NoSQL-Datenbanken

Key-Value Stores

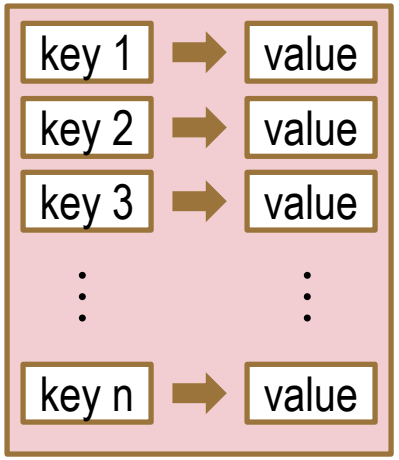
Johannes Zschache
Sommersemester 2019

Abteilung Datenbanken, Universität Leipzig
http://dbs.uni-leipzig.de

Inhaltsverzeichnis: Key-Value Stores

Key-Value Stores

- Einfache, sehr flexible Form eines DBMS
- Datenstruktur: **Assoziatives Datenfeld**
  - Sammlung von Schlüssel-Wert-Paaren
  - aka Dictionary, Map, Hash, Hash Map, Hash Table …
- Einfache API
  - Schreiben: put(key, value)
  - Lesen: get(key)
  - Löschen: remove(key)
- Zusätzliche Funktionen in konkreten KVS:
  - Iteration und Sortierung der Schlüssel
  - Suche über Werte: Inverted Index
  - Operationen auf Zeichenketten/Arrays/Mengen

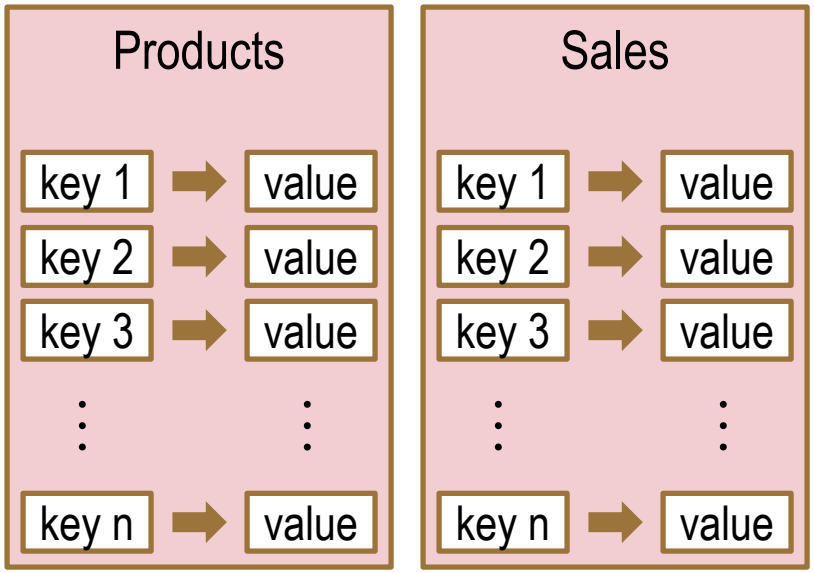


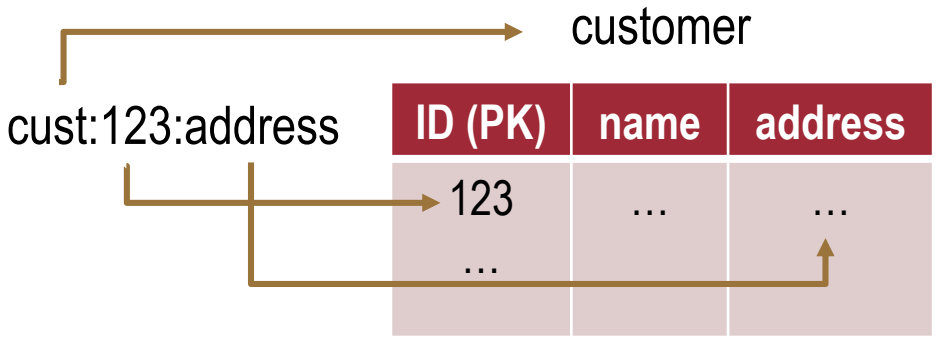
Eigenschaften

- Hohe Lese-/Schreibraten für große, unstrukturierte Datensätze
  - Einfache Verteilung über mehrere Server (Partitionierung nach Schlüssel)
  - Verwaltung im Hauptspeicher
- Einfaches Datenmodell und einfache Anfragen
  - Zugriff & Speicherung über Schlüssel
  - Keine Fremdschlüssel/Joins
  - ***Schemafrei:*** flexibel & tolerant
- Verschiedene Konsistenzstufen
- Ungeeignet für komplexe Anfragen und Aggregationen
  - Anfragen nur über Schlüssel
  - Oft keine Bereichsanfragen möglich
  - Keine standardisierten Anfragesprachen

Schlüssel

- Schlüssel = Verweis auf einen Wert
- Eindeutig innerhalb eines *Namespace* (= Menge von Schlüsseln)
- Zeichenkette oder binäre Sequenz
- Aussagekräftige Bezeichner
- Namenskonventionen
  - Informationen zu Wert
  - z.B: „Product:321:Color“
- Zusammengesetzte Schlüssel
  - für Indizierung und Aggregationen
  - z.B: „DE:Saxony:Leipzig:population“
- Schlüssel mit Zähler
  - für Logging
  - z.B. „ticketLog:20180403:1“
- Bezug zu relationaler Modellierung



