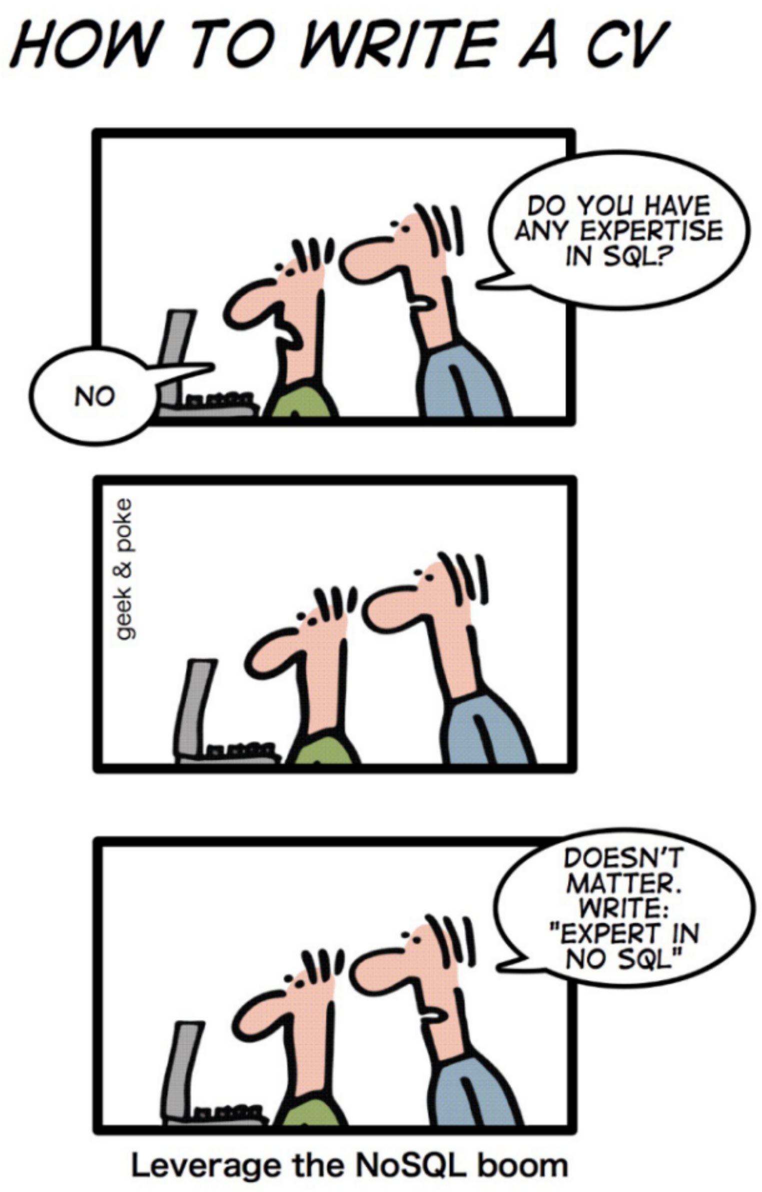
**Prof. Dr. Florian Heinz**
**florian.heinz@oth-regensburg.de**

# Modern Database Concepts

## Chapter 4: NoSQL Databases


HOW TO WRITE A CV
DO YOU HAVE ANY EXPERTISE IN SQL?
NO
geek & poke
DOESN'T MATTER. WRITE: "EXPERT IN NO SQL"
Leverage the NoSQL boom

Source: [https://geek-and-poke.com](https://geek-and-poke.com)

# Database History (in No-Notion ;-)

1970: NoSQL - We have no SQL

1980: NoSQL - Know SQL

2000: NoSQL - No SQL!

2005: **NoSQL - Not only SQL**

2013: No, SQL!

NoSQL stands for "Not only SQL".
Many benefits and features of NoSQL databases like schema flexibility and distribution is now also supported in modern relational database management systems, e.g. SQL/JSON.

# NoSQL Databases

Typical properties:

- distributed storage
- distributed computing
- ⇒ high (horizontal) scalability
- open-source / cost-effective
- schema-flexibility
- support for semi-structured and unstructured data
- non-relational data model

The list on the right shows the most popular NoSQL DBMS: MongoDB, Redis, Cassandra, DynamoDB, Neo4J, HBase. See https://db-engines.com/en/ranking.

