



**FAKULTA ELEKTROTECHNICKÁ**

České vysoké učení technické v Praze

# B4M36DS2 – Database Systems 2

## Lecture 4 – **Key-Value stores**: Redis

14. 10. 2024

**Yuliia Prokop**

[prokoyul@fel.cvut.cz](mailto:prokoyul@fel.cvut.cz), Telegram **@Yulia\_Prokop**



CourseWare Wiki

<https://cw.fel.cvut.cz/wiki/courses/b4m36ds2/>

## Key-value stores

- Introduction

## Redis

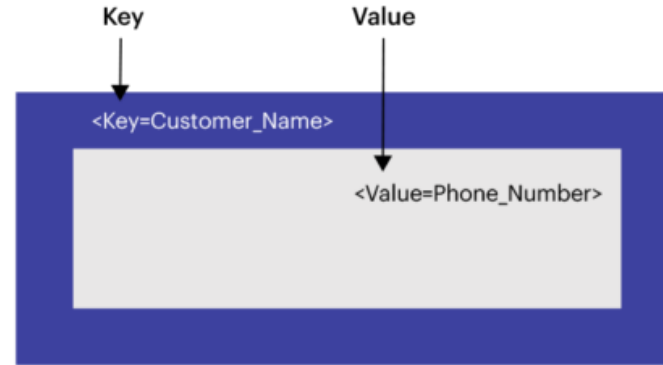
- Data model
- Keys
- Redis Data Types
- Basic commands and operations
- Examples
- NoSQL principles
- Redis architecture

# Key-value databases

# Key-Value Stores

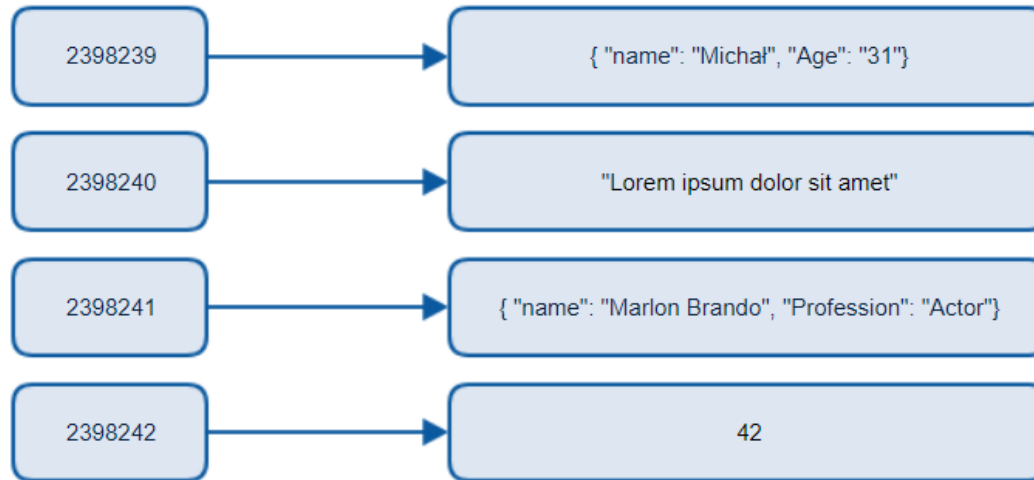
Phone directory

Key	Value
Paul	(091) 9786453778
Greg	(091) 9686154559
Marco	(091) 9868564334



Keys

Values



Source: <https://redis.com>, <https://www.michalbialecki.com>

# Key-Value Stores

## Data model

- The most simple NoSQL database type
  - Works as a simple hash table (mapping)
- **Key-value pairs**
  - **Key** (id, identifier, primary key)
  - **Value:** binary object, black box for the database system
    - ✓ The value can be any sequence of bytes. It could be strings, numbers, serialized objects, images, or any other data in binary format.
    - ✓ Modern systems often provide richer capabilities for working with structured data, while maintaining the simplicity and efficiency of the key-value mode

## Query patterns

- Create, update or remove value for a given key
- **Get value** for a given key

## Characteristics

- Simple model ⇒ **great performance, easily scaled, ...**
- Simple model ⇒ **not for complex queries nor complex data**

# Key-Value Stores

- **Key-value** storage systems store large numbers (billions or even more) of small (KB-MB) sized records.
- Records are **partitioned** across multiple machines, with the system intelligently routing queries to the appropriate node.
- Records are also **replicated** across multiple machines to ensure availability and fault tolerance
- **Consistency** mechanisms ensure that updates are propagated to all replicas, maintaining data integrity across the distributed system.
- These systems offer various **consistency models**, balancing between performance and data consistency needs.

How the keys should actually be designed?

- **Real-world** identifiers
  - E.g. e-mail addresses, login names, ...
- **Automatically generated** values
  - Auto-increment integers
    - Not suitable in peer-to-peer architectures!
  - Complex keys
    - Multiple components / combinations of time stamps, cluster node identifiers, ...
    - Used in practice instead

**Prefixes** describing entity types are often used as well

- E.g. **movie**\_medvidek, **movie**\_223123, ...

**Composite keys:** combining multiple attributes to form a key:

- E.g. **user:1234:profile**

## Basic **CRUD** operations

- Only when a key is provided
- $\Rightarrow$  knowledge of the keys is essential

Modern key-value systems like Redis offer efficient methods such as the SCAN command to list available keys without blocking the server, though this operation may still be resource-intensive for very large databases.

