FAKULTA ELEKTROTECHNICKÁ

České vysoké učení technické v Praze

B4M36DS2 – Database Systems 2

Lecture 6 – Key-Value stores: Redis

27. 10. 2025

Yuliia Prokop

prokoyul@fel.cvut.cz, Telegram **@Yulia_Prokop**

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

Outline

Key-value stores

- Introduction

Redis

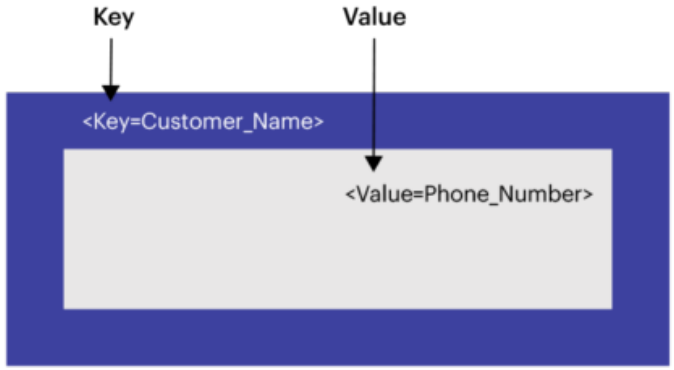
- Data model
- Keys
- Redis Data Types
- Basic commands and operations
- Examples
- NoSQL principles
- Redis architecture

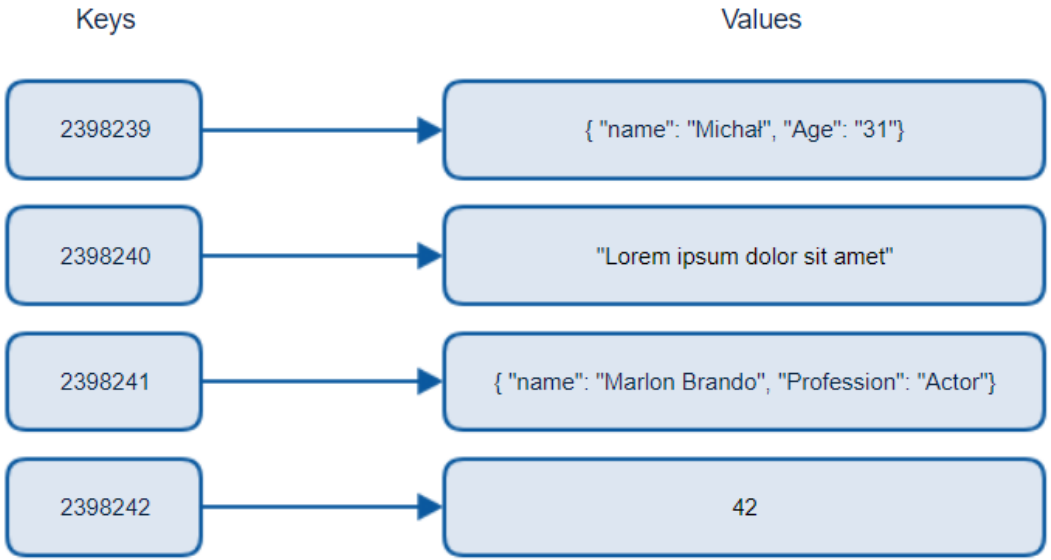
Key-value databases

Key-Value Stores

Phone directory

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





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.

Key Management

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**

Query Patterns

Basic **CRUD operations**

- Only when a key is provided
- ⇒ 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

Other Functionality

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.

Key-value database use cases

- **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.

Key-value database use cases

- **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

Redis functionality

- 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

Data Model

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**…

Redis keys

- **Keys** are binary safe – it is possible to use anybinary 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

Volatile Objects

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).

Volatile Objects

- **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)

Volatile Objects

- **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 - **suitable 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