# 4 Categories of NoSQL Databases

## Key-Value Stores

$$Key \rightarrow Value$$

## Wide-Column Stores

$$Table \rightarrow (RowID \rightarrow (Column \rightarrow Value))$$

## Document Databases

$$Database \rightarrow (Collection \rightarrow Document*)$$

## Graph Databases

$$G = (V, E)$$

The data model of key-value stores is the simplest one. They store a set of key-value pairs and allow a key-wise access. In wide-column stores, each row of a table can have arbitrary columns. Document DBs store a collection of documents (e.g., JSON). Graph databases store data within vertices and edges.

# Redis (Key-Value Store)

Data Model: Set of key-value pairs

Value Data Types:

- Strings
  "Hello World"
- Lists
  ["Hello World", "31337", "Hello World"]
- Hashes
  { "color":"red", "number":"1337" }
- Sets
  { "31337", "World", "Hello" }
- Sorted Sets
  { "31337", "Hello", "World" }
- (Bitmaps/HyperLogLogs, Streams, Geospatial Indexes)

Try out at https://try.redis.io/

# Redis (Key-Value Store)

## Strings

Stores a block of arbitrary (i.e. binary) data

```
> SET a 9
OK
> GET a
9
> INCR a
10
> SET b "Hello World"
OK
> MGET a b
1) "9"
2) "Hello World"
```

`help @string`

A String is the most basic scalar value type in Redis. It can be a text, binary data, an integer or a float

# Redis (Key-Value Store)

## Lists

Ordered lists of strings

```
> RPUSH seasons "summer"
1
> LPUSH seasons "spring"
2
> LRANGE seasons 0 -1
"spring"
"summer"
> RPOP seasons
"summer"
> BRPOPLPUSH seasons processed 1
"spring"
```

`help @list`

A Redis List is a composite data type that can store an ordered list of String values. Values can occur multiple times

# Redis (Key-Value Store)

## Sets (Unordered collection of strings)

```
> SADD pers:5:hobbies "piano"
1
> SADD pers:5:hobbies "piano"
0
> SADD pers:5:hobbies "reading"
1
> SCARD pers:5:hobbies
2
> SMEMBERS pers:5:hobbies
1) "reading"
2) "piano"
> SISMEMBER pers:5:hobbies "yoga"
0
```

`help @set`

A set is a composite data type and contains a number of values. These values have no particular order and each value can only occur once.

# Redis (Key-Value Store)

## Sorted Sets

Example: Leaderboard

```
> ZADD game 3100 king 5400 zorro 1340 mario
3
> ZREVRANGE game 0 -1
1) "zorro"
2) "king"
3) "mario"
> ZINCRBY game 10000 mario
"11340"
> ZREVRANK game mario
0
```

`help @sorted-set`

Sorted sets have a score value (float) associated with each member, according to which the elements are sorted. This score can be updated.

# Redis (Key-Value Store)

## Hashes

Mapping data type

```
> HSET pers:5 name john
1
> HSET pers:5 age 47
1
> HLEN pers:5
2
> HGET pers:5 name
"john"
> HINCRBY pers:5 age 2
49
```

`help @hash`

A Hash is another composite data type and can store key/value pairs. These pairs have no particular order, each key can only occur once.

# Redis (Key-Value Store)

## Geospatial data

Geospatial data is internally stored in a sorted set, where the score is calculated with a variation of the **Geohash** algorithm.

```
> GEOADD cities 12.1016236 49.0134297 Regensburg
> GEOADD cities 49.460983 11.061859 Nuernberg
> GEOADD cities 13.404954 52.520008 Berlin
> GEODIST cities Regensburg Berlin KM
"400.6375"
> GEORADIUS cities 11.0 49.0 70 km
1) "Nuernberg"
> GEORADIUSBYMEMBER cities Regensburg 200 KM
1) "Regensburg"
2) "Nuernberg"
```

