

# Big-Data-Technologien

## **Kapitel 9: NoSQL – Key-Value-Stores**

Hochschule Trier  
Prof. Dr. Christoph Schmitz

# Key-Value-Stores

- **Schlüsseln (Keys)** werden **Werte (Values)** zugeordnet

kd123 → „Sabine Mayer“  
kd245 → „Peter Müller“  
kd654 → „Anna Schmidt“

- **Analogien:**
  - Maps (Dictionaries, assoziative Arrays)
  - Hash-Tabellen, Suchbäume
  - RDBMS: Relation mit Primärschlüssel

# Key-Value-Stores

- Werte können **strukturiert** sein:

kd123 → („Sabine Mayer“, w, 34)

kd245 → („Peter Müller“, m, 49)

kd654 → („Anna Schmidt“, w, 72)

- Notfalls (de)**serialisieren** in der Anwendung

# Key-Value-Stores

- Schlüssel können **strukturiert** sein:

(kd123, 1) → („Finkenweg 34“, „Konz“)

(kd123, 2) → („Packstation 114“, „Konz“)

... oder alternativ:

kd123 → [(„Finkenweg 34“, „Konz“),  
          („Packstation 114“, „Konz“)]

# Haben wir das nicht schon?

## Java

java.util.HashMap


- Serverdienst?
- Persistenz?
- Skalierbarkeit?
- Verteilung?
- Benutzerverwaltung?

## RDBMS

```
CREATE TABLE kv (  
  key CHAR(10)  
  PRIMARY KEY,  
  value VARCHAR(200)  
)
```

- Durchsatz?
- Latenz?
- Skalierbarkeit?
- Verteilung?

# Was bieten Key-Value-Stores?

- Hoher **Durchsatz** bei geringer **Latenz**
- **Persistenz**
- **Skalierbarkeit** durch Verteilung
- **Anfrage-Schnittstellen**
  - Netzwerkdienst
  - APIs für Programmiersprachen

# Key-Value-Store: redis

- „**Remote Dictionary Server**“
- Key-Value-Store im **Hauptspeicher**
- extrem leistungsfähig
- sehr schlank, implementiert in C
- single-threaded
- **Persistenz** über Snapshots und/oder Log
- Zahlreiche Operationen und **Datenstrukturen**
- Einfache textbasierte **Schnittstelle**

# Beispielsitzung

```
$ ./redis-cli
```

```
127.0.0.1:6379> GET kd123 (nil)
```

```
127.0.0.1:6379> SET kd123 "Sabine Mayer"  
OK
```

```
127.0.0.1:6379> SET kd245 "Peter Müller"  
OK
```

```
127.0.0.1:6379> GET kd123 "Sabine Mayer"
```

```
127.0.0.1:6379> GET kd245 "Peter M\xc3\xbc1ler"
```

```
127.0.0.1:6379> KEYS *  
1) "kd123"  
2) "kd245"
```

```
127.0.0.1:6379> GET kd456 (nil)
```



# Redis ist \*schnell\*

```
$ ./redis-benchmark
```

```
...
```

```
===== GET =====
```

```
100000 requests completed in 0.86 seconds
```

```
50 parallel clients
```

```
3 bytes payload
```

```
keep alive: 1
```

```
99.08% <= 1 milliseconds
```

```
99.64% <= 2 milliseconds
```

```
99.79% <= 3 milliseconds
```

```
99.83% <= 4 milliseconds
```

```
99.95% <= 6 milliseconds
```

```
100.00% <= 6 milliseconds
```

```
116959.06 requests per second
```



**... pro Kern!**

# Weitere Operationen

- **Datenstrukturen** für Werte
  - Listen (→ Stacks, FIFOs, ...)
  - (Sortierte) Mengen (→ Priority Queues)
  - Hashes
  - Geo-Koordinaten
  - Zähler
  - Bitvektoren
- Time-to-Live: automatisches **Herausaltern**
- **Multi-Operationen**
- **Iteratoren** für Schlüssel, Listen, Mengen, ...
- **Publish-Subscribe**



# Zähler

```
> GET counter          (nil)
> INCR counter          (integer) 1
> INCR counter          (integer) 2
> GET counter           "2"
> INCRBY counter 17     (integer) 19
```

# Listen-Operationen

```
> GET L (nil)
> RPUSH L 3 (integer) 1
> RPUSH L 5 (integer) 2
> RPUSH L 7 (integer) 3
> LLEN L (integer) 3
> LRANGE L 0 -1
1) "3"
2) "5"
3) "7"
> LINDEX L 1 "3"
> GET L (error) WRONGTYPE
> LPOP L "3"
> LPOP L "5"
> LLEN L (integer) 1
> LPOP L "7"
> LPOP L (nil)
```