Modern key-value systems allow comprehensive access to value contents

- But we could instruct the database how to **parse the values**
- ... so that we can **index** them based on certain **search criteria**

Batch / sequential processing

## Expiration of key-value pairs

- Objects are **automatically removed** from the database **after a certain interval of time**
- Useful for user sessions, shopping carts etc.

## Links between key-value pairs

- Values can be mutually interconnected via links
- These links can be traversed when querying

## Collections of values

- Not only ordinary values can be stored, but also their collections (e.g. **ordered lists, unordered sets, ...**)

*Particular functionality always depends on the store we use!*

# When to use a key-value database

- Handling Large Volume of Small and Continuous Reads and Writes
- Storing Both Basic and Complex Data Structures
- Applications with Frequent Updates and Range from Simple to Moderately Complex Queries
- When your application needs to handle lots of small continuous reads and writes, that may be volatile

## Use cases

- Session Management on a Large Scale
- Using Cache to Accelerate Application Responses
- Storing Personal Data on Specific Users
- Product Recommendations and Personalized Lists
- Managing Player Sessions in Massive Multiplayer Online Games

# Key-value database use cases

- **Data Caching**
  - Scenario: Web application with frequent database queries.
  - Application: Query results are stored in a key-value store (e.g., Redis).
  - Advantage: Significant speedup for repetitive queries.
  - Example: Caching search results, user profiles, and popular articles.
- **User Session Management**
  - Scenario: Web application with user authentication.
  - Application: Session information (user ID, tokens, last activity time) is stored in a key-value system.
  - Advantage: Fast access to session data and easy scalability.
  - Example: Storing JWT tokens, a shopping cart on an e-commerce site.
- **Storing Settings and Configurations**
  - Scenario: Application with user preferences or dynamic configuration.
  - Application: Settings are stored as key-value pairs.
  - Advantage: Quick reading and updating of settings without complex queries.
  - Example: User preferences in an app, interface theme settings.

# Key-value database use cases

- **Message and Task Queues**

- Scenario: System with asynchronous task processing.
- Application: Tasks are added to a queue and retrieved for processing.
- Advantage: Efficient load distribution and scalability.
- Example: Email sending queue, processing user-uploaded files.

- **Counters and Statistics**

- Scenario: Need to track various metrics in real time.
- Application: Using atomic increment/decrement operations.
- Advantage: High performance for frequent updates.
- Example: Counting page views, likes, and number of online users.

- **Temporary Data Storage**

- Scenario: Data needed only for a short period.
- Application: Storage with Time-To-Live (TTL) setting.
- Advantage: Automatic deletion of outdated data.
- Example: Confirmation codes for two-factor authentication, temporary tokens.

- **Distributed Locks**

- Scenario: Need for synchronization in a distributed system.
- Application: Using atomic operations to create and remove locks.
- Advantage: Preventing conflicts in parallel access.
- Example: Resource locking in a microservices architecture.

- **Computation Result Caching**

- Scenario: Application with resource-intensive computations.
- Application: Saving computation results for reuse.
- Advantage: Significant speedup for repeated requests.
- Example: Caching results of complex SQL queries, web page rendering.

- **IoT Device Data Storage**

- Scenario: Collecting and processing data from multiple IoT devices.
- Application: Fast writing and reading of device data.
- Advantage: High-speed processing of a large number of small records.
- Example: Storing sensor readings and device statuses.

- **Geospatial Indexes**

- Scenario: Applications with geolocation features.
- Application: Storing object coordinates for quick search.
- Advantage: Efficient search for nearest objects.
- Example: Finding nearby restaurants, tracking moving objects.

- **Rate Limiting Management**

- Scenario: Need to limit API request frequency.
- Application: Tracking the number of requests from a user/IP in a given time interval.
- Advantage: Quick checking and updating of limits.
- Example: Limiting the number of API requests to prevent overload.

- **Large Object Storage**

- Scenario: Need to store and quickly retrieve large data objects.
- Application: Storing objects with a unique key.
- Advantage: Fast access to large objects without complex database structure.
- Example: Storing images, documents, and audio files.

# Limitations of Key-Value Stores

- Limited data structure: optimized only for simple key-value pairs.
- Lack of value-based filtering: inability to query based on value content.
- Inefficiency in collection scanning: no optimization for complete data traversal.
- Key-value stores are not compatible with SQL.
- Limited transaction support: while some systems offer atomic operations, they often lack full ACID-compliant transaction capabilities, including automatic rollbacks.
- There is no standard query language as opposed to SQL.

# Redis

## REDIS (Remote Dictionary Server)

- **In-memory data structure store**

Can be used as Database, Cache, Message broker, and Streaming engine

- Open-source software: <https://redis.io/>

- Developed by **Redis Labs**

- Implemented in **C**

- First released in 2009

- Redis is really fast: 100,000+ read/writes per second

# Redis: unique characteristics

- **High performance:**
  - In-memory operations ensure very low latency.
- **Flexible data model:**
  - Redis supports various data structures (strings, hashes, lists, sets, etc.).
- **Persistence:**
  - Despite operating in-memory, Redis offers mechanisms for saving data to disk.