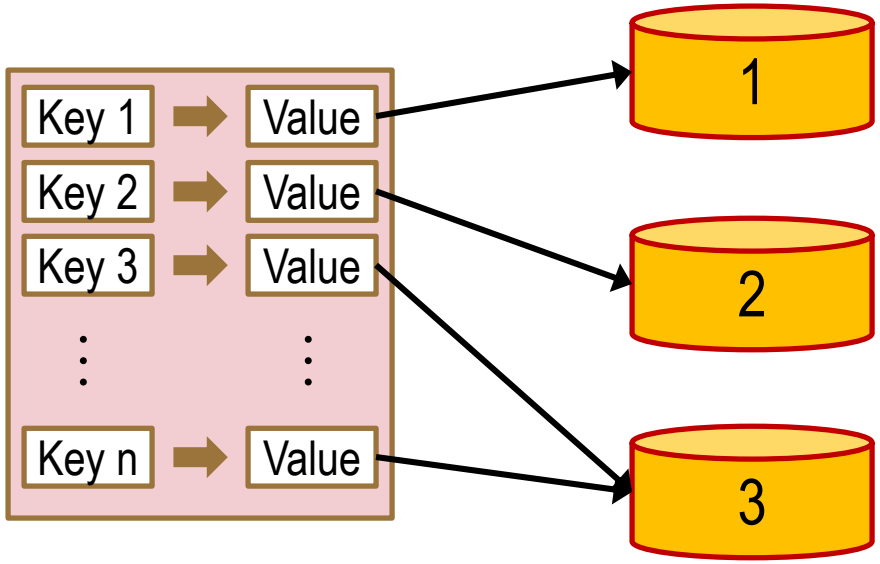


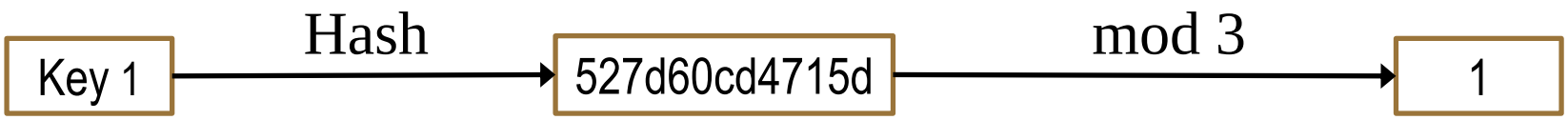
Werte

- Prinzipiell jeder Datentyp möglich: String, Integer, Image, BLOB ..
  - *Schemafrei*: Keine DB-seitigen Integritätsbedingungen
  - Applikationsseitige Überprüfung von Datentyp, Wertebereich und Abhängigkeiten
- DB-spezifische Beschränkungen des Typs oder der Größe
- Das Design eines KVS sollte in Abhängigkeit der zu erfüllenden Funktionalität geschehen
- Designprinzipien
  - Attribute, die oft zusammen benötigt werden (z.B. Name und Adresse), sollten als ein Wert gespeichert werden
  - ***Denormalisierung:***
    - Speicherung von Duplikaten (z.B. zusätzliche separate Speicherung des Namens)
    - Ermöglicht evtl. höheren Durchsatz
    - Mehr Aufwand bei Aktualisierungen
  - Keine zu großen Werten (z.B. verschachtelte Dokumente)

Partitionierung in Key-Value Stores

- Verwendung des Schlüssels eines Schlüssel-Wert-Paares als Partitionierungsschlüssel
- Ziel: Gleichmäßige Verteilung
- Einsatz von Hash-Funktionen





- Key-Value Stores erlauben lose Kopplung der Server
  - Minimale Kommunikation (Weiterleitungen, „still alive“)
  - Bei Ausfall: Übernahme der Anfragen durch andere Server

DB Ranking

Quelle: http://db-engines.com/en/ranking/

67 systems in ranking, April 2019

Rank Apr 2019	Rank Mar 2019	Rank Apr 2018	DBMS	Database Model	Score Apr 2019	Score Mar 2019	Score Apr 2018
1.	1.	1.	Redis	Key-value, Multi-model	146.38	+0.25	+16.27
2.	2.	2.	Amazon DynamoDB	Multi-model	56.01	+1.52	+12.86
3.	3.	3.	Memcached	Key-value	28.73	0.00	-5.06
4.	4.	4.	Microsoft Azure Cosmos DB	Multi-model	26.28	+1.45	+9.09
5.	5.	5.	Hazelcast	Key-value	8.35	+0.33	-0.85
6.	↑ 7.	6.	Ehcache	Key-value	6.49	+0.03	-0.45
7.	↓ 6.	↑ 9.	Aerospike	Key-value	6.21	-0.52	+2.03
8.	8.	8.	OrientDB	Multi-model	6.19	+0.06	+0.55
9.	9.	↓ 7.	Riak KV	Key-value	5.70	-0.11	-0.51
10.	10.	↑ 11.	Ignite	Multi-model	5.09	-0.09	+1.92
11.	11.	↓ 10.	ArangoDB	Multi-model	4.29	+0.03	+0.49
12.	12.	↑ 15.	InterSystems Caché	Multi-model	3.08	-0.07	+0.73
13.	13.	↓ 12.	Oracle NoSQL	Key-value, Multi-model	2.97	-0.17	+0.48
14.	↑ 15.	↓ 13.	Oracle Berkeley DB	Multi-model	2.89	+0.07	+0.45
15.	↓ 14.	↓ 14.	LevelDB	Key-value	2.86	+0.04	+0.50
16.	16.	↑ 19.	RocksDB	Key-value	2.60	+0.12	+1.03
17.	↑ 19.	17.	Infinispan	Key-value	2.47	+0.17	+0.19
18.	↓ 17.	↓ 16.	Oracle Coherence	Key-value	2.40	+0.08	+0.11
19.	↑ 20.	↑ 20.	GridGain	Multi-model	2.35	+0.39	+1.14
20.	↓ 18.	↓ 18.	Amazon SimpleDB	Key-value	2.32	+0.01	+0.43

Inhaltsverzeichnis: Key-Value Stores

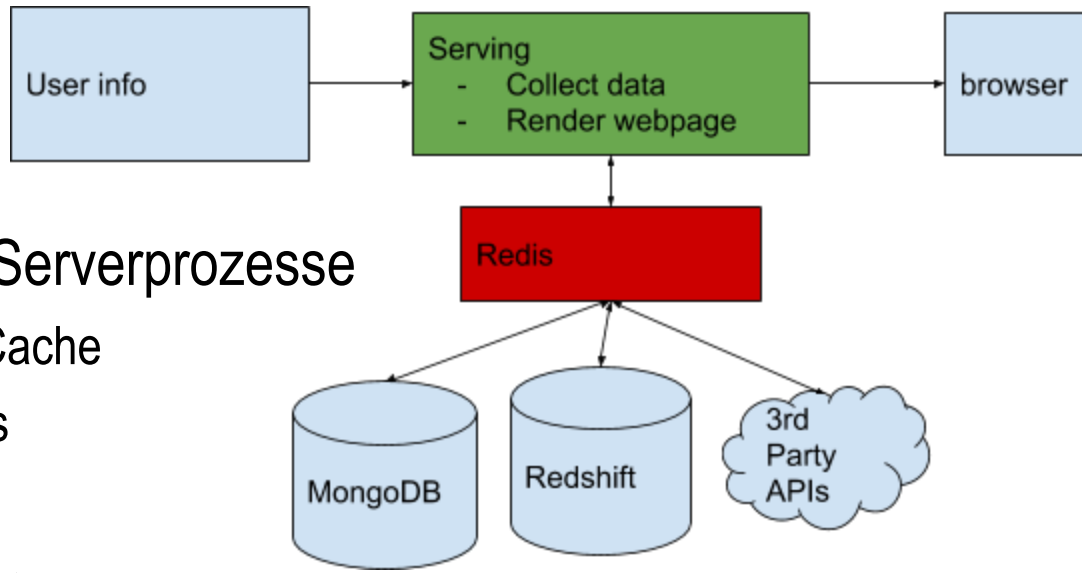
Redis - REmote DIctionary Server

- Single-Threaded Key-Value Store; seit 2009; Aktuelle Version: 5.0
- ***Leistung***: schnelles Lesen und noch schnelleres Schreiben
  - Geschwindigkeit anstatt Persistenz
  - Verwendung z.B. als Cache, Job Queue, oder Message Broker
- ***Data Structure Server****:*
  - Komplexe Datentypen und Mengen-bezogene Anfragen
  - Blocking Queue, Publish-Subscribe System, Streams
- Schemafrei
- Expiry Policies und Modifizierbare Persistenz
- Replikation und Redis Cluster
- Anbindungen für viele Programmiersprachen