**L → [3, 5, 7]**

# Mengen-Operationen

```
> SADD M 12      (integer) 1
> SADD M 18      (integer) 1
> SADD M 64      (integer) 1
> SCARD M        (integer) 3
> SADD M 17      (integer) 1
> SCARD M        (integer) 4
> SADD M 12      (integer) 0
> SCARD M        (integer) 4
> SSCAN M 0
```

```
1) "0"
2) 1) "12"
   2) "17"
   3) "18"
   4) "64"
```

M → {12, 17, 18, 64}

# Sortierte Mengen/Priority Queues

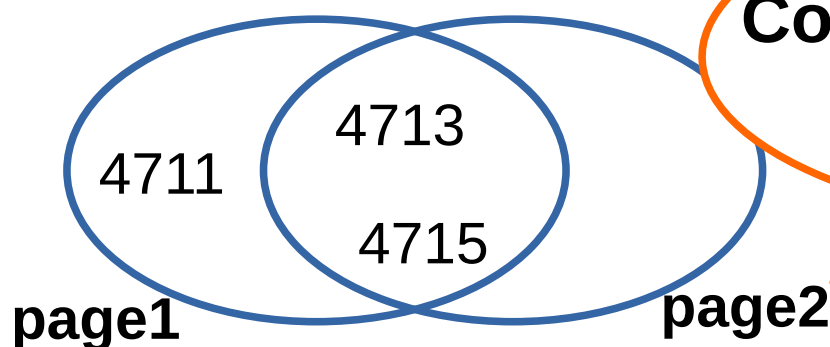
```
> ZINCRBY prioqueue 1 alice      "1"
> ZINCRBY prioqueue 3 bob        "3"
> ZINCRBY prioqueue 2 charlie    "2"
> ZINCRBY prioqueue 4 alice      "5"

> ZREVRANGE prioqueue 0 -1 WITHSCORES
    1) "alice"
    2) "5"
    3) "bob"
    4) "3"
    5) "charlie"
    6) "2"
```

# Bitvektoren

- Beispiel: Wie viele User besuchen sowohl **page1** als auch **page2**?

```
> BITFIELD page1 SET u1 4711 1 (integer) 0
> BITFIELD page1 SET u1 4715 1 (integer) 0
> BITFIELD page1 SET u1 4713 1 (integer) 0
> BITFIELD page2 SET u1 4713 1 (integer) 0
> BITFIELD page2 SET u1 4715 1 (integer) 0
> BITOP AND 1and2 page1 page2 (integer) 590
> BITCOUNT 1and2 (integer) 2
```



**Count-Distinct-Problem  
gelöst!**

# Publish-Subscribe

```
> subscribe sports politics  
Reading messages...
```

```
1) "subscribe"  
2) "sports"  
3) (integer) 1  
1) "subscribe"  
2) "politics"  
3) (integer) 2  
1) "message"  
2) "sports"  
3) "warriors-cavaliers 112:98"
```

```
> subscribe sports cinema  
Reading messages...
```

```
1) "subscribe"  
2) "sports"  
3) (integer) 1  
1) "subscribe"  
2) "cinema"  
3) (integer) 2  
1) "message"  
2) "sports"  
3) "warriors-cavaliers 112:98"
```

```
> publish sports "warriors-cavaliers 112:98"  
(integer) 1
```