# WHY Redis? Who uses Redis?

- Very flexible
- Very fast
- No schemas, column names
- Rich Datatype Support
- Caching & Disk persistence



- Standard **key-value store**
- Support for **structured values** (e.g. lists, sets, ...)
- **Time-to-live**
- Transactions
- Cluster support for horizontal scaling
- Pub/Sub (publish/subscribe system) and Streams for processing streaming data
- Lua scripting for executing complex operations
- Real-world users:  
Twitter, GitHub, Pinterest, StackOverflow, Instagram, Snapchat, Airbnb, Uber,  
and many others

Instance → **databases** → **objects**

- **Database** = collection of objects
  - Databases do not have names but integer identifiers
- **Object** = **key-value pair**
  - Key is a **string** (i.e. any binary data)
    - ✓ The maximum key size is 512MB
  - Values can be...
    - Atomic: **string**
    - Structured: **list, set, sorted set, hash, bitmaps, hyperloglog, streams...**

- **Keys** are binary safe – it is possible to use any binary sequence as a key
- The empty string "" is also a valid key
  - ✓ Use with caution to avoid confusion
- Key length considerations
  - ✓ Too long: Can consume more memory and increase lookup time
  - ✓ Too short: May lack clarity and increase chances of key collisions
- Key naming best practices
  - ✓ Use descriptive names  
(e.g., "**user:1000:password**" instead of "**u:1000:pwd**")
  - ✓ Improves readability and maintainability
- Recommended schema: "**object-type:id:field**" (Use consistent separators!) :
  - ✓ Example: "user:1000:email"

# Redis keys commands

- **SET** key value [EX seconds]  
Sets the string value of a key.  
Optional EX parameter sets an expiration time in seconds
- **GET** key  
Returns the string value of a key
- **EXISTS** key [key ...]  
Checks if one or more keys exist  
Can check multiple keys in one command
- **TYPE** [key]  
Returns the data type of the value stored at the key
- **DEL** key [key ...]  
Deletes one or more keys  
Can delete multiple keys in one command

# Keys – Examples

**SET** a hello  
OK

**GET** a  
"hello"

**RENAME** a ahoj  
OK

**GET** a  
(nil)

**EXISTS** a  
(integer) 0

**GET** ahoj  
"hello"

**TYPE** ahoj  
String

**SET** x 120  
OK

**TYPE** x  
String

**DEL** x  
(integer) 1

**EXISTS** x  
(integer) 0

## Keys with limited time to live

- When a specified timeout elapses, a given object is automatically removed
  - Works with any Redis data type
- 
- ✓ Temporary lockdowns
  - ✓ Temporary subscriptions
  - ✓ Burnable bonuses
  - ✓ SMS timeout (not more often than in a minute)
  - ✓ Verification by code (within 1 minute).

- **Set:**
  - **EXPIRE** key seconds
    - ✓ Sets a timeout for a given object, making it volatile.
    - ✓ Can be called repeatedly to change the timeout.
  - **EXPIRE AT** key timestamp
    - ✓ Sets an absolute Unix timestamp for expiration
  - **PEXPIRE** key milliseconds
    - ✓ Sets timeout in milliseconds
  - **PEXPIREAT** key milliseconds-timestamp
    - ✓ Sets expiration at a specific millisecond timestamp
  - **SET** key value [EX seconds | PX milliseconds]
    - ✓ Sets a value and expiration in one command (addition)

- **Examine:**
  - **TTL key**
    - ✓ Returns the remaining time to live for a key that has a timeout (in seconds)
  - **PTTL key**
    - ✓ Returns the remaining time to live for a key that has a timeout (in milliseconds)
- **Remove:**
  - **PERSIST key**
    - ✓ Removes the existing timeout, i.e. makes the object persistent
- **Additional notes**
  - Redis uses the LFU (Least Frequently Used) eviction policy
    - ✓ When the cache is full, it removes the least frequently used items first
  - **Lazy freeing** of large objects is supported for better performance
    - ✓ Marks large objects for deletion
    - ✓ Actual memory freeing happens in a background process
    - ✓ Reduces latency for commands that delete large objects

# Volatile Objects: passive and active expiration processes

- **Passive expiration:**
  - Triggered when a key with an expiration time is accessed
  - Redis checks if the key has expired upon access
  - If expired, the key is deleted before processing the command
- **Active expiration:**
  - Redis uses a probabilistic algorithm to expire keys actively.
  - The algorithm runs 10 times per second (every 100ms).
  - In each run, Redis does the following:
    - ✓ Samples 20 random keys with expiration times
    - ✓ Deletes all sampled keys that have expired
    - ✓ If more than 25% of sampled keys expired, repeats the process

# Key eviction

- **Purpose:**
  - Free up memory when Redis reaches maximum memory limit
  - Allow new data to be added by removing less essential or outdated data
- **Eviction policies:**
  - **noeviction:** No keys evicted – **when data retention is critical**
  - **allkeys-lru:** Least Recently Used keys
  - **volatile-lru:** LRU among keys with expiration
  - **allkeys-random:** Random keys – **good for caching**
  - **volatile-random:** Random among keys with expiration
  - **volatile-ttl:** Shortest time-to-live keys - **suited for preserving keys with longer TTL**
  - **allkeys-lfu:** Least Frequently Used keys
  - **volatile-lfu:** LFU among keys with expiration

