
2	34
3	11

Abbildung von Daten auf Knoten

Gegeben Daten mit Schlüsseln 13, 34, 11, 9
und Hashfunktion $f(k) := 17 \times k \textbf{ mod } m$

Für $m = 4$ Knoten

Knoten	Daten
0	
1	13, 9
2	34
3	11

Für $m = 5$ Knoten

Knoten	Daten
0	
1	13
2	11
3	34, 9
4	

Consistent Hashing

Anforderungen

- Nur **lokales Verschieben**: Wenn neue Maschinen hinzukommen oder Maschinen entfernt werden sollen Daten nur lokal neu organisiert werden.
- Ebenso: **Lastbalancierung**, d.h. Knoten bekommen gleiche Menge an Daten ab.

Consistent Hashing

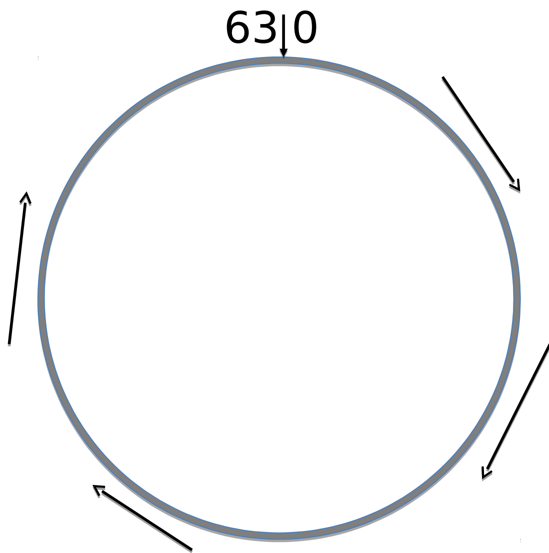
Anforderungen

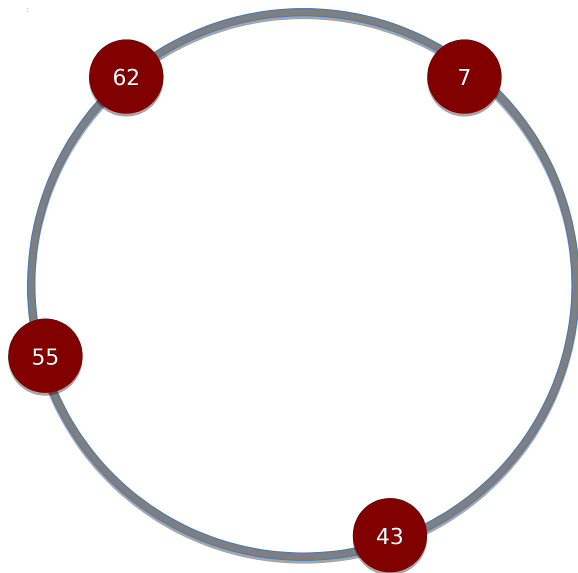
- Nur **lokales Verschieben**: Wenn neue Maschinen hinzukommen oder Maschinen entfernt werden sollen Daten nur lokal neu organisiert werden.
- Ebenso: **Lastbalancierung**, d.h. Knoten bekommen gleiche Menge an Daten ab.

Consistent Hashing

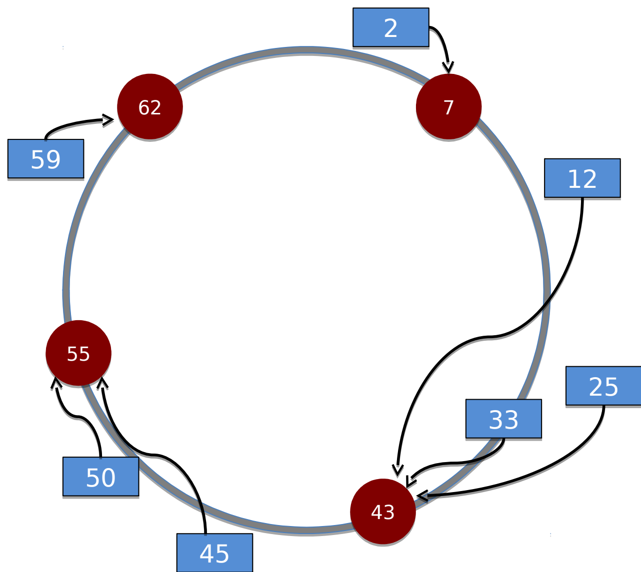
- Weise allen Knoten und Datenelementen je eine ID zu
- z.B. von 0 bis 63 auf "zyklischem Identifier-Space"
- Dann: Weise Daten dem nächsten Knoten mit größerer ID zu.

Karger et al.: Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. STOC 1997.


63|0



Weise Knoten einen Platz auf dem Identifier-Ring zu, z.B. mittels "normaler" Hashfunktion.