# Time-To-Live

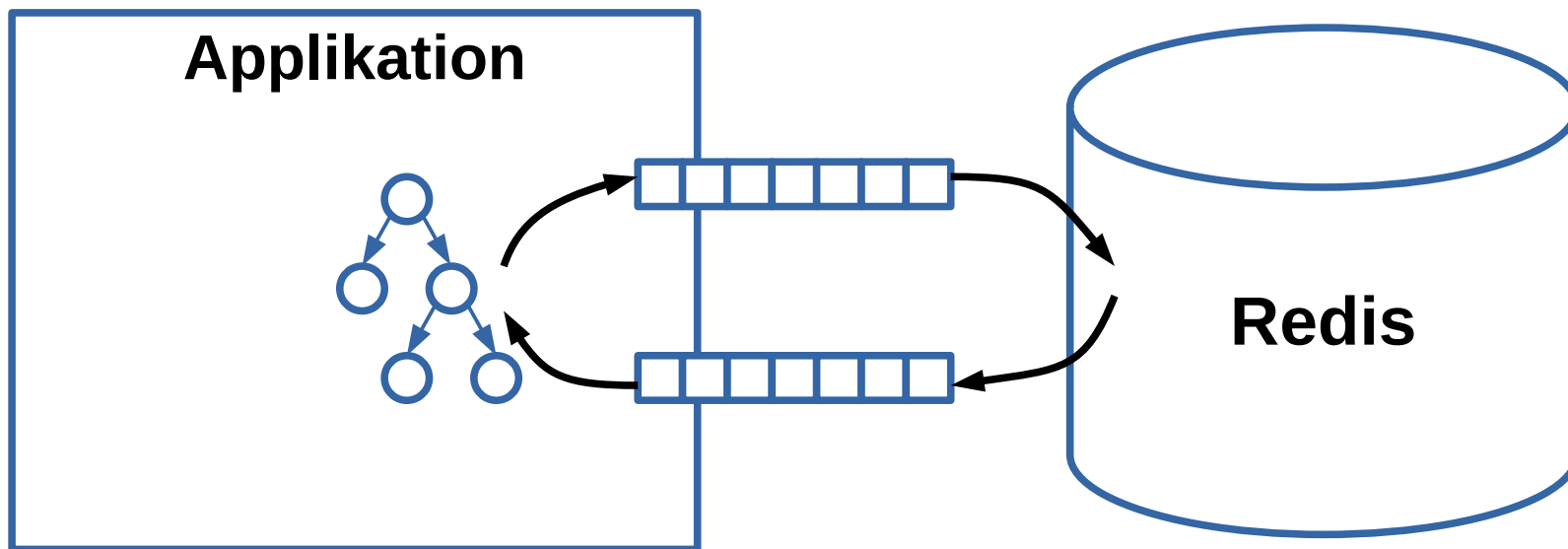
```
> SETEX name 10 wert    OK  
> GET name              "wert"
```

(... 10 Sekunden später...)

```
> GET name              (nil)
```

# Und wo ist der Haken?

- Strukturierte Datentypen sehr **speicherintensiv**
- Eventuell **applikationsseitig serialisieren**



# Schnittstellen: Python

```
import redis

r = redis.Redis("localhost", 6379)

r.set("foo", "bar")
print r.get("bla")
# None
print r.get("foo")
# bar

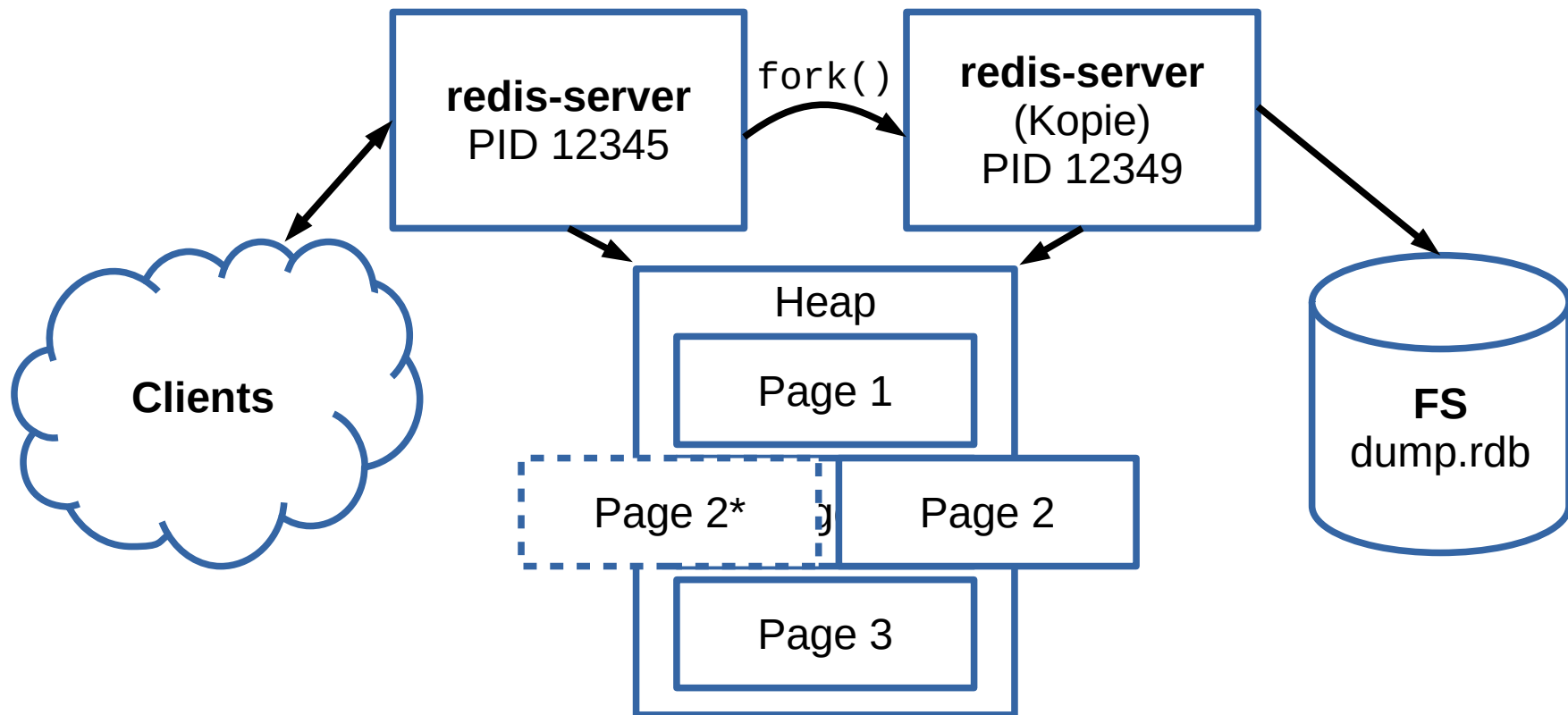
r.sadd("myset", "foo", "bar", "baz")
print r.sscan("myset")
# (0L, ['baz', 'bar', 'foo'])
```