# Redis: Volatile Objects examples

**SET** counter 100  
OK

**GET** counter  
"100"

**EXPIRE** counter 10  
(integer) 1

**EXISTS** counter  
(integer) 1

**EXISTS** counter  
(integer) 0

**SET key 100 EX 10**  
OK

**TTL key**  
(integer) 2

# Data Types

- Strings
- Lists
- Sets
- Sorted sets
- Hashes
- Bitmaps
- Hyperlogs
- Geospatial indexes

**Datatypes cannot be nested!**  
**(no lists of hashes)**

Further information: <https://redis.io/docs/data-types/>

hello world	String
011011010110111101101101	Bitmap
{23334}{6634728}{916}	Bitfield
{a: "hello", b: "world"}	Hash
[A>B>C>C]	List
{A<B<C}	Set
{A:1, B:2, C:3}	Sorted set
{A: (50.1, 0,5)}	Geospatial
01101101 01101111 01101101	Hyperlog
{id1=time1.seq(( a: "foo", a: "bar"))}	Stream

Source: <https://architecturenotes.co/redis/>

- **String**
  - The only **atomic data type**
  - May contain any binary data  
(e.g. string, integer counter, PNG image, ...)
  - Maximal allowed size is 512 MB
  - Use cases:
    - ✓ Store JPEG's
    - ✓ STORE serialized objects

hello world String

# Data Types : Lists

- **List** – a double-linked list of Strings
  - Max size 4294967295 ( $2^{32} - 1$ )
- Can be used for
  - F. ex. timelines of social networks
- Speed
  - Actions at the start or end of the list are very fast
  - Actions in the middle are a little less fast
- Lists support blocking operations (e.g., BLPOP), making them convenient for implementing queues

[A>B>C>C] List

- **Set**

- **Unordered collection of Strings**

- ✓ Each time you retrieve or view the elements of a set, their order may be different

- Duplicate values are not allowed

- Useful for tracking unique items

- Allow extracting random members

Using SPOP, SRANDMEMBER

- Useful for intersect and diffs

- Operation efficiency: set allows for adding, removing, and checking for the presence of an element with  $O(1)$  time complexity



{A<B<C}

Set

- **Sorted set**
  - **Ordered collection of strings**
  - The order is given by a score (floating number value) associated with each element (from the smallest to the greatest score)
  - Members are unique. Scores are not
  - Ordering for items with the same score is alphabetic
  - Useful for leader boards or autocomplete
  - Maximum number of members in a Redis sorted set is 4,294,967,295 ( $2^{32} - 1$ )

`{A:1, B:2, C:3}` Sorted set

# Data Types: Sorted Sets and Hashes

- **Hash**

- **Associative map between string fields and string values**
- Ideal for storing objects
- Field names have to be mutually distinct
- Support operations like retrieving specific fields (HGET), multiple fields (HMGET), or all fields (HGETALL)

{a: "hello", b: "world"}

Hash

## Basic commands

- **SET** *key value*  
Inserts / replaces a given string
- **GET** *key*  
Gets a given string

## Counter operations

- **INCR** *key*
- **DECR** *key*  
Increments/decrements a value by 1
- **INCRBY** *key increment*
- **DECRBY** *key decrement*  
Increments/decrements a value by a given amount

# String operations

- **STRLEN** *key*

Returns a string length

- **APPEND** *key value*

Appends a value at the end of a string

- **GETRANGE** *key start end*

Returns a substring

- Both boundaries are considered to be inclusive
- Positions start at 0
- Negative offsets for positions starting at the end

- **SETRANGE** *key offset value*

Replaces a substring

- Binary 0 is padded when the original string is not long enough

# String examples

**SET** mykey somevalue  
OK

**GET** mykey  
"somevalue"

**STRLEN** mykey  
(integer) 9

**APPEND** mykey 666  
(integer) 12

**GET** mykey  
"somevalue666"

**GETRANGE** mykey 2 8  
"mevalue"

**SETRANGE** mykey 5 xxxx  
(integer) 12

**GET** mykey  
"somevxxxx666"

**INCR** counter  
(integer) 101

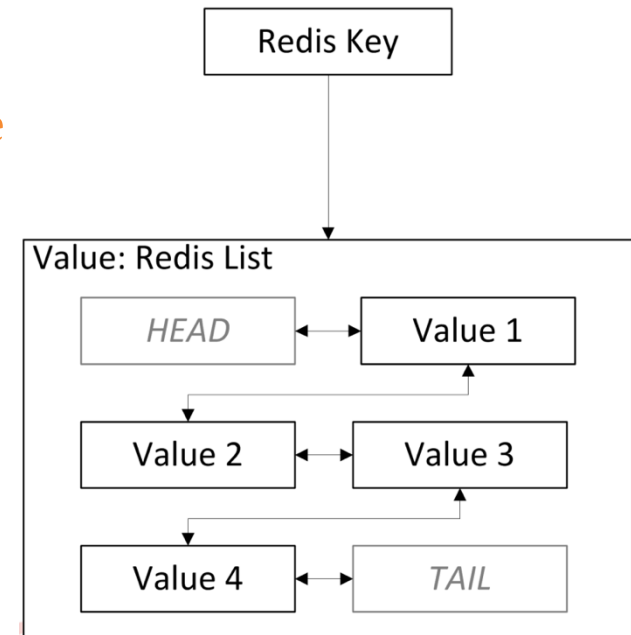
**INCR** counter  
(integer) 102

**INCRBY** counter 50  
(integer) 152

# Lists commands

## Insertion of new elements

- **LPUSH** *key value*
- **RPUSH** *key value*
  - Adds a new element to the head / tail
- **LINSERT** *key BEFORE|AFTER pivot value*
  - Inserts an element before / after another one
- Retrieval of elements