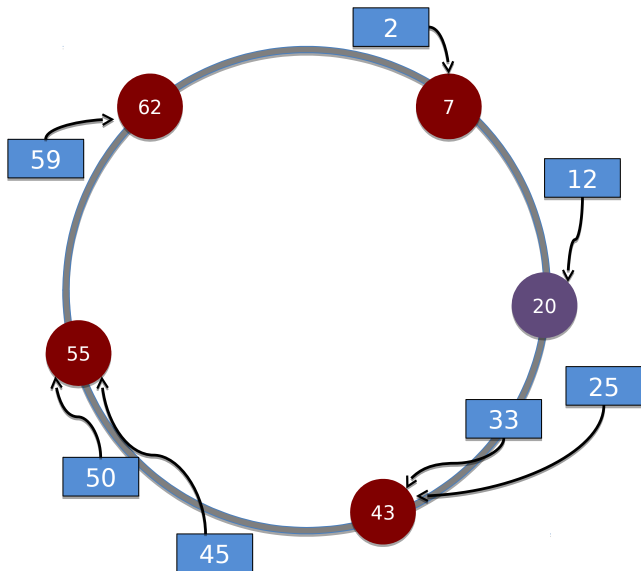


Daten werden ebenfalls auf Ring platziert.

Weise Item mit key i dem Knoten mit id j zu, sodass $j \geq i$ und es existiert kein Knoten j' mit $j > j' \geq i$

(Achtung: Spezialfall am Anfang/Ende des Rings beachten)

Hinzfügen von Knoten



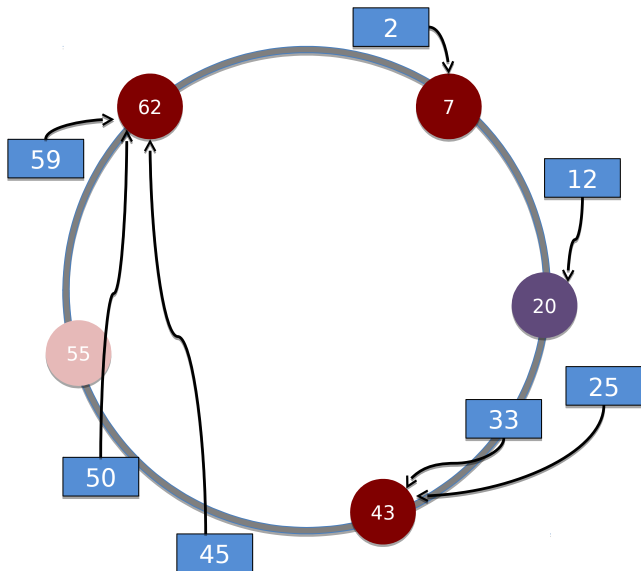
Knoten mit id 21 kommt neu hinzu.

Was muss angepasst werden?

Nur der Datensatz mit key 12 muss verschoben werden.

Also, keine globale Reorganisation!

Entfernen von Knoten

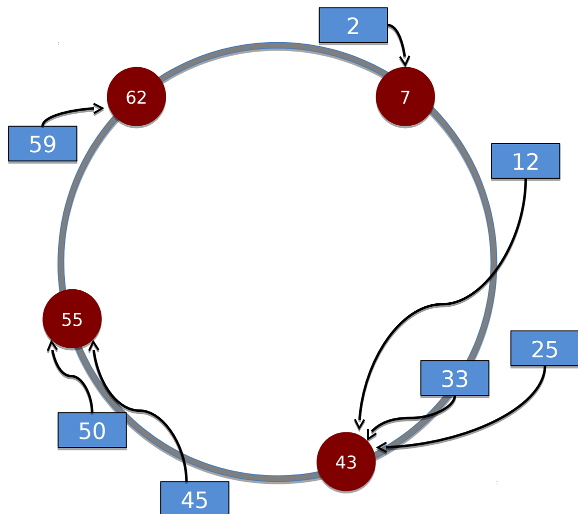


Knoten mit id 55 fällt weg, was muss getan werden?

Daten mit id 50 und 45 müssen dem Knoten 62 zugewiesen werden.

Also, keine globale Reorganisation!

Kosten für Suche (aka. Lookup) / Monotonie



Ausgehend von einem Knoten, wie kann man den Knoten für $f(key)$ finden?

Wie kann man formal beschreiben was "Consistent" heißt?