# Schnittstellen: Java

```
try (Jedis jedis = new Jedis("localhost", 6379)) {  
    jedis.set("foo", "bar");  
  
    System.out.println(jedis.get("bla"));  
    // null  
    System.out.println(jedis.get("foo"));  
    // bar  
  
    jedis.sadd("myset", "foo", "bar", "baz");  
    System.out.println(  
        jedis.sscan("myset", "0").getResult());  
    // [baz, bar, foo]  
}
```

# Persistenz durch Snapshots

- Snapshots durch Unix-Funktion **fork**



# Persistenz durch Snapshots

```
$ ./redis-server
```

```
4662:M * The server is now ready to accept connections  
on port 6379
```

```
$ ./redis-cli
```

```
127.0.0.1:6379> BGSAVE
```

```
Background saving started
```

```
4662:M * Background saving started by pid 4693
```

```
4693:C * DB saved on disk
```

```
4693:C * RDB: 6 MB of memory used by copy-on-write
```

```
4662:M * Background saving terminated with success
```

# Persistenz durch Redo-Log („Append-Only File“)

- Protokolliert jede Operation auf Platte
- „Append-Only“ → nur **sequentielles Schreiben**

```
$ src/redis-cli  
> set kkk vvv  
OK
```

```
$ tail -f appendonly.aof  
*3  
$3  
set  
$3  
kkk  
$3  
vvv
```

- **Replay** beim Systemstart
- Kann periodisch **gekürzt** werden

# Scripting

- **Skripte** in der Datenbank (vgl. Stored Procedures)
- **Lua** 5.1 + Bibliotheken (JSON, struct/msgpack, bitop)
- Aufruf:

EVAL script numkeys key [key ...] arg [arg ...]  
                                    Schlüssel in                   beliebige  
                                    der Datenbank               Argumente

- Beispiel:

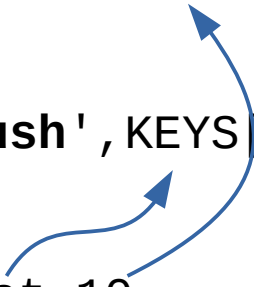
```
> EVAL "return 10 + 2" 0  
(integer) 12
```



# Scripting: Beispiel

EVAL script numkeys key [key ...] arg [arg ...]

```
> EVAL "local i = tonumber(ARGV[1]);  
  local res  
  while (i > 0) do  
    res = redis.call('lpush', KEYS[1], math.random())  
    i = i-1  
  end  
  return res" 1 random_list 10  
(integer) 10
```



```
> LRANGE random_list 0 -1  
1) "0.74509509873813717"  
2) "0.87390407681181281"  
...  
10) "0.1708280361121651"
```

# Script Cache

- Skripte **vorkompilieren**

```
> SCRIPT LOAD "return 10 + 2"  
"e62cdf fd393e1eed0202b8af004fb57ff17e2dd2"
```

- Kompiliertes Skript **ausführen**

```
> EVALSHA e62cdf fd393e1eed0202b8af004fb57ff17e2dd2 0  
(integer) 12
```

- **Weitere Operationen:** DEBUG, FLUSH, KILL, ...