LPUSH, RPUSH, LPOP, and RPOP are  $O(1)$  operations

LINSERT is  $O(N)$  where  $N$  is the number of elements to traverse before reaching the pivot

## Retrieval of elements

- **LINDEX** *key index* – gets an element by its index
  - The first item is at position 0
  - Negative positions are allowed as well
  - LINDEX is  $O(N)$  where  $N$  is the number of elements to traverse
- **LRANGE** *key start stop* – gets a range of elements
  - LRANGE is  $O(S+N)$  where  $S$  is the start offset and  $N$  is the number of elements returned

## Removal of elements

- **LREM** *key count value*
  - Removes a given number of matching elements from a list,  $O(N)$ 
    - Positive / negative = moving from head to tail / tail to head
    - 0 = all the items are removed
- **LTRIM** *key start stop*
  - Trims an existing list so that it will contain only the specified range of elements

## Other operations

- **LLEN** *key* – gets the length of a list,  $O(1)$

# Lists examples

```
RPUSH mylist Alpha Beta first
```

```
(integer) 3
```

```
LRANGE mylist 0 -1
```

```
1) "Alpha"
```

```
2) "Beta"
```

```
3) "first"
```

```
# Retrieves all elements from "mylist" from the first (index 0) to the last (index -1) element.
```

```
RPUSH mylist 1 2 3 4 5 "foo bar"
```

```
(integer) 9
```

```
RPOP mylist
```

```
"foo bar "
```

```
# This removed and returned "foo bar", leaving 8 elements.
```

```
LTRIM mylist 0 2
```

```
OK
```

```
# This command trimmed the list to keep only the first three elements (indices 0, 1, and 2).
```

# Sets commands

## Basic operations

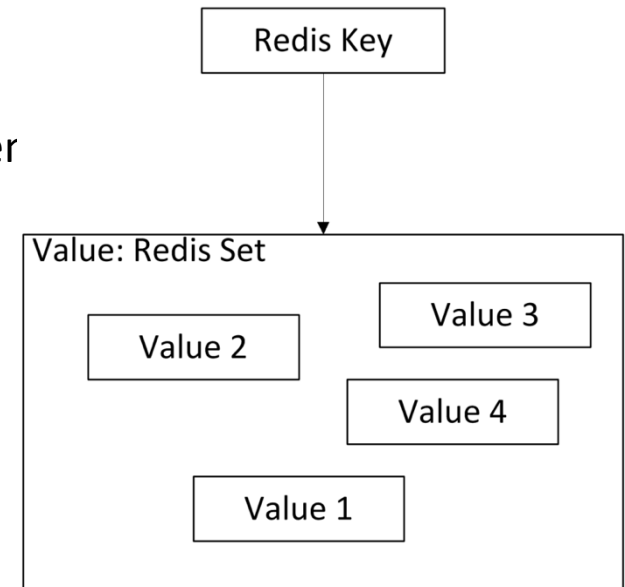
- **SADD** *key value ...*  
Adds an element / elements into a set
- **SREM** *key value ...*  
Removes an element / elements from a set

## Data querying

- **SISMEMBER** *key value*  
Determines whether a set contains a given eler
- **SMEMBERS** *key*  
Gets all the elements of a set

## Other operations

- **SCARD** *key*  
Gets the number of elements in a set



- **SUNION** *key ...*
- **SINTER** *key ...*
- **SDIFF** *key ...*
  - Calculates and returns a set union / intersection / difference of two or more sets (result is a new set)

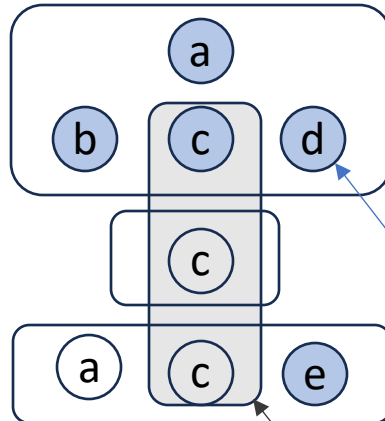
# Set example

Create a set with elements 1, 2, 3

```
SADD myset 1 2 3  
(integer) 3
```

```
SMEMBERS myset
```

- 1) "1"
- 2) "2"
- 3) "3"



```
SADD set1 a b c d  
(integer) 4
```

```
SADD set2 c  
(integer) 1
```

```
SADD set3 a c e  
(integer) 3
```