Quelle: [Redis]

Redis: Anwendung als Cache

- Aufbau einer HTML-Seite abhängig von Nutzerinformationen, Suchanfragen, Zeitpunkt, Ort, …)
- Flaschenhals: Anfragen der Daten aus DB bzw. von externen APIs
- Automatische Skalierung der Serverprozesse
  - Hauptspeicher ungeeignet als Cache
  - **Auslagerung** des Cache: Redis
- Redis als Cache für
  - Oft gebrauchte MongoDB-Objekte
  - Ergebnisse komplexer MongoDB-Anfragen
  - Rechenintensive Berechnungen



Quelle: https://medium.com/ni-tech-talk/how-we-use-redis-for-real-world-web-application-3b1a44fa6585

Redis: Anwendung als Cache

- Vergleich der Leistung für eine einfache Anfrage (Heraussuchen eines Werts für eine ID aus 100 000 Dateneinträgen)

Solution	Load time (sec)	Size	Number of time loaded	1000 queries time (msec)
Redshift	2.4 sec	40MB	Once	16,000 msec
MongoDB	22.1 sec	30MB	Once	1,500 msec
Redis	41.5 sec	50MB	Once in awhile (TTL dependent) , can be done by a worker machine	400 msec
Memory	3.8 sec	13MB	Once per process on each server	1 msec

Quelle: https://medium.com/ni-tech-talk/how-we-use-redis-for-real-world-web-application-3b1a44fa6585

Inhaltsverzeichnis: Key-Value Stores

Redis: Befehle

- Redis Command Line Interface `redis-cli`
- Set und Get

```
SET dbs https://dbs.uni-leipzig.de/
GET dbs
```

- Erkennung von Zahlen:

```
SET count 2
INCR count
GET count
```

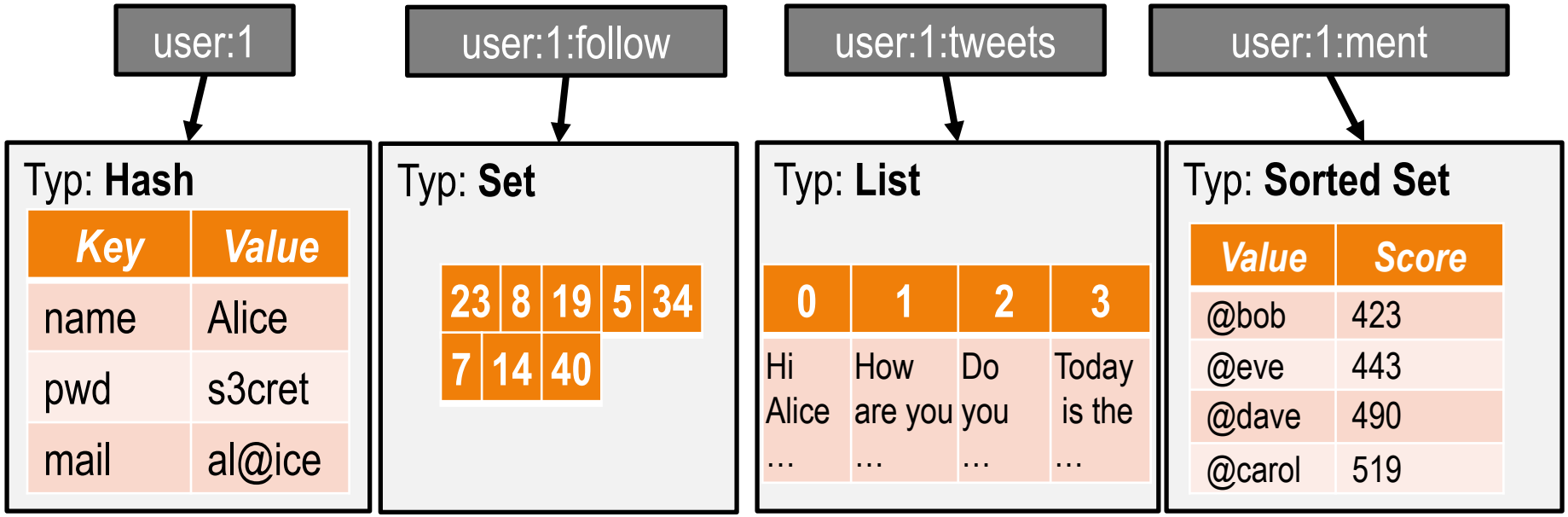
- Transaktionen

```
MULTI
SET unile http://www.uni-leipzig.de/
INCR count
EXEC
```

- Cheatsheet: https://github.com/LeCoupa/awesome-cheatsheets/blob/master/databases/redis.sh

Redis: Datentypen

- **Schlüssel**: String
- **Wert**: String oder *Collection*




```
HMSET user:1
  name Alice
  pwd s3cret
```

```
HGET user:1 name
```

```
SADD user:1:follow
23 8 19 5 34 7 14
```

```
SMEMBERS
user:1:follow
```

```
RPUSH user:1:tweets
 „Hi Alice ...“
 „How ...“
```

```
LPOP user:1:tweets
```

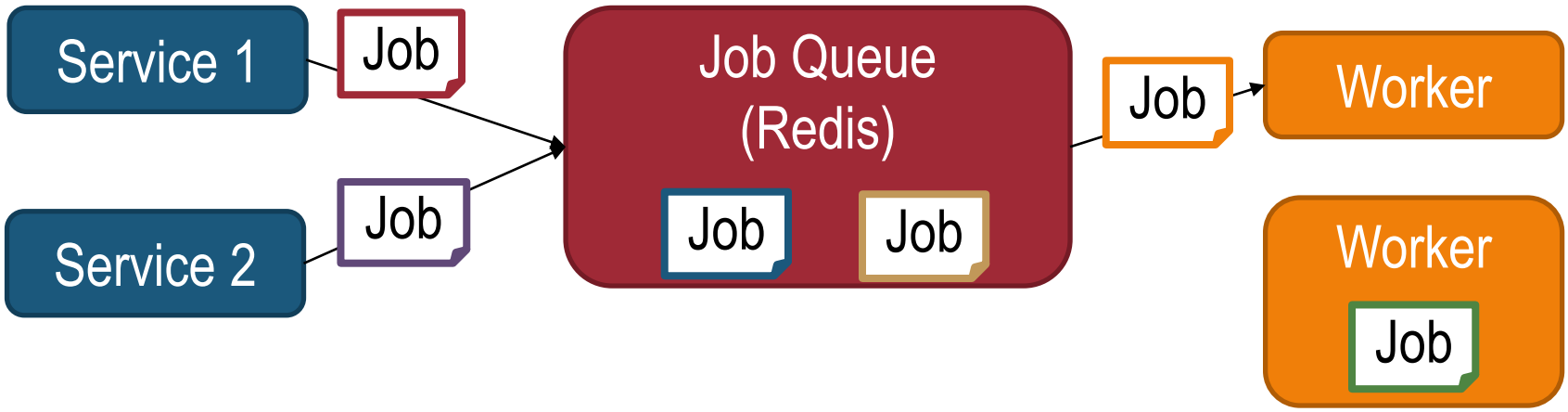
```
ZADD user:1:ment
423 @bob 443 @eve
```

```
ZRANGEBYSCORE
user:1:ment 400 500
```