# Transaktionen

- Zur Erinnerung: Transaktionen sind klassisch

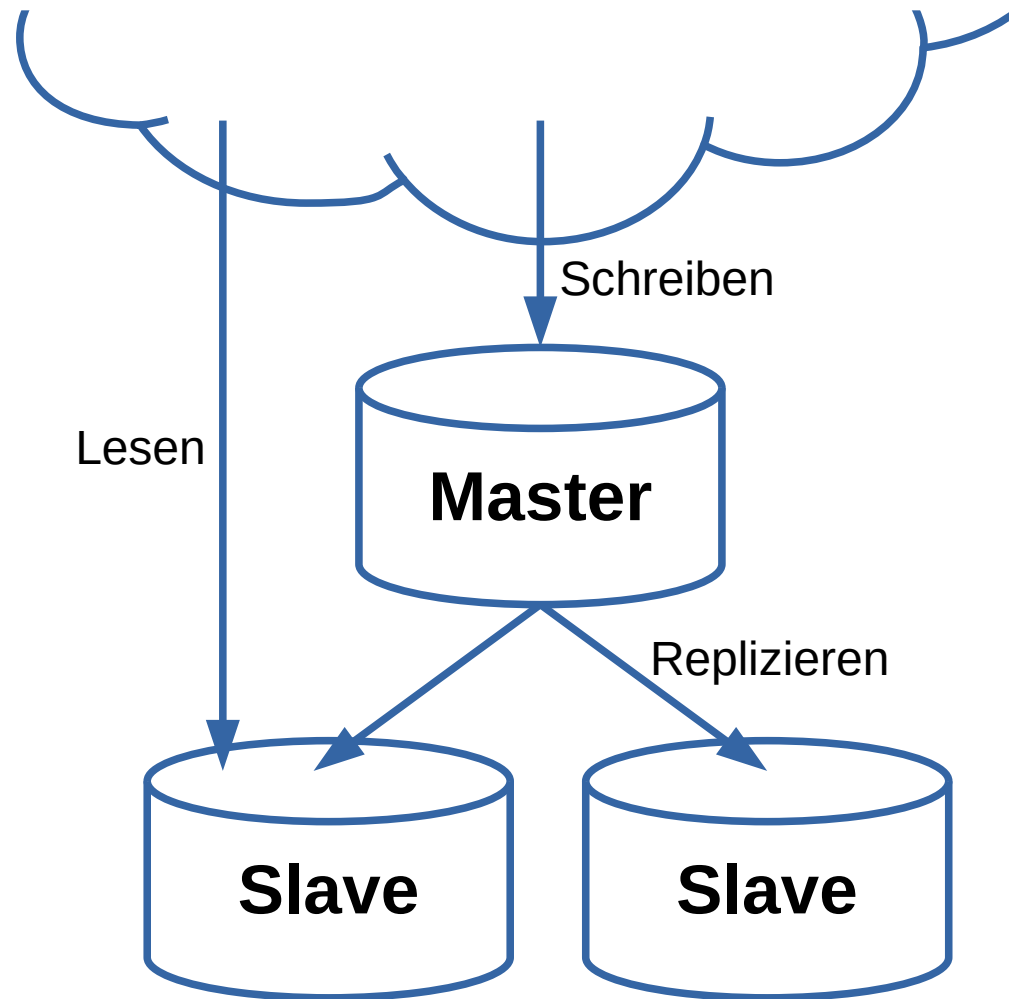
<b>A</b>	atomar
<b>C</b>	<del>konsistenzerhaltend</del>
<b>I</b>	isoliert
<b>D</b>	<del>dauerhaft</del>

- Hier also nur: **atomar, isoliert**
- Zur Erinnerung (II): Redis ist **single-threaded!**

# Transaktionen

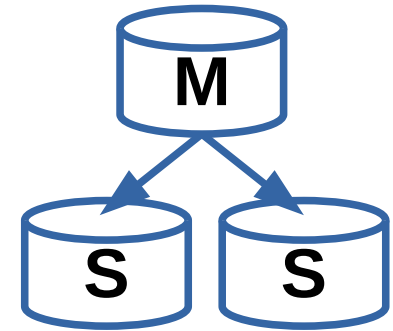
- Atomares **Zusammenfassen** mehrerer Kommandos
- **Isolieren** voneinander
- **Kein Rollback!**
  
- Variante 1: **MULTI** → Kommandos bündeln
- Variante 2: **EXEC** → Skripte

# Replikation (Master/Slave)



# Warum Replikation?

- **Master entlasten**
  - Lesen nur vom Slave
- **Verfügbarkeit**
  - Slave wird beim Ausfall „befördert“
- **Backup**
  - Master beantwortet Anfrage
  - Slave schreibt Sicherheitskopie