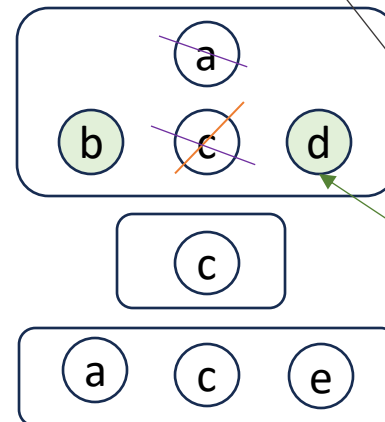
```
SUNION set1 set2 set3
```

- 1) "a"
- 2) "e"
- 3) "c"
- 4) "d"
- 5) "b"

Test existence of 3 and 30 in the set

```
SISMEMBER myset 3  
(integer) 1
```

```
SISMEMBER myset 30  
(integer) 0
```



```
SINTER set1 set2 set3
```

- 1) "c"

```
SDIFF set1 set2 set3
```

- 1) "b"
- 2) "d"

## Basic operations

- **ZADD** *key score value .....*
  - Inserts one element / multiple elements into a sorted set
- **ZREM** *key value ...*
  - Removes one element / multiple elements from a sorted set

## Working with score

- **ZSCORE** *key value*
  - Gets the score associated with a given element
- **ZINCRBY** *key increment value*
  - Increments the score of a given element

## Retrieval of elements

- **ZRANGE** *key start stop* [WITHSCORES]
  - Returns all the elements within a given range based on positions
  - Optional WITHSCORES returns both elements and their scores
- **ZRANGEBYSCORE** *key min max* [WITHSCORES]
  - Returns all the elements within a given range based on scores
- **ZREVRANGE** and **ZREVRANGEBYSCORE** are reverse order variants

147

## Other operations

- **ZCARD** *key*  
Gets the overall number of all elements
- **ZCOUNT** *key min max*  
Counts all the elements within a given range based on score

# Sorted Sets example

**ZADD** hackers **1940** "Alan Kay"

(integer) 1

**ZADD** hackers **1957** "Sophie Wilson" **1953** "Richard Stallman" **1949** "Anita Borg"

**1965** "Yukihiro Matsumoto" **1914** "Hedy Lamarr"

**1916** "Claude Shannon" **1969** "Linus Torvalds" **1912** "Alan Turing"

148

**ZRANGE** hackers **0 3**

- 1) "Alan Turing"
- 2) "Hedy Lamarr"
- 3) "Claude Shannon"
- 4) "Alan Kay"

# Sorted Sets example

**Born before 1950:**

**ZRANGEBYSCORE hackers -inf 1950**

- 1) "Alan Turing"
- 2) "Hedy Lamarr"
- 3) "Claude Shannon"
- 4) "Alan Kay"
- 5) "Anita Borg"

**Find the number of hackers born before World War II (before 1939)**

**ZCOUNT hackers -inf 1939**

(integer) 3

**Remove hackers born between 1940 and 1960:**

**ZREMRANGEBYSCORE hackers 1940 1960**

(integer) 4

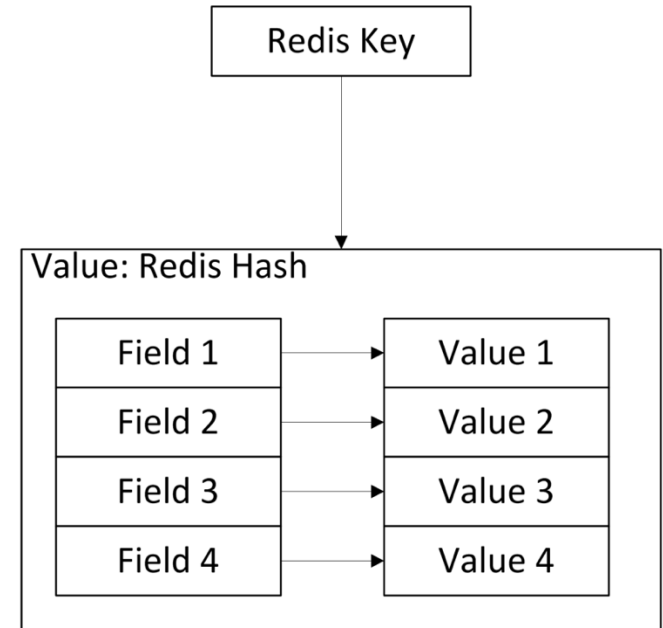
# Hash operations

## Basic operations

- **HSET** *key field value*
  - Sets the value of a hash field
- **HGET** *key field*
  - Gets the value of a hash field

## Batch alternatives

- **HMSET** *key field value ... DEPRECATED*
  - Sets values of multiple fields of a given hash
- **HMGET** *key field ...*
  - Gets values of multiple fields of a given hash



## Field retrieval operations

- **HEXISTS** *key field* – determines whether a field exists
- **HGETALL** *key* – gets all the fields and values
  - Individual fields and values are interleaved
- **HKEYS** *key* – gets all the fields in a given hash
- **HVALS** *key* – gets all the values in a given hash

## Other operations

- **HDEL** *key field [field ...]*
  - Removes a given field / fields from a hash
- **HLEN** *key* – returns the number of fields in a given hash

# Hash example

```
HSET user:1000 username antirez birthyear 1977 verified 1  
(integer) 3
```

```
HGET user:1000 username  
"antirez"
```

```
HGET user:1000 birthyear  
"1977"
```

```
HGETALL user:1000
```

- 1) "username"
- 2) "antirez"
- 3) "birthyear"
- 4) "1977"
- 5) "verified"
- 6) "1"