Redis: Blocking Queue

- Liste mit Blockierungsmechanismus
  - Warten auf Element (mit 300 Sekunden Timeout): `BRPOP comments 300`
  - Hinzufügen von Elementen wie in Listen:

    ```
    LPUSH comments "Prag is great!"
    ```
- Mehrere Leser und Schreiber möglich, aber nur ein Leser pro Element (Leser mit längster Wartezeit)
- Anwendung: Job Queue, z.B. in Microservice-Architektur

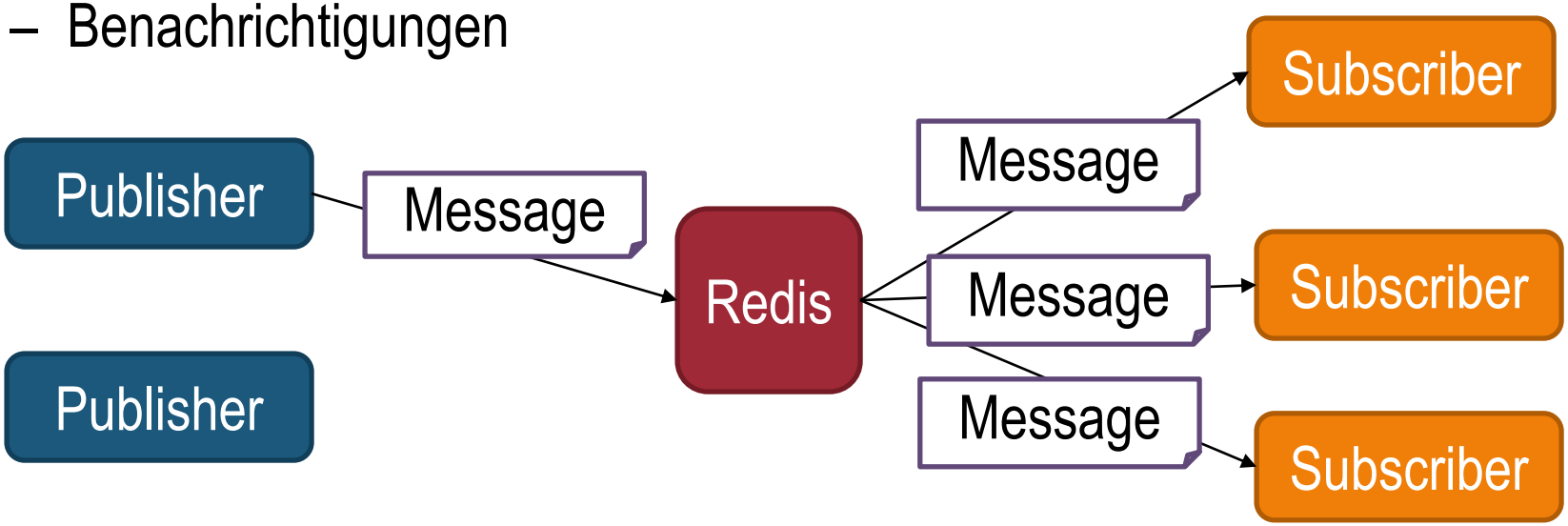


Redis: Publish-Subscribe

- Verteilung von Elementen über mehrere Leser
- Mehrere Schreiber möglich:
- „Fire and Forget“: Elemente werden nicht gespeichert (selbst wenn es keine Leser gibt)
- Anwendungen:
  - Live-Chat
  - Benachrichtigungen

```
SUBSCRIBE comments
```

```
PUBLISH comments "Check this out: … "
```



Redis: Streams

- Datenstruktur für Zeitreihendaten
- Elemente haben Zeitstempel (ID), z.B. 1523166062-0 (Zeit in ms)-(Zähler)
- Elemente eines Stream sind Schlüssel-Wert Paare

```
XADD yourstream * key1 value1 key2 value2
```

- Lesen aller Nachrichten mit ID > 0-0

```
XREAD STREAMS yourstream 0
```

```
1) "yourstream"
2) 1) 1) "1554980685-0"
      2) 1) "key1"
         2) "value1"
         3) "key2"
         4) "value2"
```

- Blockierend; ohne Zeitsperre (0) und nur neue Elemente ($)

```
XREAD BLOCK 0 STREAMS yourstream $
```

- Bereichsanfragen

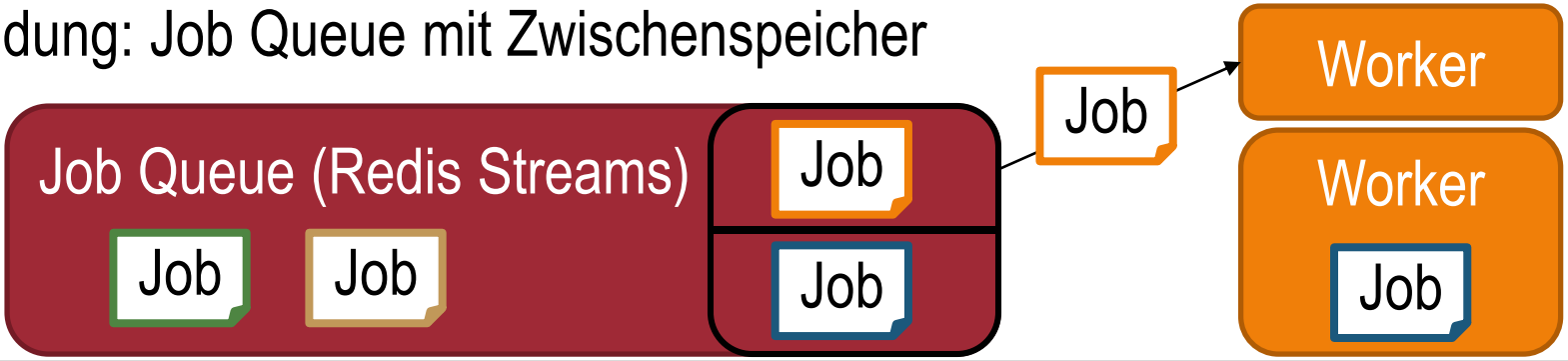
```
XRANGE yourstream - +
XRANGE yourstream 1518951480106 1518951480107
XRANGE yourstream 1518951480107-1 +
```