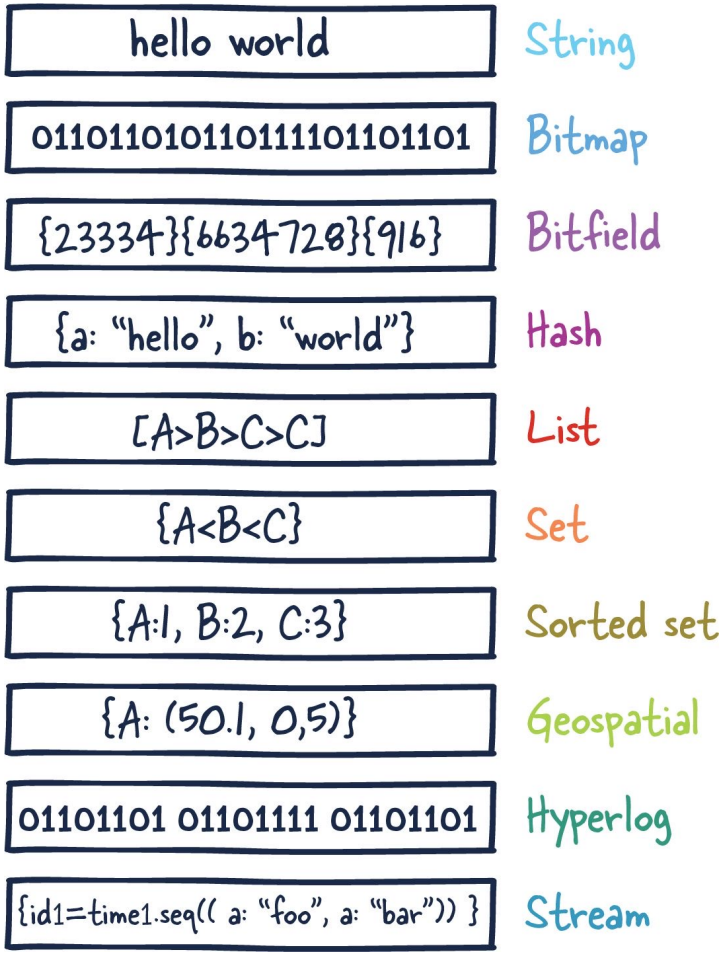
SET key 100 **EX** 10
OK

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/

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

Data Types : Strings

- **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

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

Data Types : Sets

- **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

Data Types: Sorted Sets

- **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)

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)

String commands

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

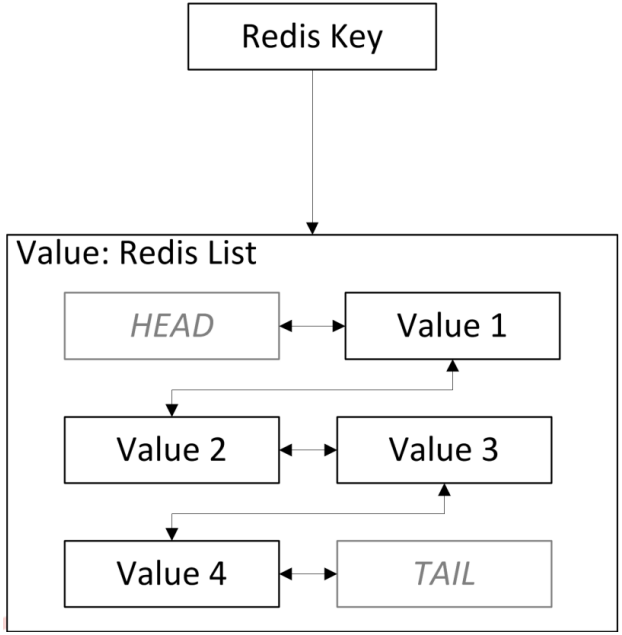
INCRBY counter 50
(integer) 152

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
  - **LPOP** key
  - **RPOP** key
    - Removes and returns the first / last element



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

Lists commands

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

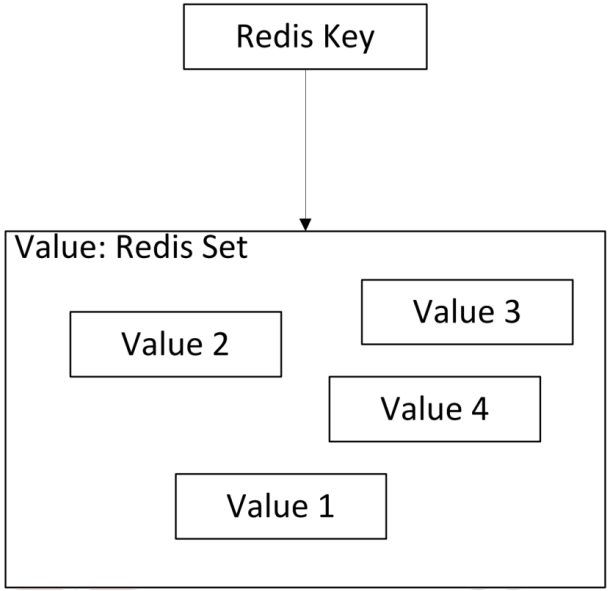
- **SMEMBERS** key

  Gets all the elements of a set

Other operations

- **SCARD** key

  Gets the number of elements in a set



SET OPERATIONS

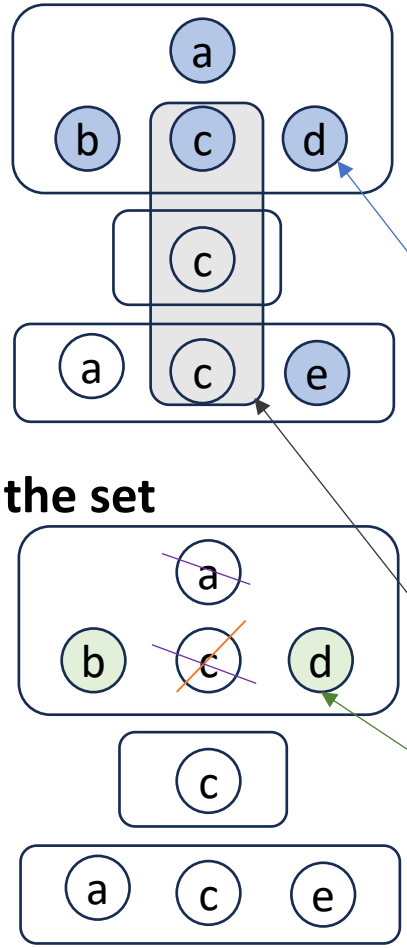
- **SUNION** key …
- **SINTER** key …
- **SDIFF** key …
  - Calculates and returns a set union / intersection / difference of two or more sets (result is a new set)

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"



Test existence of 3 and 30 in the set

SISMEMBER myset 3
(integer) 1

SISMEMBER myset 30
(integer) 0

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"

SINTER set1 set2 set3
1) "c"

SDIFF set1 set2 set3
1) "b"
2) "d"