# Hash example - Shopping Cart

## Relational model

*cars*

<u>CartID</u>	User
1	matej
2	tomas
3	matej

*cart\_lines*

<u>Cart</u>	<u>Product</u>	Qty
1	45	1
1	78	2
2	51	3
2	213	2
2	94	6

UPDATE cart\_lines

SET Qty = Qty + 3

WHERE Cart = 1 AND Product = 45

## Redis model

**SADD** carts\_matej 1 3  
(integer) 2

**SADD** carts\_tomas 2  
(integer) 1

**HSET** cart:1 user "matej" product:45 1 product:78 2  
(integer) 3

**HSET** cart:2 user "tomas" product:51 3 product:213  
2 product:94 6  
(integer) 4

**HINCRBY** cart:1 product:45 3

# Hash example - Shopping Cart

Querying data: what is in the carts of matej?

**SMEMBERS** carts\_matej

- 1) "1"
- 2) "3"

**HGETALL** cart:1

- 1) "user"
- 2) "matej"
- 3) "product:45"
- 4) "4"
- 5) "product:78"
- 6) "2"

**HGETALL** cart:3

...

Redis supports Lua scripting, which allows you to execute a series of commands atomically.

# NoSQL principles & Redis Architectures

# Redis: NoSQL principles

- **Scalability:**
  - Redis supports horizontal scalability.
- **Sharding:**
  - Redis Cluster: Each node is responsible for a specific key range.
- **Replication:**
  - Primary-replica (master-slave) replication model.
- **CAP Theorem:**
  - **CP** system (Consistency and Partition tolerance).
    - ✓ Certain configurations and use cases might lean towards **AP** (Availability and Partition tolerance)
- **Consistency:**
  - Strong consistency within a single node.
  - Eventual consistency model in Redis Cluster.
  - ACID transactions at individual node level.

# Hash Function configuration

- **Hash Function** is configured and managed by the Redis system itself
  - **CRC16** (Cyclic Redundancy Check) algorithm.
  - Lookup table with pre-calculated CRC values for every possible byte value (256 entries)
- **Hash Slot Configuration:**
  - Data is distributed across 16384 hash slots (the number is fixed).
  - $\text{HASH\_SLOT} = \text{CRC16}(\text{key}) \bmod 16384$ .
- **Automatic Distribution**
- **Cluster Setup:**
  - An administrator defines the number of nodes and their roles.
  - Redis then automatically distributes hash slots among primary (master) nodes.
- **Rebalancing:**
  - Redis automatically redistributes hash slots when adding or removing.
- **Lack of User Customization:**
  - CRC16 is not changeable in standard Redis configuration.
- **Consistent Hashing (with Redis specifics)**

## Availability:

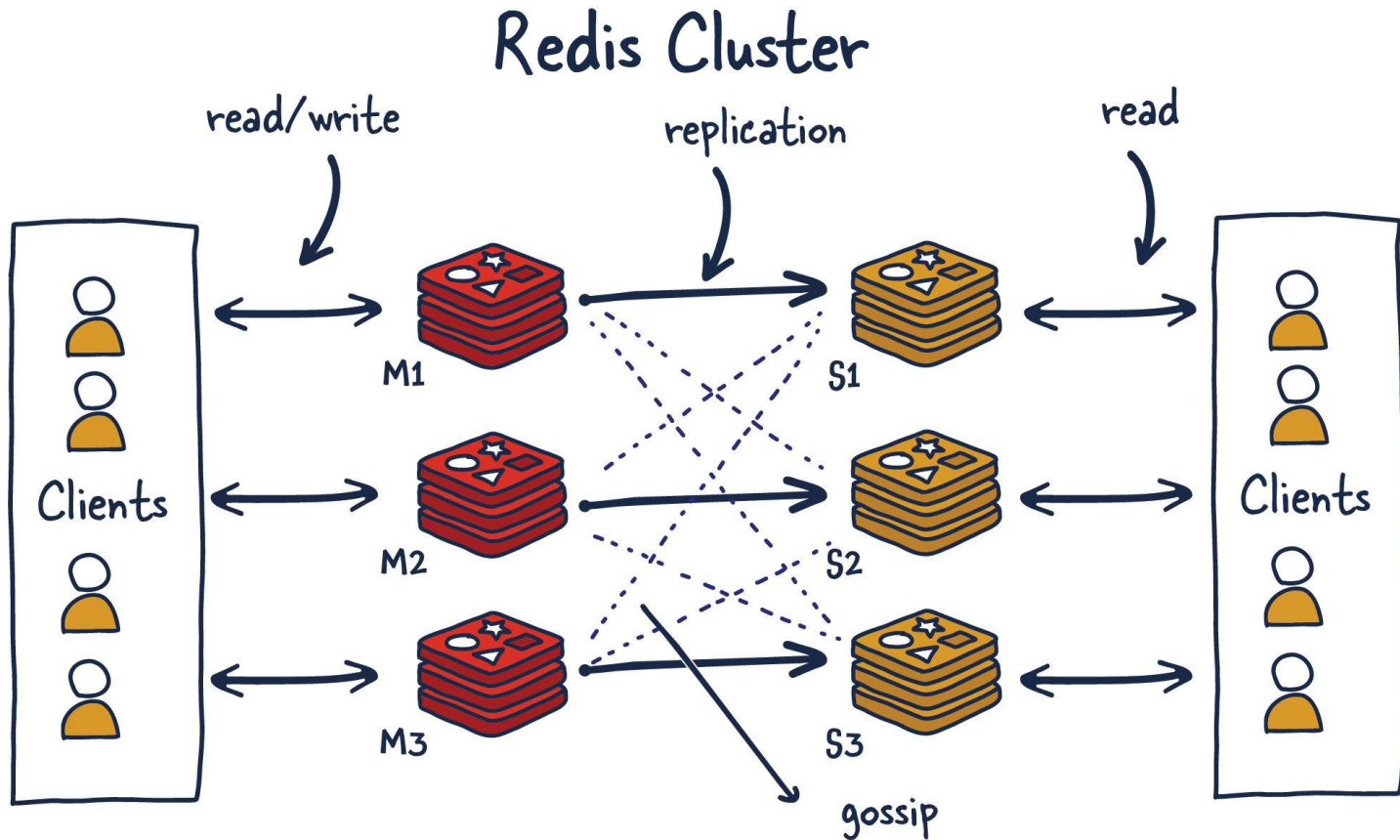
- Redis uses a master-slave replication model to ensure high availability.
  - There is a single “master” node that accepts all writes and multiple “slave” nodes that replicate data from the master in real time.
  - In the event the master node fails, one of the slave nodes can be promoted to become the new master.