Redis: Streams

1. Nachrichten können generell von mehreren Nutzern gelesen werden
- Im Gegensatz zu Pub/Sub: Nachrichten werden gespeichert
- Anwendung: Chat mit Speicherung der Nachrichten
  - Neue Nutzer können vergangene Unterhaltungen nachvollziehen
  - Wiederaufnahme der Unterhaltung nach Netzwerkfehler

2. Einschränkung auf nur einen Leser pro Nachricht: **Consumer Groups**
  - Zuordnung von Gruppen zu einem Stream
  - Jeder Nutzer einer Gruppe muss sich eindeutig identifizieren
  - Abfrage von Elementen, die noch kein anderer Nutzer über die Gruppe abgefragt hat
  - Speicherung dieser Zuordnung zu einem Nutzer bis Verarbeitung bestätigt (XACK)

- Anwendung: Job Queue mit Zwischenspeicher



Inhaltsverzeichnis: Key-Value Stores

Redis: Expiration

- Automatisches Löschen von Daten nach einem Zeitraum
- Zeitsperre setzen (in Sekunden)

```
SET ice "I'm melting..." EX 10
```

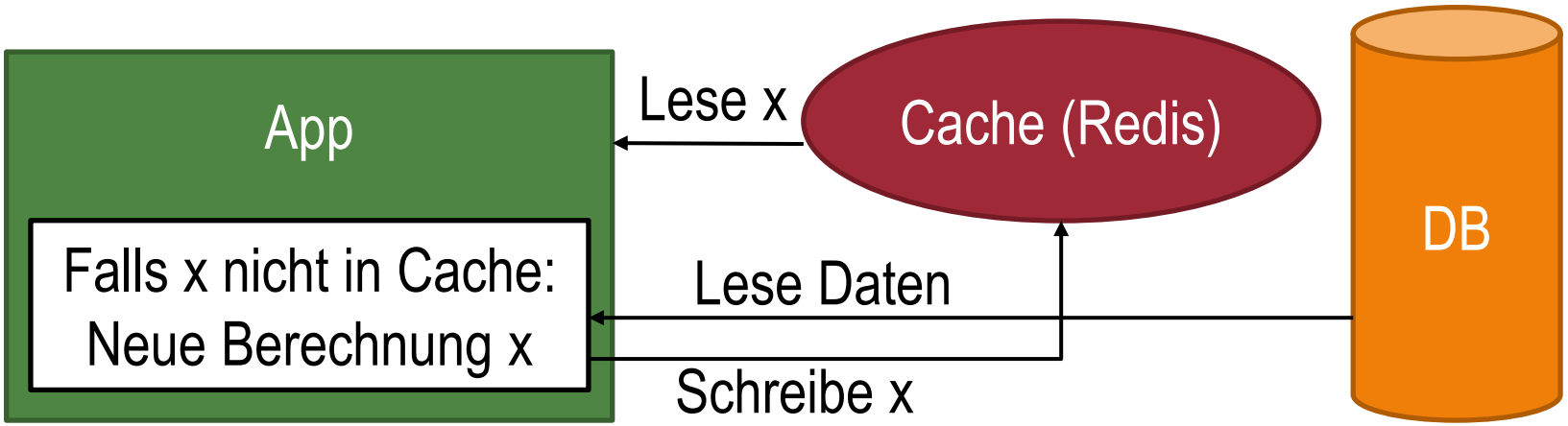
- Time-to-live

```
TTL ice
```

- Zeitsperre entfernen

```
PERSIST ice
```

- **Verwendung**: Caching mit regelmäßiger Erneuerung der Werte
  - z.B. aller 5 Minuten



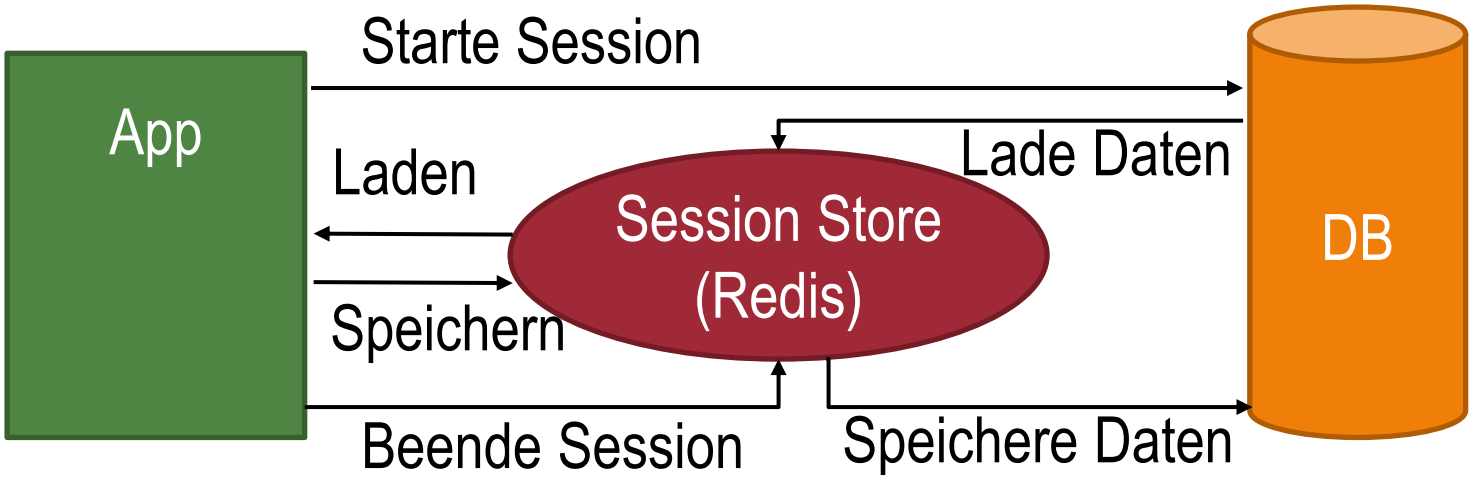
  - **Problem?**

Redis: Persistenz

- Speicherung der Daten **zunächst nur im Hauptspeicher**
- Gefahr des *Datenverlusts*
- **RDB Persistenz** (aka Snapshotting, Periodic Dump)
  - Regelmäßiges Abspeichern der gesamten DB auf Festplatte
  - Durch Nutzer (BGSAVE) oder automatisiert
  - Asynchrone Übertragung der Daten aus dem Hauptspeicher
  - Rate konfigurierbar (redis.conf)
    - z.B. nach 5 Minuten falls mind. eine Aktualisierung: save 300 1
    - z.B. nach 2 Minuten falls mind. 10 Aktualisierungen: save 120 10
  - Redis startet neuen Prozess, welcher sich um die Übertragung der Daten kümmert
    - Schreiben der Daten in temporäre RDB-Datei
    - Ersetzen der alten RDB-Datei
  - Nachteile:
    - Gefahr des Datenverlusts während Snapshotting
    - Redis ist evtl für einige Millisekunden nicht ansprechbar