Sorted Sets commands

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

Sorted Sets commands

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:

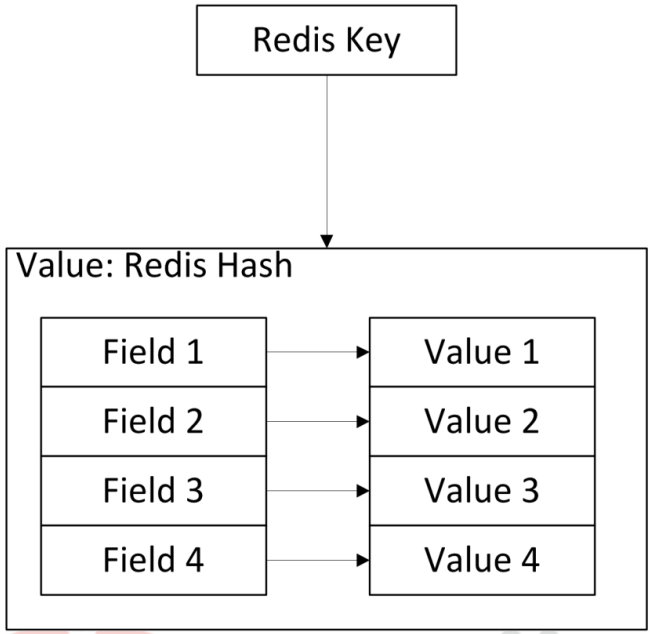
ZREMRANGEBYSCORE hackers 1940 1960

(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

Hash operations

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

carts

CartID	User
1	matej
2	tomas
3	matej

cart_lines

Cart	Product	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).
  - HASH_SLOT = CRC16(key) mod 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)**

Redis and the CAP Theorem

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.

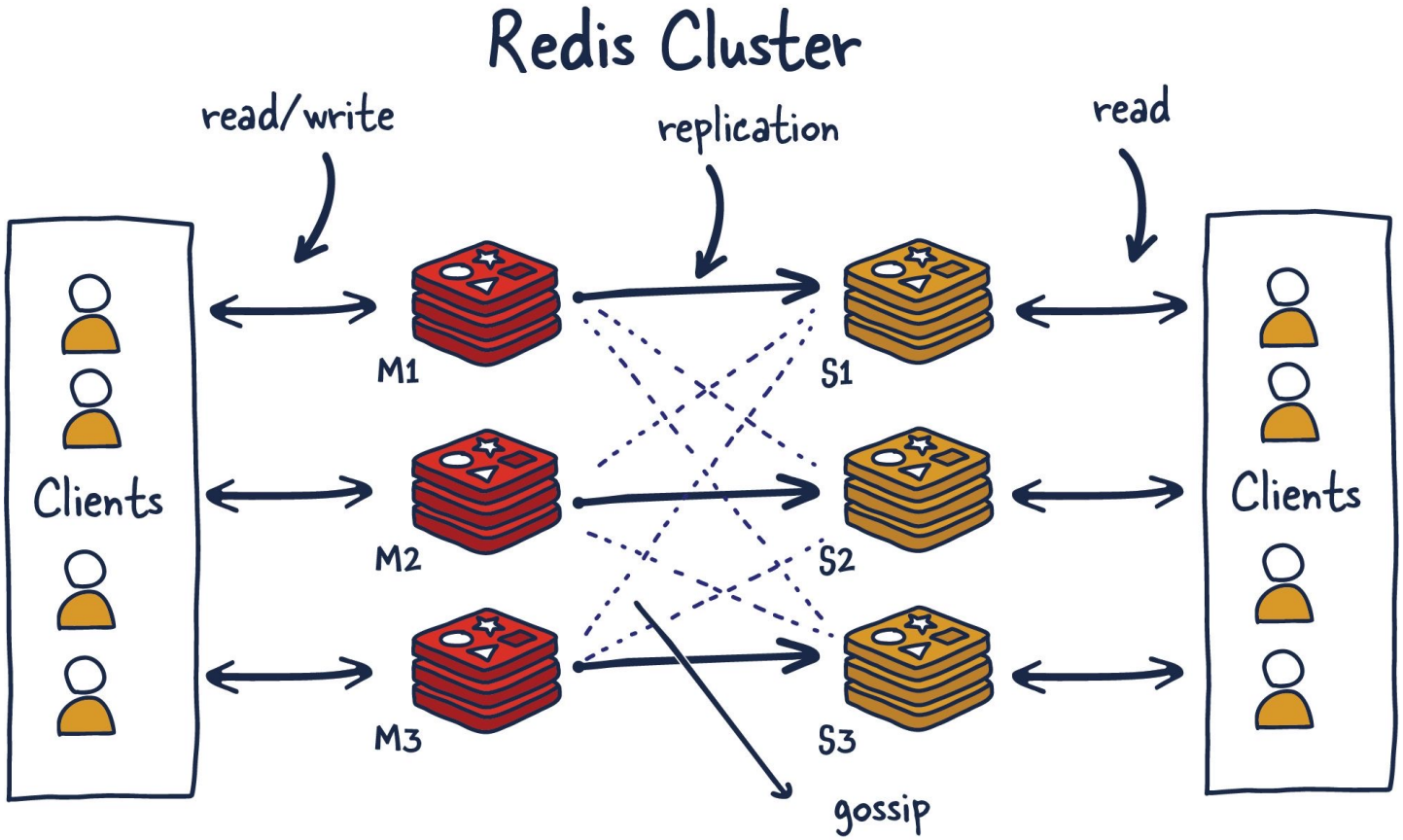
Consistency:

- 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:

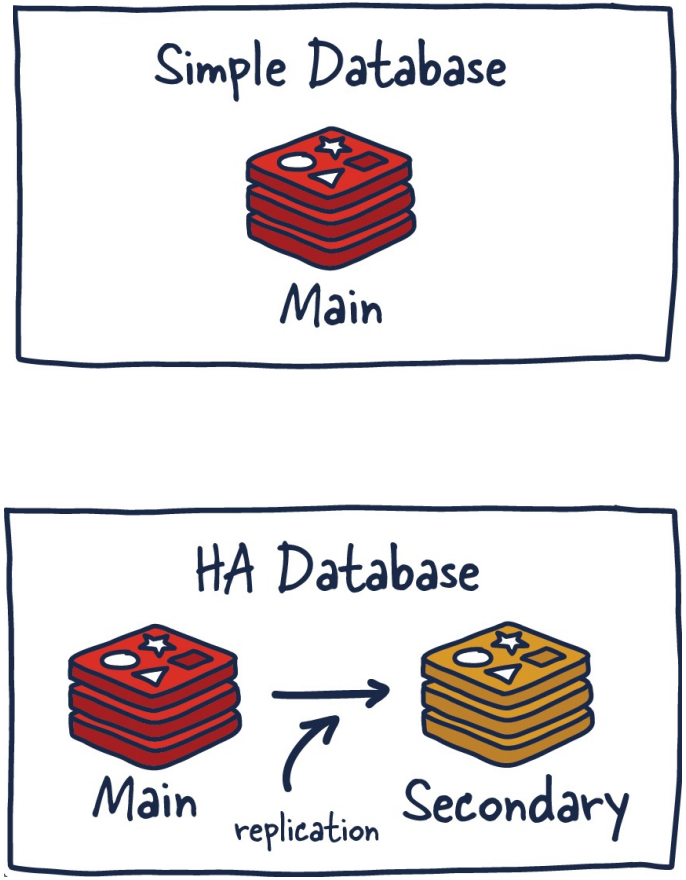
- Redis supports sharding, which allows the data set to be partitioned across multiple nodes.
  - Redis uses a hash-based partitioning scheme, assigning each key to a specific node based on its hash value.
  - Redis also provides a mechanism for redistributing data when nodes are added or removed from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

Redis Architecture



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

Redis Architecture



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

Redis Sentinel

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.

EXAMPLES

Redis as a primary data store

1. Rating Systems and Leaderboards

- Implementation details:
  - Uses Redis **sorted** sets.
  - Each player or participant is represented as a member of the set.
  - The player's score is used as the score in the sorted set.
  - The ZADD command adds or updates a player's score.
  - ZREVRANGE retrieves the top N players.
  - ZRANK determines a specific player's position in the ranking.

```
# Adding players to the leaderboard
ZADD leaderboard 1000 "player1" 2000 "player2" 3000 "player3"
# Getting the top 3 players with their scores
ZREVRANGE leaderboard 0 2 WITHSCORES
# Getting the rank of a player (zero-based ranking)
ZRANK leaderboard "player1"
# Incrementing a player's score by 500 points
ZINCRBY leaderboard 500 "player1"
# Getting the count of players with scores between 2000 and 3000
ZCOUNT leaderboard 2000 3000
```

Redis as a primary data store

2. Session Management Systems

- Implementation details:
  - Each session is stored as a hash in Redis.
  - The key is a unique session identifier.
  - The hash stores various session attributes (user ID, last access time, etc.).
  - The EXPIRE command is used to automatically delete outdated sessions.

```
HSET session:abc123 user_id 1000 last_access 1631234567 is_logged_in 1
# Session expires in one hour
EXPIRE session:abc123 3600
# Get all session data
HGETALL session:abc123
```

Redis as a primary data store

3. Caching Systems

- Implementation details:
  - It uses simple string keys to store cached data.
  - Values can be serialized objects or JSON strings.
  - The **SET** command with **EX** parameter sets a value with a lifetime.

```
SET "user:profile:1000" "{\"name\":\"John\",\"email\":\"john@example.com\"}" EX 300
GET "user:profile:1000"
```

4. Real-time Counters and Statistics

- Implementation details:
  - Uses string keys for simple counters.
  - Hashes are used for more complex statistics.
  - The **INCR** command increases the counter value.
  - **HINCRBY** is used to increase values in a hash.

```
HSET stats:2023-09-10 pageviews 150 unique_visitors 75
INCR "visits:total"
HINCRBY "stats:2023-09-10" pageviews 1
HINCRBY "stats:2023-09-10" unique_visitors 1
```

Redis as a primary data store

5. Rate Limiting Systems

- Implementation details:
  - Uses keys that include a user identifier or IP address.
  - The key value is the number of requests.
  - **INCR** increases the request counter.
  - EXPIRE sets the key's lifetime.

```
INCR "rate:ip:192.168.1.1"

# Limit resets after 60 seconds
EXPIRE "rate:ip:192.168.1.1" 60

# Check the current number of requests
GET "rate:ip:192.168.1.1"
```