## Consistency:

- Redis provides strong consistency guarantees for single-key operations.
  - If a value is written to a key, it will be immediately available for reads from any node in the cluster.
  - However, Redis does not provide transactional consistency for multi-key operations, meaning that some nodes may see a different view of the data than others.

## Partitioning:

- Redis supports sharding, which allows the data set to be partitioned across multiple nodes.
  - Redis uses a hash-based partitioning scheme, assigning each key to a specific node based on its hash value.
  - Redis also provides a mechanism for redistributing data when nodes are added or removed from the cluster.



Source: <https://architecturenotes.co/redis/>

## Simple Database



Main

## HA Database



Main



replication



Secondary

## Redis sentinel

Sentinel nodes



Main



replication



Secondary  
1



Secondary  
2

**Redis Sentinel** is a distributed system. Its key functions:

- **Monitoring**
  - ✓ Constantly checks the health of master and slave Redis nodes.
  - ✓ Tracks the connection between master and slave nodes.
- **Notification**
  - ✓ Alerts administrators or other computer programs about issues in the Redis cluster
- **Automatic failover**
  - ✓ When a master node isn't available, and enough (quorum of) nodes agree that it is, it selects the most suitable slave to promote to master.
  - ✓ It informs other Sentinels about the master node change.
  - ✓ Reconfigures remaining slave nodes to work with the new master.
- **Client Configuration**
  - ✓ Acts as a discovery service for clients, informing them about the current master node.

# Consistency vs. Availability in Redis

## Scenario: **High Write Load**

Example: Order processing system in an e-commerce store during a sale.

- Availability-focused setup:
  - Redis is configured with asynchronous replication. The master node confirms writes immediately, without waiting for replica acknowledgment.
  - Advantage: High performance and availability.
  - Drawback: Risk of data loss if the master fails before replication completes.
- Consistency-focused setup:
  - Using the WAIT command for synchronous replication. Write confirmation occurs only after replication to a certain number of nodes.
  - Advantage: Guaranteed data durability.
  - Drawback: Increased response time and reduced availability during network issues.

# Consistency vs. Availability in Redis

Scenario: **Geographically Distributed System**

Example: Global user session caching system.

- Prioritizing availability:
  - Using an active-active configuration with multiple master nodes in different regions.
  - Advantage: Low latency for users in different regions.
  - Drawback: Possible data conflicts when simultaneous updates occur in different regions.
- Prioritizing consistency:
  - Using a single global master node with replicas in different regions.
  - Advantage: Guaranteed data consistency.
  - Drawback: Increased latency for remote regions, risk of unavailability if the main node has issues.

# Consistency vs. Availability in Redis

## Scenario: **Processing Financial Transactions**

Example: User balance management system in online banking.

- Strict consistency:
  - Using Redis in synchronous replication mode with confirmation from a majority of nodes.
  - Advantage: Guaranteed integrity of financial data.
  - Drawback: Reduced performance and possible delays during network issues.
- Relaxed consistency:
  - Asynchronous replication with periodic synchronization.
  - Advantage: High performance and availability.
  - Drawback: Risk of temporary balance discrepancies between different nodes.

# Consistency vs. Availability in Redis

## Scenario: **Real-time Monitoring System**

Example: Monitoring server performance metrics.

- Emphasis on availability:
  - Using local Redis instances on each server with asynchronous replication to a central node.
  - Advantage: Continuous data collection even during network issues.
  - Drawback: Possible data inconsistency on the central node.
- Emphasis on consistency:
  - Direct writing of all metrics to a central Redis cluster.
  - Advantage: Consistent real-time data representation.
  - Drawback: Risk of data loss during network issues.

# Summary

1. Redis is ultra-fast in-memory data store
  - Not a database, used along with databases
2. Supports strings, numbers, lists, hashes, sets, sorted sets, publish / subscribe messaging
3. Used for caching / simple apps