Redis: Persistenz

- **AOF Persistenz**: Append-Only-Log
  - Speicherung aller Schreibbefehle vor Ausführung → Rekonstruktion nach Absturz
  - Einstellungen für Flush-To-Disk: *Always* (ACID), *Every Second* (Datenverlust von max. 1 Sekunde an Befehlen), oder *No* (Betriebssystem entscheidet)
  - Nachteil: Längere Latenzzeiten
- Ohne RDB/AOF Persistenz: Sinnvoll bei Einsatz als Cache
- Doch Einsatz als z.B. Session-Store erfordert Persistenz



Quelle: https://redislabs.com/blog/cache-vs-session-store/

Redis: Datenkompression

- Keine *eingebaute* Unterstützung automatischer Komprimierung der Daten
- Doch Komprimierung erhöht die Speicherkapazität *und kann sogar die Latenzzeiten verringern*
  - Netzwerkprobleme, falls sehr große Werte sehr häufig gelesen werden
  - Benötigte Zeit für Komprimierung wird über geringe Latenz ausgeglichen
- Fallstudie: DoorDash (Lieferservice)
  - Häufiges Lesen von Speisekarten → Redis als Cache
  - Experiment: Schreiben und Lesen von 10 000 Datensätzen
  - Snappy (https://github.com/google/snappy): 39.71% Kompression (Durchschnitt)
  - LZ4 (https://lz4.github.io/lz4/): 38.54% Kompression (Durchschnitt)

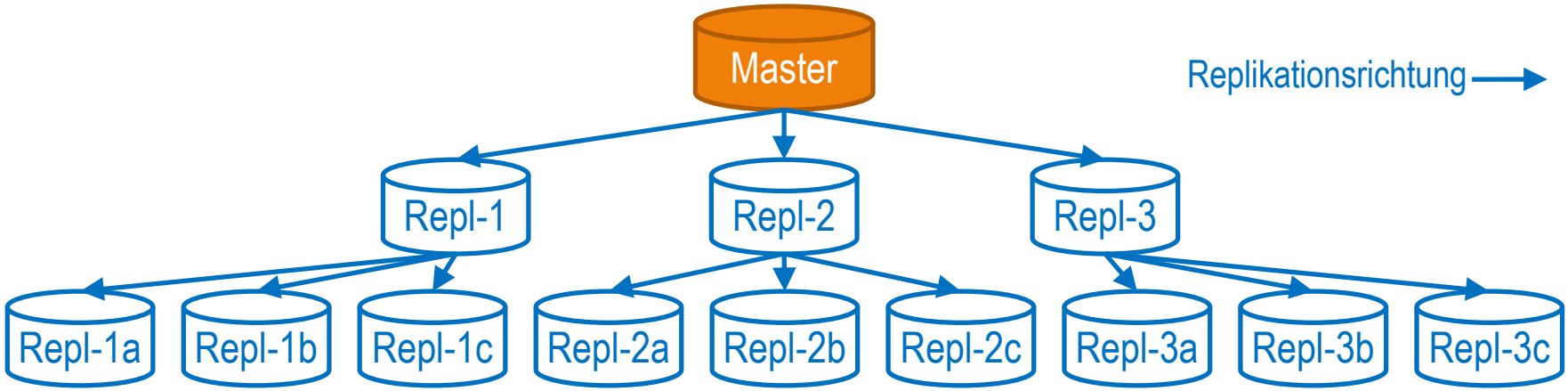
Redis Operation	No Compression (sec.)	Snappy (sec.)	LZ4 (sec.)
Set(10000)	16.53	12.64	12.8
Get(10000)	12.05	7.56	6.43

Quelle: https://doordash.engineering/2019/01/02/speeding-up-redis-with-compression/

Inhaltsverzeichnis: Key-Value Stores

Redis: Replikation

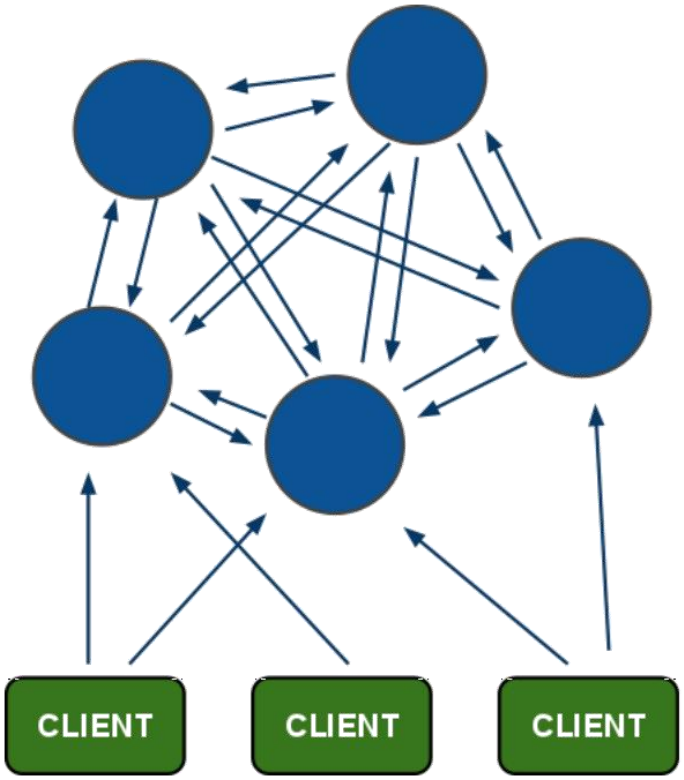
- Unterstützung einer Master-Slave-Replikation
  - Auslagerung von rechenintensiven Leseanfragen an Replikate
  - Replikate auch für Persistenz zuständig
  - Initiale Synchronisation über RDB-Datei
  - Master nimmt Schreibefehle an und leitet diese an Replikate weiter (asynchron)
  - Beliebige Anzahl an Replikate und Baumstruktur (für schnellere Synchr.) möglich



- Erstellung eines Replikats: Kopie `redis-s1.conf`
  - Änderung Port (z.B. 6380) und Eintrag: `slaveof 127.0.0.1 6379`
  - `run redis-server redis-s1.conf`

Redis Cluster

- Automatische Partitionierung der Daten auf bis zu 1000 Servern
- Alle Befehle von Redis werden unterstützt, solange alle Objekte eines Befehls auf gleicher Partition liegen
- Begrenzte Fehlertoleranz über asynchrone Master-Slave Replikation und Wahl eines neuen Master
- Eher CP- als AP-System (CAP):
  - Ausfall bei größeren Netzwerkfehlern
  - Aber nur schwache Konsistenzgarantien: unwahrscheinliche Szenarien können zu Inkonsistenzen führen



Quellen: http://redis.io/topics/cluster-tutorial, http://redis.io/topics/cluster-spec
Bild: http://redis.io/presentation/Redis_Cluster.pdf