6. Simple Message Queue Systems

- Implementation details:
  - Uses Redis lists to implement queues.
  - **LPUSH** adds elements to the beginning of the queue.
  - **RPOP** extracts elements from the end of the queue.
  - For reliability, **BRPOP** can be used to block extraction.

```
# Add a task to the queue
LPUSH "queue:tasks" "{"task": "send_email", "to": "user@example.com"}"
# Get the length of the queue
LLEN "queue:tasks"
# Limit the queue size to 1000 elements
LTRIM "queue:tasks" 0 999
# Wait and extract a task
BRPOP "queue:tasks" 0
```

Redis as a primary data store

7. Application Configuration Storage

- Implementation details:
  - Uses hashes to store settings.
  - Each configuration section can be a separate hash.
  - **HSET** sets or updates settings.
  - **HGETALL** retrieves all settings of a section.

```
HSET "config:app" debug_mode "true"
HSET "config:app" max_connections "1000"
HGETALL "config:app"
```

Output:

1) "debug_mode"
2) "true"
3) "max_connections"
4) "1000"

Redis example - Banners

We want to place banners on the page at a specific position and rotate them evenly; after each page reloads, they change.

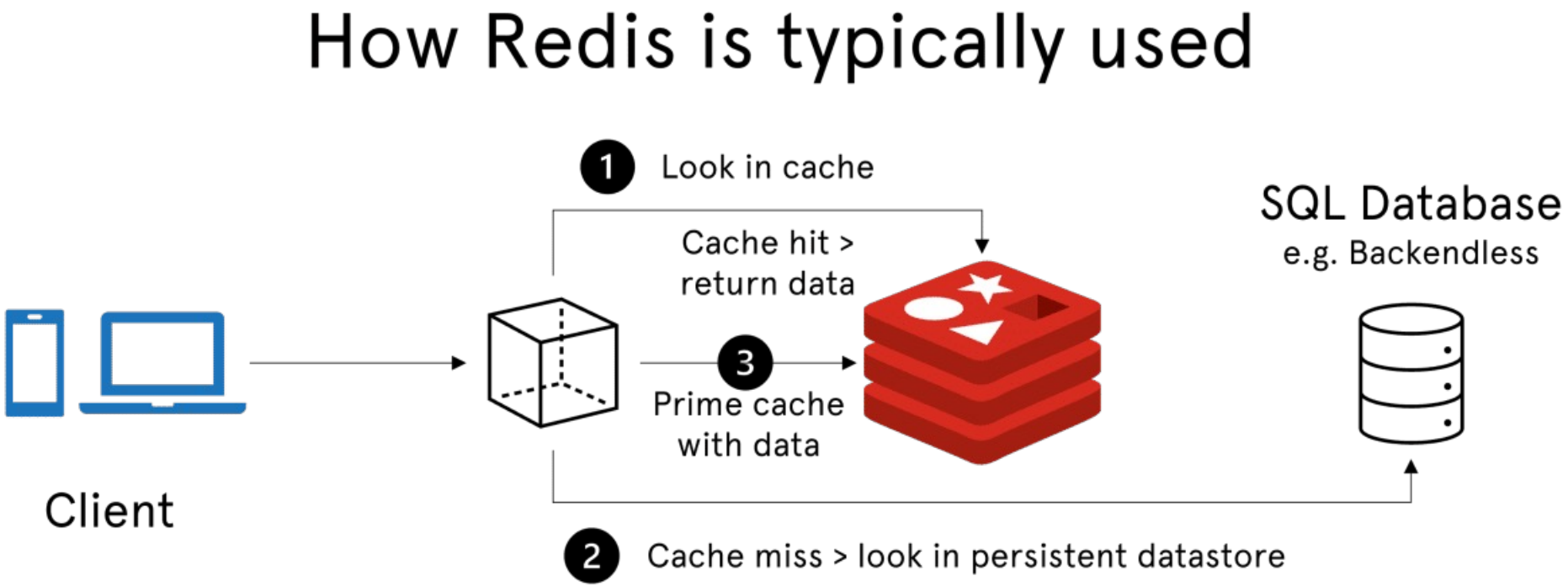
```
# Add a banner to the rotation
ZADD banners 0 {banner}
# Return a banner with fewer views
ZRANGE banners 1
#Increase banner's views
ZINCRBY banners 1 {banner}
# Remove an outdated or irrelevant banner from rotation
ZREM banners {banner}
# Get the total number of banners in rotation
ZCARD banners
# Get up to 10 banners with view counts between 0 and 100
ZRANGEBYSCORE banners 0 100 LIMIT 0 10
```

Redis example - Payment

Purchasing and payment

```
# Top up balance
ZINCRBY balance 500 user:1234

# Withdraw the sum
ZINCRBY balance -500 user:1234

# Add information to a log
ZADD purchases 1634567890 "product:1234:quantity:2:price:100"
```