# Redis (Key-Value Store)

## Geohash

Developed by G.M. Morton in 1966 and G.Niemeyer in 2008

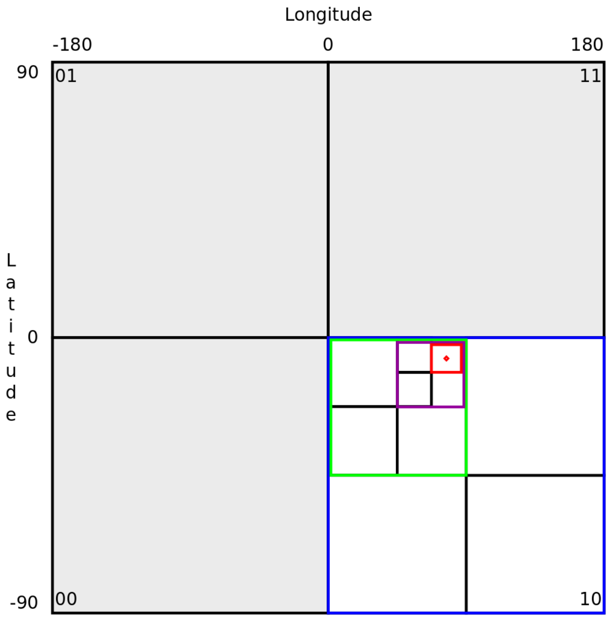
General idea: Recursively divide the world in quadrants

The red dot corresponds to the Geohash:

10 01 11 11 = $2133_4$

Number can be shortened to save space, but decrease accuracy of the location

For fast lookup: Calculate the ranges of the surrounding geohash boxes included in the radius.



# Redis Expiration

Keys can have an expiration time

```
> EXPIRE somekey 10
```

Useful for session data and other ephemeral values

Caching:

- Avoids reading stale data
- Avoids wasting memory with rarely used data

Variants:

```
> SETEX somekey 10 Hello
> GETEX somekey EX 10
```

# Redis Transactions

- Very basic transactional model
- Sequence of commands can be executed at once

  (i.e. other clients can not see a state in the middle of a transaction)
- Commands are just queued prior to execution
- No rollback on failed commands

```
> MULTI
> SET foo Hello
> SET bar 10
> INCR foo
> INCR bar
> EXEC
```

`INCR foo` throws an error, but the rest of the commands is executed anyway

# Redis Clustering

## Clustering support:

- General operation:
  - All cluster nodes are fully meshed
  - Gossip protocol for cluster communication

- Sharding (Partitioning)
  - Stores different keys at different nodes
  - Improves storage capacity and distributes query processing

- Replication
  - Creates a copy of a single Redis node
  - Provides high availability of the system

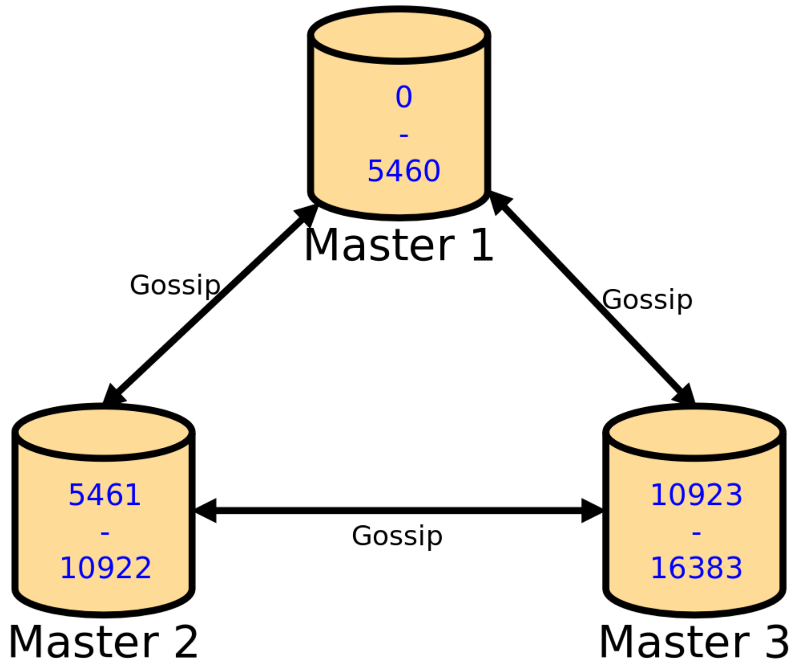
# Redis Clustering

## Sharding:

- 16384 key-slots
- Each key is mapped to one key slot
- Each key slot is served by a master node

## Topology change:

- Key-slots can be reassigned to other nodes
- Data (KV-pairs) is migrated to the new place
- Necessary when adding or removing cluster nodes