Consistent Hashing: Eigenschaften und Implementierung

Monotonie

Sei $f(V,i)$ eine Hashfunktion die Schlüssel i auf einen der Knoten der Menge V abbildet. Gegeben zwei solche Mengen von Knoten, V_1 und V_2, mit $V_1 \subseteq V_2$, dann muss gelten:

$$f(V_2, i) \in V_1 \Rightarrow f(V_1, i) = f(V_2,i)$$

(Lineare, Logarithmische oder Konstante) Lookup-Kosten

- Naive Implementierung: Lineare Kosten für Suche nach Knoten für einen Schlüssel
- Besser: Jeder Knoten mit id i speichert Verweise auf Knoten, die für $(i+2), (i+4), (i+8), (i+16), \ldots$ zuständig sind. Periodische Instandhaltung. Dann: logarithmische Kosten für Suche.
- Oder: Jeder Knoten kenne id und Zuständigkeitsbereich aller anderen Knoten. Dann: Konstante Lookup-Kosten.

Ausblick

- Viele weitere Problemstellungen in der Verarbeitung von Daten in verteilten Systemen.
- Wie können Systeme ausfallsicher gemacht werden ("fault tolerance")?
- Wie können Ereignisse global geordnet werden?
- Wie können Replikate garantiert konsistent gehalten werden?
- Welche Abstufungen gibt es bzgl. Konsistenz aus Sicht von Anwendern oder global?
- ...

VL Distributed Data Management (DDM), SoSe 2017

CAP Theorem (Brewer's Theorem)

- System cannot provide all 3 properties at the same time:
  - Consistency
  - Availability
  - Partition Tolerance

http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf

The 4th Paradigm

- For scientific discovery, traditionally
  - experimental (since thousands of years)
  - theoretical (since hundreds of years)
  - computational (like simulations) (since few decades)
  - Now: data driven (i.e., discovery through analyzing huge amounts of data)

Read on: http://research.microsoft.com/en-us/collaboration/fourthparadigm/

Graph Processing in MapReduce

- No global state in MapReduce
- Need to pass on results AND graph structure

```
map(id, node) {
  emit(id, node)
  partial_result = local_compute()
  for each neighbor in node.adjacencyList {
    emit(neighbor.id, partial_result)
  }
}
```

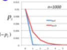
Anti-Entropy as Secondary Protocol (2)

- If used in the case of majority of nodes are in sync already, then
- Pull or pull-push is much better suitable then push only. Why?

Say P_i is probability that a node is not informed, then in next round

for pull: $P_{i+1} = (P_i)^2$

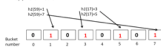
for push: $P_{i+1} = P_i * (1 - \frac{1}{n})^{n(1-P_i)}$



source: Demers et al.

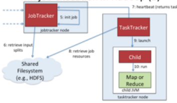
Bloom Filter: Insert + Query

h1(x) = 3*x mod 8
h2(x) = 5*x mod 8

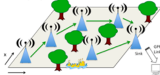


- Query: is x contained in the set (=filter)?
  - Check if bits at both h1(x) and h2(x) are set to 1. Yes? Then x "might be" in the set. No? Then x is for sure not in!

MR job execution in Hadoop (2)



2D Deployment/Infrastructure Example



- Multi-hop communication toward one sink node.
- Or direct communication to consumer
- Or Sensors might move, creating ad-hoc networks
- ...

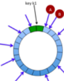
Topology Builder

- Operator/Source Topology is created by registering consumers to producers using unique names of sources/operators.

https://github.com/nathanmarz/storm/wiki/Tutorial

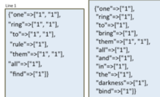
Random Placement with equal Sized Partitions

- Have Q equal sized partitions (Q >> T*S), where
- Nodes are (as before) placed randomly.
- Partition is assigned to N nodes that follow (successors) the end of the partition.



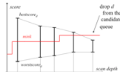
Decoupling of partitioning and partition placement. Partition bounds don't change. Efficient maintenance.

Map Line to Terms and Counts



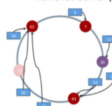
Evolution of a Candidate's Score

Observation: pruning often overly conservative (deep scans, high memory consumption)



- Approximate top-k
  - "What is the probability that d qualifies for the top-k ?"

Removed Server (id 55)



Suffix Based

- Emit only suffixes in map phase
- Each of them represents multiple n-grams corresponding to its prefixes
  - For instance, axbbay represents
    - a, ax, axb, axbb, axbba, and axbbay

```
map(did, content):
  for all suffixes in content:
    emit(suffix, did)
```

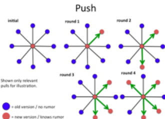
Process 1 | Process 2 | Process 3

Example: Co-occurrences (Cont'd)

- Solution 1: pairs approach
  - mapper for string s:
    - for all term pairs (a,b) in s: emit((a,b), 1)
  - reducer just aggregates counts
- Solution 2: "stripes" approach
  - mapper for string s:
    - collect all t_i that co-occur with a
    - emit (a,{t_1, t_2, ... t_n})
  - reducer aggregates

What is the difference?

Push



http://dbis.informatik.uni-kl.de/index.php/en/teaching/summer-2015/distributed-data-management