Redis as cash



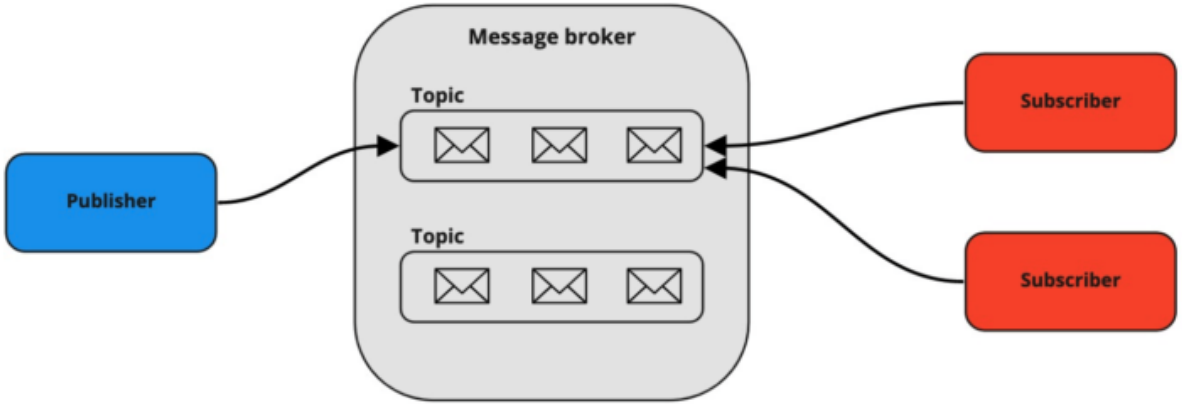
Source: https://backendless.com

Redis as cache

```
def get_tourists():
    # Check the cache first
    key = "tourists"
    tourists = redis.get(key)
    if tourists:
        # The tourists are in the cache, return them
        return tourists.decode('utf-8’)

        # The tourists are not in the cache, query the database
        cursor = pool.cursor()
        cursor.execute("SELECT * FROM tourists")
        tourists = cursor.fetchall()

        # Save the tourists to the cache
        redis.set(key, str(tourists))

        return tourists
```

```
tourists1 = get_tourists()
print('Tourists from DB:', tourists1, '\n')

# Will be retrieved faster because of caching
tourists2 = get_tourists()
print('Tourists from CACHE:', tourists1)
```

Publish / Subscribe



https://hevodata.com

Publish / Subscribe

For notifications and allerts

Redis is also a message broker that supports typical pub/sub operations.

- The first user subscribes to certain channel , "**news**"

  SUBSCRIBE news

- Another user sends messages to the same channel , "**news**"

  PUBLISH news "hello"

- Subscribed clients receive the message:

  1) "message"
  2) "news"
  3) "hello"

Subscribers can listen to multiple channels, and publishers can send to multiple channels.

- Learn more at http://redis.io/commands#pubsub

Redis Streams

Redis Streams

- A data structure for stream processing
- Designed to store multiple records ordered by insertion time

Main Characteristics:

- Append-only logs
- Unique IDs for each entry (timestamp + sequence number)
- Consumer Groups support
- Built-in persistence

Applications:

- Event logging
- Message processing
- Real-time analytics
- Event sourcing
- Messaging systems

Read more: https://redis.io/docs/latest/develop/data-types/streams/

Redis Streams: Basic commands

XADD mystream [ID] field1 value1 [field2 value2 ...]

Add new entry

XREAD [COUNT count] [BLOCK milliseconds] STREAMS key [key ...] ID [ID ...]

Read stream entries

XRANGE key start end [COUNT count]

Read stream entries in a specific range

XLEN key

Get stream length

XGROUP CREATE key groupname id [MKSTREAM]

Creating and Managing Groups

XREADGROUP GROUP group consumer [COUNT count] STREAMS key [key ...] ID [ID ...]

Reading as Consumer Group

XACK key group ID [ID ...]

Message Acknowledgment

Redis Streams: Examples

XADD race:france * rider Castilla speed 30.2 position 1 location_id 1 "1692632086370-0"

Add a stream entry for each racer that includes the racer's name, speed, position, and location ID

XREAD COUNT 100 BLOCK 300 STREAMS race:france $

Read up to 100 new stream entries, starting at the end of the stream, and block for up to 300 ms if no entries are being written .

XRANGE race:france 1692632086370-0 + COUNT 2

Read two stream entries starting at ID 1692632086370-0

XLEN race:france

Get the number of items inside a Stream

XGROUP CREATE race:france france_riders $

Create a consumer group

XREADGROUP GROUP italy_riders Alice COUNT 1 STREAMS race:italy >

Add riders to the race:italy stream and try reading something using the consumer group

XACK race:italy italy_riders 1692632639151-0

Acknowledges processing of message ID 1692632639151-0

Redis Bitfields

Redis Bitfields

- Space-efficient data structure for storing multiple counters/integers
- Allows manipulation of integer values at the bit level
- Introduced to handle binary data and numeric arrays efficiently

Key Features:

- Multiple integers packed into a single Redis key
- Support for different integer sizes (1-63 bits)
- Atomic operations guaranteed
- Signed and unsigned integers support
- Memory efficient storage

Use Cases :

- Rate limiting
- User presence tracking
- Performance metrics
- Feature flags

Read more: https://redis.io/docs/latest/develop/data-types/bitfields/

Redis Bitfields : Basic command structure

BITFIELD key [GET type offset] [SET type offset value] [INCRBY type offset increment]

Type Specification:

- i[bits] – signed integer (i8, i16, i32)
- u[bits] – unsigned integer (u8, u16, u32)

Overflow Handling:

- WRAP: wrap around values (default)
- SAT: saturate at min/max
- FAIL: return null on overflow

Command Options:

- GET: retrieve integer values
- SET: set integer values
- INCRBY: increment values
- OVERFLOW: set overflow behavior

Redis Bitfields : Examples

BITFIELD mykey SET u8 0 42

Sets an 8-bit unsigned integer at offset 0 to value 42

BITFIELD mykey SET u8 0 42 SET u8 8 24

Sets two 8-bit integers: 42 at offset 0 and 24 at offset 8

BITFIELD mykey OVERFLOW SAT INCRBY u8 0 100

Increments value by 100, saturating at maximum value (255 for u8)