Redis Cluster: Sharding

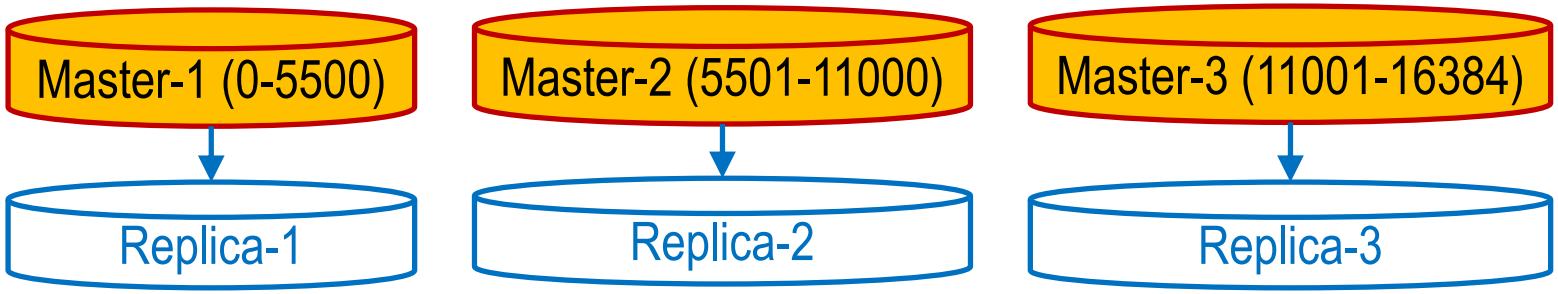
- Jeder Schlüssel wird einem von 16384 Hash-Slots zugeteilt
- Zuordnung: CRC-16 Prüfsumme des Schlüssels modulo 16384

- Jeder Server ist für Teilmenge der Hash-Slots verantwortlich, z.B.

- Manuelles Verschieben von Slots nach Hinzufügen neuer Server bzw. vor dem Entfernen vorhandener Server
- Mit Hashtags kann gleicher Slot erzwungen werden
  - Verwende `{}` in key → nur innerer Teil wird „gehasht“
  - `this{foo}key` und `another{foo}key` landen im gleichen Hash Slot
- Keinen Proxy: Weiterleitung der Anfragen an zuständigen Server

Redis Cluster: Fehlertoleranz

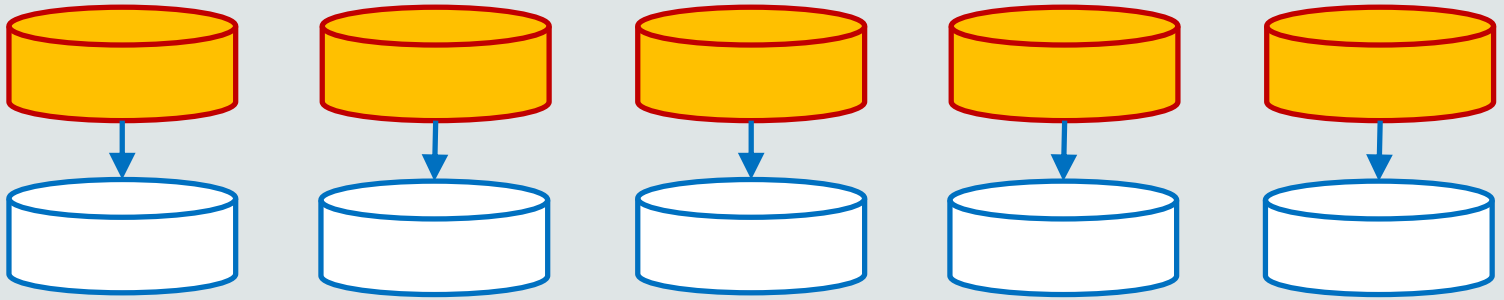
- Ausfall der Clusters, falls ein Server (und dessen Slots) nicht erreichbar
- *Fehlertoleranz* über zusätzliche Master-Slave Replikation



- Falls Netzwerkpartitionierung oder Ausfall von Servern, Cluster bleibt verfügbar solange
  - Mehrheit der Master noch funktioniert/erreichbar, und
  - Mind. ein Replikat pro nicht funktionierendem/erreichbarem Master erreichbar
- Replikate übernehmen die Rolle der Master
- Verfügbar nach TIMEOUT + Zeit für Wahl der neuen Master (1-2 Sek)
- Keine Verfügbarkeit bei größeren Netzwerkausfällen

Übung: arsnova.rz.uni-leipzig.de 81 60 01 90

Szenario: 5 Master-Server mit je 1 Replikat

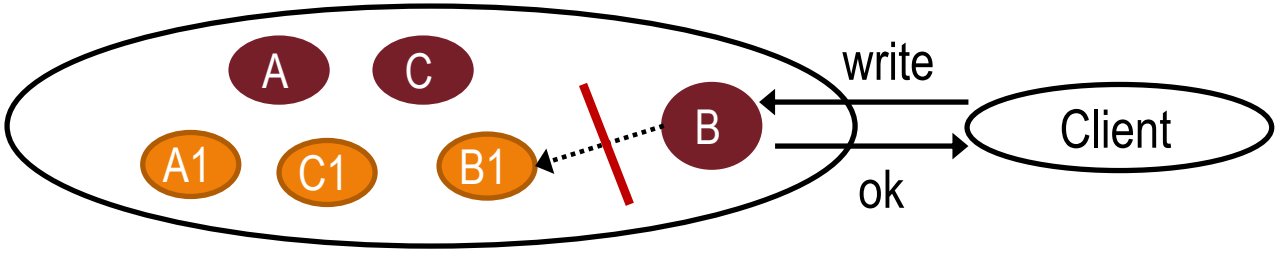


- Wie viele Server dürfen maximal ausfallen, um eine sichere Verfügbarkeit der *Majority Partition* zu gewährleisten?
  - A. 1
  - B. 2
  - C. 3
  - D. 4
- Wie hoch ist die Wahrscheinlichkeit, dass bei Ausfall eines weiteren Servers das System nicht mehr verfügbar ist?