# Redis Clustering

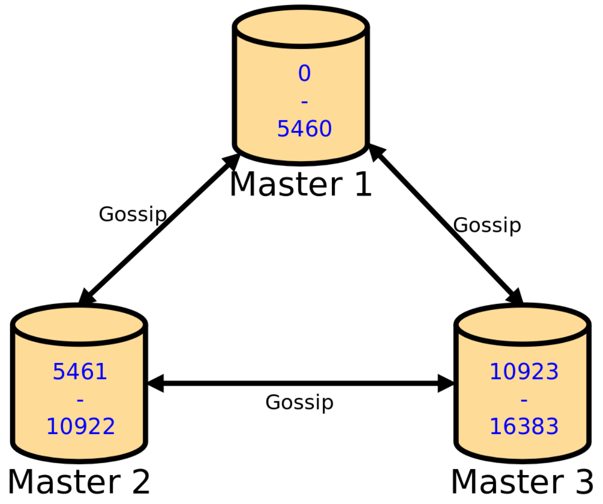
## Sharding:

When storing a key "foo" with value "bar", the client ...

- ... calculates `CRC16_XMODEM("foo") % 16384 = 12182`
- ... determines and connects to the correct shard
- ... executes the operation

Responsibility of the client to ask the correct node:

```
> SET foo bar
(error) MOVED 12182 127.0.0.1:30003
```



! Redis nodes do <u>not</u> proxy requests for the client; the client has to honour redirect messages from the server or fetch a cluster map regularly.
Command: `CLUSTER NODES`

# Redis Clustering

## Sharding and Multi-Key operations:

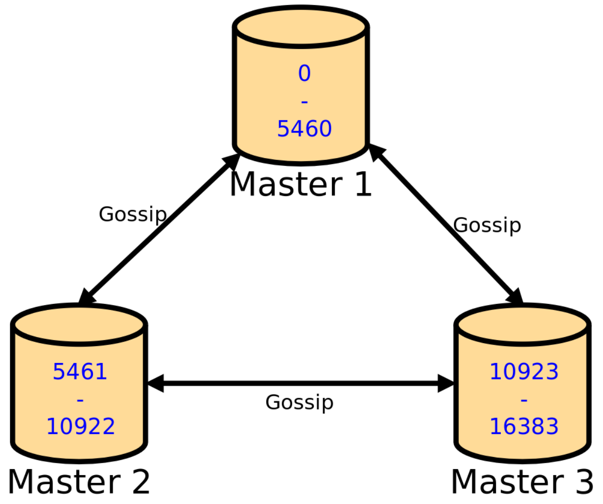
Keys for multikey operations must hash to the same slot (same node is insufficient!)

```
> MSET john:age 35 john:favcolor red
(error) CROSSSLOT Keys in request don't
hash to the same slot
```

**Why:** Otherwise successful execution of a command would depend on cluster topology
**Solution:** hashtags

```
> MSET {john}:age 35 {john}:favcolor red
OK
```

! Only the part between the first pair of curly brackets is used for hash calculation.

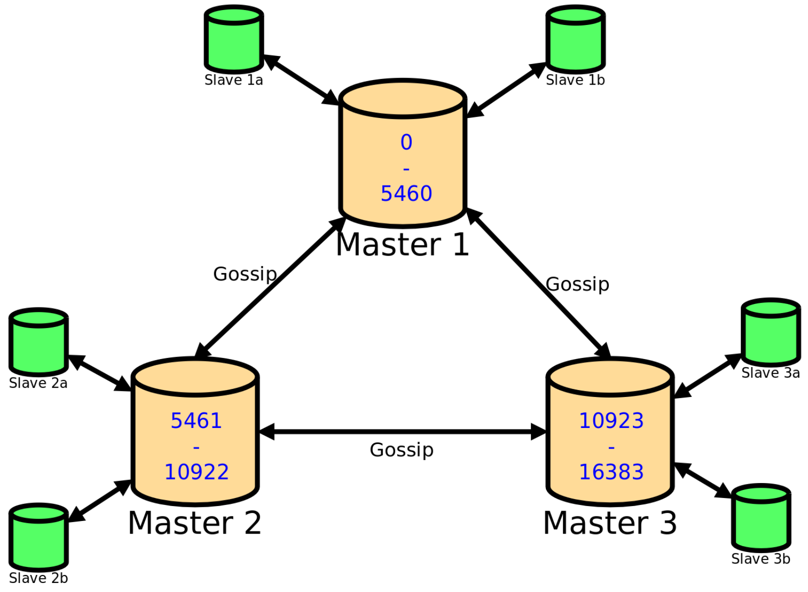


# Redis Clustering

## Replication:

Each Redis master node can have one or more associated slave nodes

- Exact copy of the data (same hash slots)
- Fully meshed with all other nodes
- Automatic failover on failure
- Asynchronous replication