BITFIELD mykey GET u8 0 GET u8 8

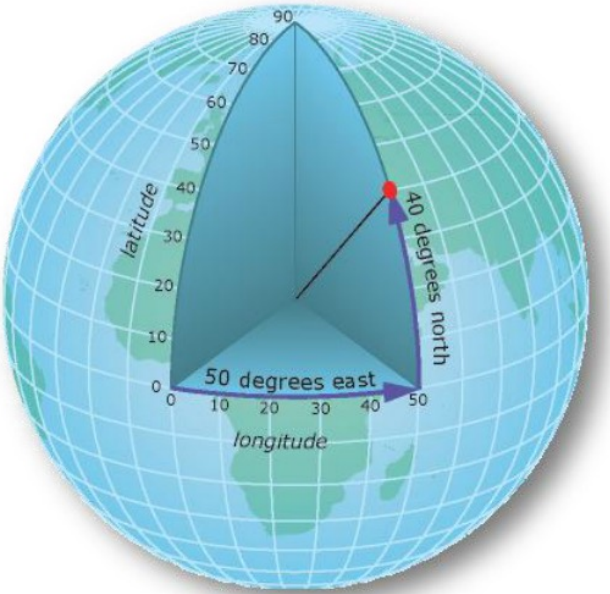
Retrieves two 8-bit integers from offsets 0 and 8

BITFIELD mykey OVERFLOW WRAP

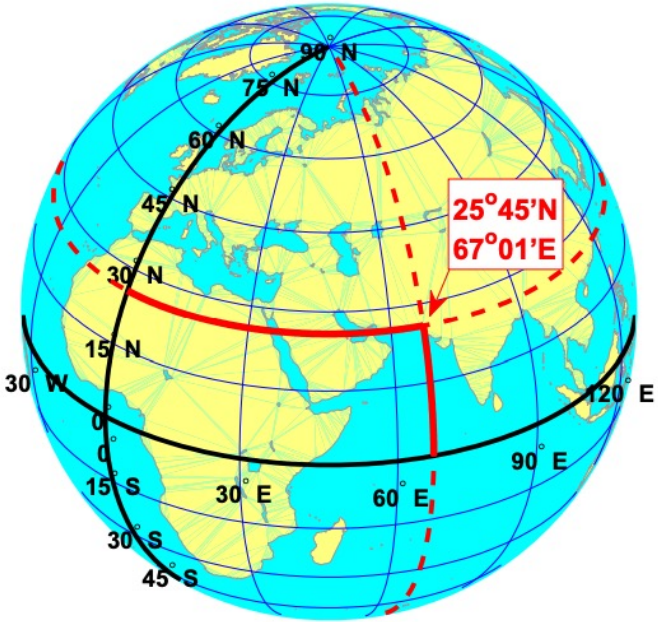
SET i8 0 100
INCRBY u16 8 1
GET i8 0

Sets 8-bit signed integer to 100, increments 16-bit unsigned integer at offset 8, retrieves the first value, uses WRAP overflow behavior.

GEOSPATIAL



Source: https://uwescience.github.io/SQL-geospatial-tutorial/04-geospatialQueries/



Source: https://carpentries-incubator.github.io/r-geospatial-urban/08-intro-to-geospatial-concepts.html

Redis geospatial

Redis **geospatial** indexes let you store **coordinates** and search for them. This data structure is useful for finding nearby points within a given radius or bounding box.

GEOADD

adds a location to a given geospatial index
(note that **longitude** comes before **latitude** with this command).

GEOPOS

Returns the position of one or more members in a geospatial index.

GEOSEARCH

Returns locations with a given radius or a bounding box.

GEODIST

Returns the distance between two members in the geospatial index represented by the sorted set.

GEORADIUS

Queries a sorted set representing a geospatial index to fetch members within a given distance.

Redis example – Data format

For Redis geospatial commands, the correct format is longitude followed by latitude. Examples:

12.4964 41.9028

12.4964, 41.9028

Valid ranges:
Longitude: -180 to 180
Latitude: -85.05112878 to 85.05112878

The first number is the **longitude**, and the second is the **latitude**.

An option like

longitude 2.2945 latitude 48.8584

is not in the correct format for Redis geospatial commands.

Redis uses geohashing internally for efficient storage and querying of coordinates.
Coordinate precision in Redis is limited to 5-6 decimal places.

Redis example – Filter by location

GEOADD Addresses 43.361389 18.115556 "Addr1"
25.087269 37.502669 "Addr2"

GEODIST Addresses Addr1 Addr2 km
Distance between two addresses

GEOSEARCH Addresses FROMLONLAT 15 37 BYRADIUS 15 km ASC
Everything within a 15-kilometer radius of the point

Read more about geospatial: https://redis.io/docs/data-types/geospatial/

Transactions

Transaction

- All commands are serialized and executed sequentially
- Either all commands or no commands are processed
- Keys must be explicitly specified in Redis transactions
- Redis does not support transactions between multiple shards.