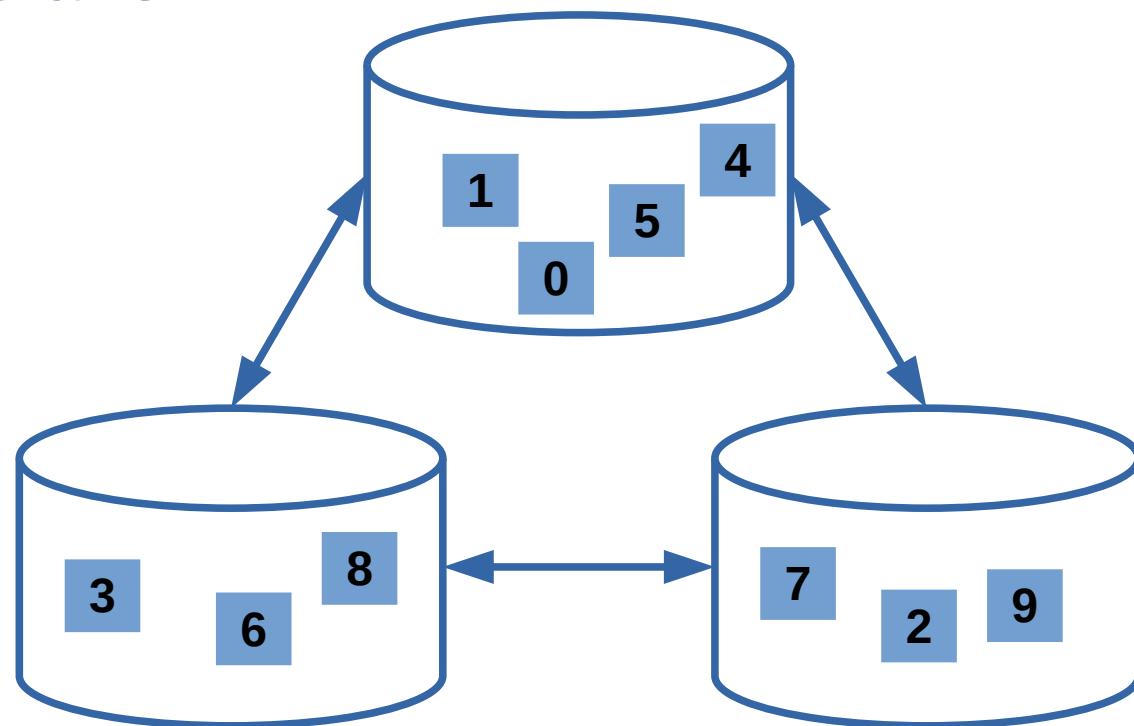
# Clustering

„a pragmatic  
approach to  
distribution“

- Grundidee: Sharding des Schlüsselraums

$\text{shard} = \text{hash}(\text{key}) \% n$

- „**Gossiping**“: ständiger Austausch von Statusinformationen
- Kombiniert mit **Master/Slave**  
→ Redundanz
- Mehrere Prozesse pro Knoten



# Clustering – Dummer Client

Client → node\_a: **GET foo**

node\_a → Client: -MOVED 8 node\_b:6379

Client → node\_b: **GET foo**

node\_b → Client: "bar"

„-MOVED 8 ...“:


- Key „foo“ ist in Shard 8
- Shard 8 liegt auf node\_b



# Clustering – Schlauer Client

> CLUSTER SLOTS

1) 1) (integer) 0  
2) (integer) 4095  
3) 1) "127.0.0.1"  
2) (integer) 7000  
4) 1) "127.0.0.1"  
2) (integer) 7004



Shard 1

2) ...

→ direkt den richtigen Knoten fragen




Shard 2

# Redis Cluster – Pro und Kontra

- ✓ **Schlanke** Lösung
- ✓ Unterstützung bei **Ausfällen**
- ✓ Unterstützung für **Resharding**
- ✗ Client sieht viele **Details**
- ✗ Keine starken **Garantien**
- ✗ Nicht alle **Kommandos** möglich

# Sicherheit?



„Redis is designed to be accessed by **trusted** clients inside **trusted** environments.“

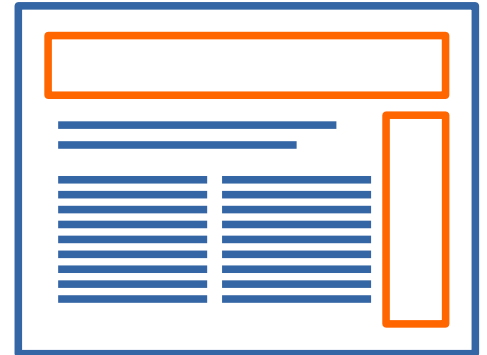
- **Perimeter Security**
  - Ports auf Netzwerkebene schützen
- Einfache **Authentifizierung** per Passwort
- **Abschalten** von Kommandos möglich (z. B. CONFIG)
- Keine **Autorisierung** über Rollen, Rechte, ...
- Keine **Verschlüsselung**