Read-only queries possible:

```
> GET foo
(error) MOVED 12182 127.0.0.1:30003
> READONLY
OK
> GET foo
"bar"
```

# Redis Scripting

Server-side scripting with LUA 5.1

Advantages:

- Complex operations built from basic redis functions
- Lower latency for chained operations

```
> EVAL "return 'Hello World'" 0
"Hello World"
> EVAL "return 13 * 37" 0
(integer) 481
```

Lua:

- Minimalistic language
- Easy to implement and to embed
- Object oriented programming supported

See https://www.lua.org/manual/5.1/

## Redis Scripting

```
EVAL <script> <nrkeys> [key1 ...] [arg1 ...]
```

```
EVAL "return 'Hello ' .. ARGV[1]" 0 World
"Hello World"
EVAL "return ARGV[1] * ARGV[2]" 0 13 37
(integer) 481
```

With key name arguments:

```
EVAL "return 'Key:'..KEYS[1]..' Arg:'..ARGV[1]" 1 mykey World
"Key:mykey Arg:World"
```

Key name arguments are used for keys, that are fetched or stored in a LUA script

# Redis Scripting

Manipulating keys in LUA scripts

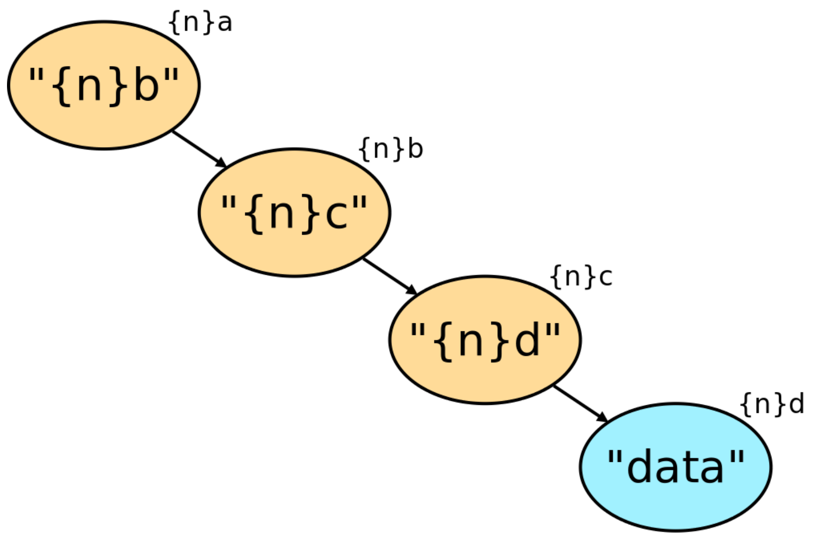
```lua
local nr1 = redis.call('GET', '{nr}1')
local sum = nr1 + redis.call('GET', '{nr}2')
redis.call('SET', '{nr}3', sum)
return "Sum is " .. sum
```

Execution cached and with key arguments:

```
> SCRIPT LOAD
   "local sum=redis.call('GET', KEYS[1])+redis.call('GET', KEYS[2])
    redis.call('SET', KEYS[3], sum)
    return 'Sum is '..sum"
"2bdc424aeb7f41d6e4f6084f0f3fb7616ec1bd19"
> EVALSHA "2bdc424aeb7f41d6e4f6084f0f3fb7616ec1bd19" 3 {nr}1 {nr}2 {nr}3
"Sum is 35"
```

# Redis Scripting

Loops and Latency:



```
> MSET {n}a {n}b  {n}b {n}c  {n}c {n}d  {n}d data
OK
> EVAL "local key=KEYS[1]
        for i=1,4 do key=redis.call('GET',key) end
        return key" 1 {n}a
"data"
```

## Redis Integration

Bindings for all popular programming languages

Many also have a cluster-aware implementation

```
>>> import redis
>>> c = redis.cluster.RedisCluster(host="localhost",port=30001)
>>> c.set("Hello", "World")
True
>>> c.get("Hello")
b'World'
>>> c.lpush("terms", "foo")
1
>>> c.lpush("terms", "bar")
2
>>> c.lpush("terms", "baz")
3
>>> c.lrange("terms", 0, -1)
[b'baz', b'bar', b'foo']
```