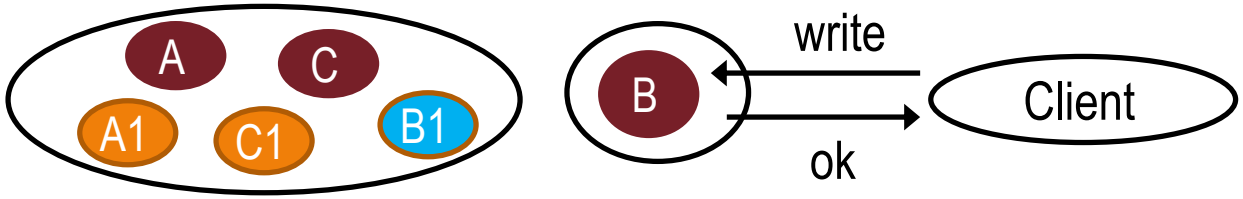
Redis Cluster: Konsistenz

- Asynchrone Replikation & „*Last Failover Wins*“ (Replikate werden von zuletzt gewähltem Master überschrieben)
- Keine *Konsistenzgarantien*
- Lost Updates sind selten aber möglich:

1. Aufgrund asynchroner Replikation und Absturz eines Master



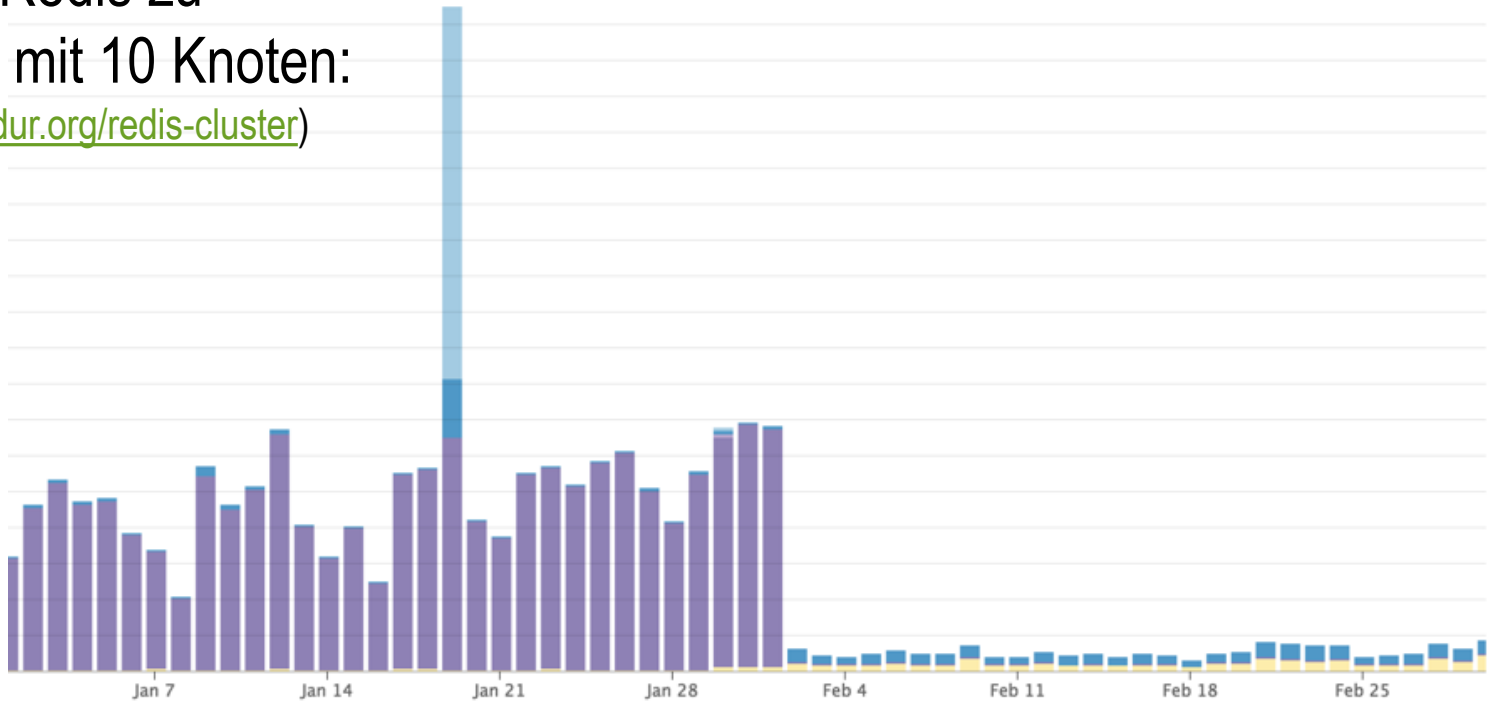
2. Aufgrund einer noch nicht erkannten Netzwerkpartitionierung (TIMEOUT)



Redis Cluster: Erfahrungsbericht

- Stripe: Online-Bezahldienst
- Redis ist Basis für “Rate Limiter”
- Hunderttausende Operationen pro Sekunde möglich

Fehlerrate vor und nach Wechsel von Redis zu Redis Cluster mit 10 Knoten:
(Quelle: https://brandur.org/redis-cluster)



Inhaltsverzeichnis: Key-Value Stores

Übung

- Schreiben Sie ein Programm, welches die Daten aus business2006.json in eine lokale Redis-Instanz schreibt. Verwenden Sie z.B. Jedis (https://github.com/xetorthio/jedis). Alternativen: https://redis.io/clients
  - Speichern Sie alle Einträge (JSON-String) unter Verwendung des Attributs business_id als Schlüssel (z.B. „business:tnhfDv5Il8EaGSXZGiuQGg“) ab.
  - Außerdem soll jeder Unternehmensname als Schlüssel auftauchen. Der zugehörige Wert ist eine Menge von business_ids, deren Unternehmen diesen Namen führen.

Schlüssel	Wert
business:S5RLqt9XkvZxoTMKXuF	‘{"business_id":"S5RLqt9XkvZxoTMK", "name": "Starbucks", "city":"Las Vegas", "stars":2.5, "categories": ["Coffee&Tea","Food"], …}’
business:UEpKf4g8oaAIr0ZIM0TXg	‘{"business_id":"UEpKf4g8oaAIr0ZIM0TXgw","name":"Starbucks", "city": "Cleveland", "stars":3.0, "categories": ["Food“, “Drinks”], …}’
…	…
Starbucks	{‘S5RLqt9XkvZxoTMKXuF-MA‘, ‘UEpKf4g8oaAIr0ZIM0TXg‘, …}
…	…

Übung

- Verwenden Sie Redis zur Implementierung einer *Autovervollständigung*
  - Schreiben Sie eine Funktion, die bei der Eingabe einer Zeichenkette alle Einträge, deren Anfang dieser Zeichenkette gleicht, heraussucht.
  - Zum Beispiel sollen bei der Eingabe von „Star" Unternehmensnamen wie „Starbucks" und „Star of India" gefunden werden.
  - Verwenden Sie z.B. den Redis-Befehl KEYS.
- Schreiben Sie eine Funktion, die, nach der Auswahl eines Namens, über die hinterlegten *business_ids* die dazugehörigen JSON-Strings abfragt.
  - Anschließend könnten weitere Attribute, z.B. alle Standorte der Unternehmen oder alle Kategorien, extrahiert werden.
  - Alternativ könnten die JSON-Strings zur Darstellung der Informationen zu HTML konvertiert werden.
- Sie könnten auch alle Attribute einzeln in Redis hinterlegen, z.B. über Schlüssel der Form „*business:tnhfDv5Il8EaGSXZGiuQGg:name".* Für mengenwertige Attribute sollten Sie die entsprechenden Datentypen von Redis verwenden (d.h. Hash, Set, List, Sorted Set).