# Ist das Big Data?

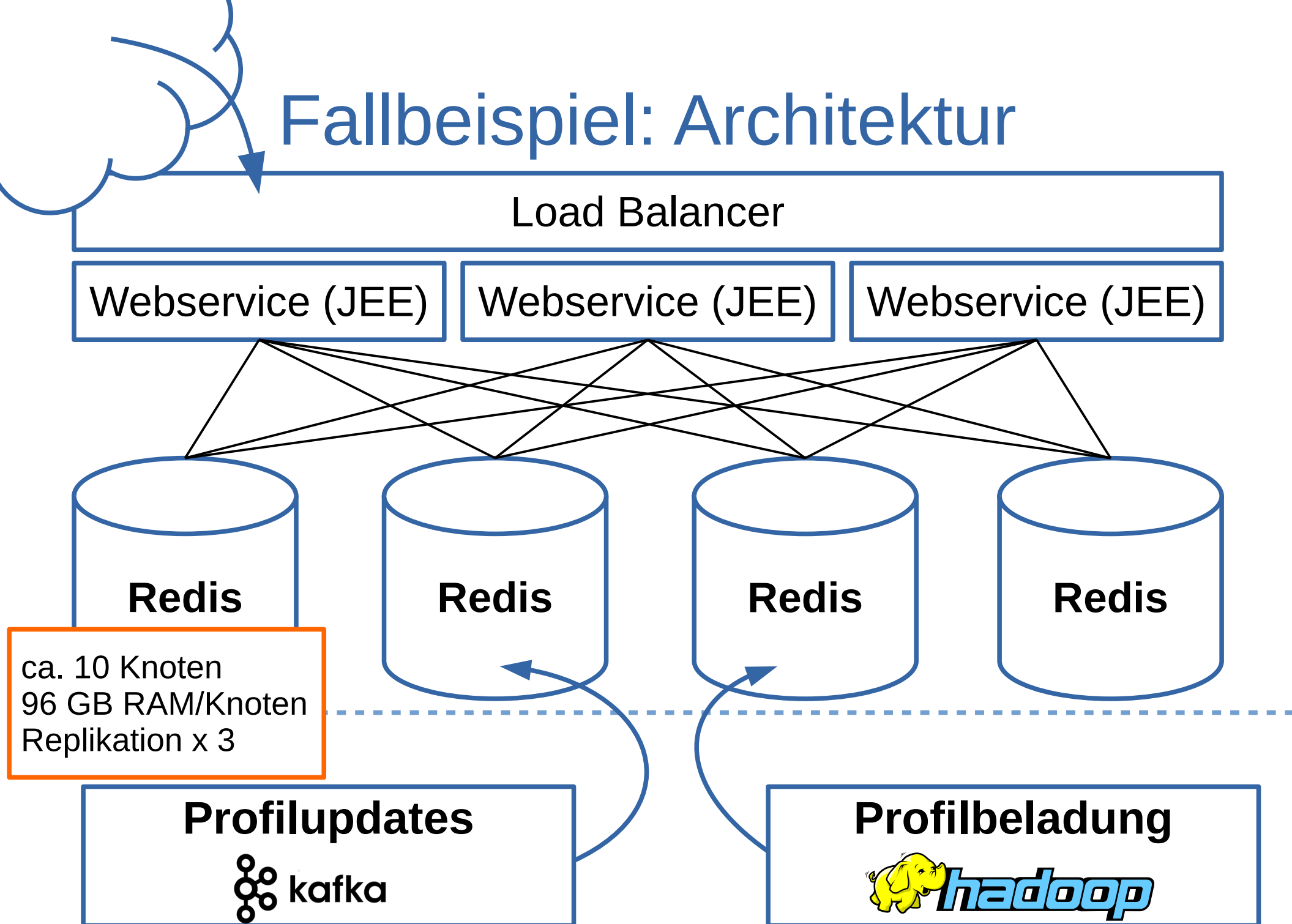
- Ist eine Hauptspeicherdatenbank **Big** Data?
- **Velocity** ✓
- **Variety** ✓
- **Volume** ?

# Ist das Big Data? Fallbeispiel

- Ausliefern von **Benutzer-Profilen** für Online-Werbung
- **Anforderungen**
  - 200 Mio. User-IDs
  - Profilgröße ca. 1 kB pro User
  - Latenz 1-2 ms
  - bis 40.000 Abrufe/s, 800 Mio. am Tag

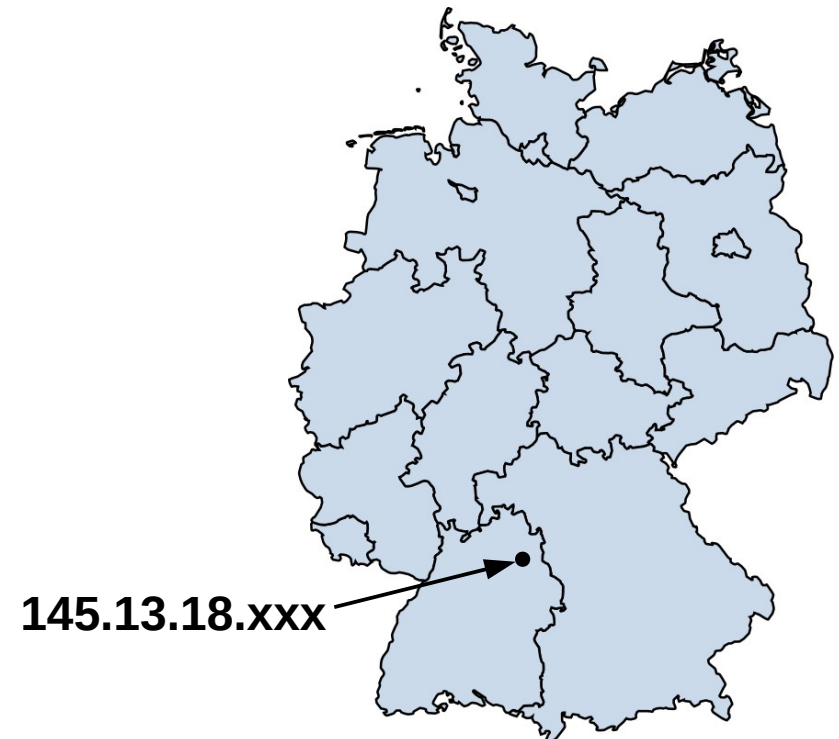


# Fallbeispiel: Architektur



# Weitere Anwendungsfälle

- **Aktuelle Informationen**
  - Wer ist gerade online?
  - die aktuellsten Nachrichten
  - die letzten  $n$  Tweets
- **Lookup-Tabellen**
  - Geo-IP-Auflösung
  - Auslieferungssysteme



# Weitere Anwendungsfälle

- **Caching**
  - Session Cache für Webapplikationen
  - Konfigurationsdatenbank
- **Queueing und Publish/Subscribe**
  - Message Queue
  - Börsenticker
  - Chat



# Weitere Anwendungsfälle

- **Zähler und Statistiken**

- Monitoring
- Web-Traffic

- **Echtzeit-Analysen**

- Fraud Detection
- Top-k-Probleme






# Fazit


- **Key-Value-Stores**
  - Zuordnung von Werten zu Schlüsseln
  - im Allgemeinen sehr schnell
  - strukturierte Werte
  - (strukturierte Schlüssel)
- **Anwendungsfälle**
  - Auslieferungsdienste
  - Transiente Daten (Zähler, Statistiken)
  - Unterstützende Dienste (Caching, Queueing)

# Fazit Redis

- **Hauptspeicherbasiert**
- Extrem **schnell**, geringe **Latenz**
- Reichhaltige **Operationen**
- Viele **Datenstrukturen** möglich
  - FIFO, Stack, Mengen, Hashes
  - Pub-Sub
  - Queues
- **Scripting** und Transaktionen
- **Verteilung** möglich
- Schwache **Sicherheitsmaßnahmen**

# Weitere Ressourcen

   | <https://redis.io/commands#> 67%  

 **redis** [Commands](#) [Clients](#) [Documentation](#) [Community](#) [Download](#) [Support](#) [License](#)

Filter by group: All or search for:

**APPEND** `key value`  
Append a value to a key

**AUTH** `password`  
Authenticate to the server

**BGREWRITEAOF**  
Asynchronously rewrite the append-only file

**BGSAVE**  
Asynchronously save the dataset to disk

**BITCOUNT** `key [start end]`  
Count set bits in a string

**BITFIELD** `key [GET type offset] [...]`  
Perform arbitrary bitfield integer operations on strings

**BITOP** `operation destkey key [key...]`  
Perform bitwise operations between strings

**BITPOS** `key bit [start] [end]`  
Find first bit set or clear in a string

**BLPOP** `key [key ...] timeout`  
Remove and get the first element in a list, or block until one is available