- Redis commands for transactions:
  - ✓ **WATCH**
    - Marks the given keys to be watched for conditional execution of a transaction.
  - ✓ **MULTI**
    - Marks the start of a transaction block. Subsequent commands will be **queued** for atomic execution using **EXEC**.
  - ✓ **DISCARD**
    - Flushes all previously queued commands in a transaction
  - ✓ **EXEC**
    - Executes all previously queued commands in a transaction
    - If a watched key has been modified, the transaction will fail, no command will be executed
  - ✓ **UNWATCH**
    - Forgets about all watched keys

Transaction - Example

 # We update the player's score,
 # their position on the leaderboard,
 # and set a TTL for the player's data

MULTI
OK
HINCRBY user:1234 score 50
QUEUED
ZINCRBY leaderboard 50 "player1234"
QUEUED
EXPIRE user:1234 86400
QUEUED
EXEC
1) (integer) 250 # New player score
2) (float) 1430.5 # New leaderboard position
3) (integer) 1 # Successfully set key expiration

 # We attempt to update the player's
 # inventory and decrease their gold,
 # but the transaction is discarded

MULTI
OK
SADD inventory:5678 "health_potion"
QUEUED
SREM inventory:5678 "empty_bottle"
QUEUED
HINCRBY user:5678 gold -10
QUEUED
DISCARD
OK

Transaction – Example (all or nothing)

MULTI
SET key1 "value1"
INCR key2
SADD set1 "member1"
INCR nonexistent_key
SET key3 "value3"
EXEC
1) OK
2) (integer) 1
3) (integer) 1
4) (integer) 1
5) OK

MULTI
MULTI
SET key1 "value1"
INCR key2
SADD set1 "member1"
INCRBY key4 # Error: missing second argument
SET key3 "value3"
EXEC

MULTI
+ OK
SET key1 "value1"
+ QUEUED
INCRBY key2 # Syntax error: missing second argument
- ERR wrong number of arguments for 'incrby' command
SET key3 "value3"
+ QUEUED
EXEC
- EXECABORT Transaction discarded because of previous errors.

Transaction – Example

SET nonexistent_string "abc"
OK
MULTI
OK
SET key1 "value1"
QUEUED
INCR key2
QUEUED
HSET hash1 field1 "value"
QUEUED
LPUSH list1 "item1"
QUEUED
SADD set1 "member1"
QUEUED
INCR nonexistent_string
QUEUED
EXEC

1) OK
2) (integer) 1
3) (integer) 1
4) (integer) 1
5) (integer) 1
6) (error) ERR value is not an integer or out of range

Transaction - Errors inside a transaction

- ✓ Before EXEC is called
  - The command may be syntactically wrong (wrong number of arguments, wrong command name, ...),
  - There may be some critical conditions like an out-of-memory condition.
- ✓ After EXEC is called
  - If we operated against a key with the wrong value (like calling a list operation against a string value).

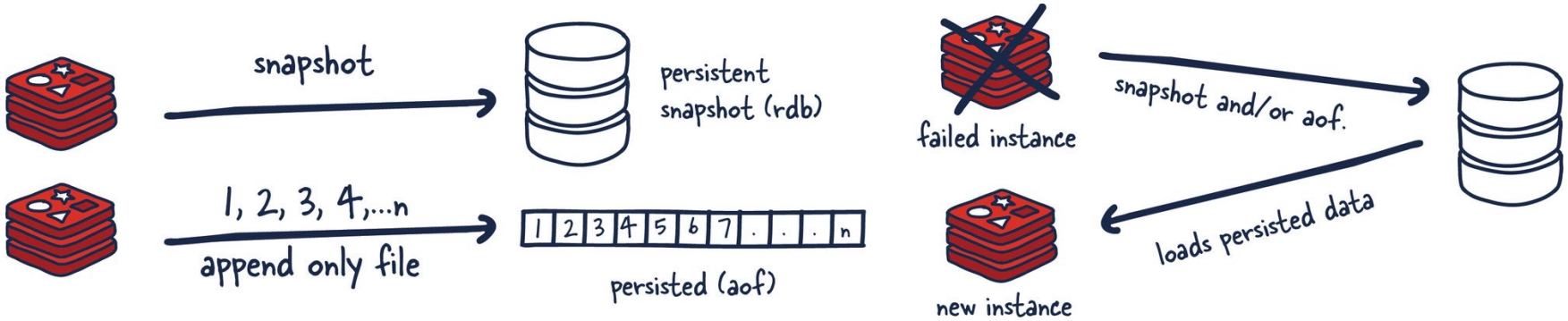
- ❑ Redis does not support rollbacks of transactions.
- ❑ DISCARD can be used to abort a transaction. In this case, no commands are executed, and the state of the connection is restored to normal.

PERSISTENCE

Persistance

Datasets can be saved to disk

Persistence refers to the writing of data to durable storage, such as a solid-state disk (SSD).



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

Persistence

Redis provides a range of persistence options. These include:

- **RDB** (Redis Database): RDB persistence performs point-in-time snapshots of your dataset at specified intervals.
- **AOF** (Append Only File): AOF persistence logs every write operation the server receives. These operations can then be replayed at server startup, reconstructing the original dataset.
- **No persistence**: You can disable persistence completely. This is sometimes used when caching.
- **RDB + AOF**: You can combine AOF and RDB in the same instance.

RDB Example

Example 1: RDB Persistence with Custom Save Points

In **redis.conf**, set the following options

save 900 1

    # Save the DB if at least 1 key changed in 900 seconds

save 300 10

    # Save the DB if at least 10 keys changed in 300 seconds

save 60 10000

    # Save the DB if at least 10000 keys changed in 60 seconds

AOF Example

Example 2: AOF Persistence with Every Second fsync

In **redis.conf**, set the following options

appendonly yes

    # Enable AOF persistence

appendfsync everysec

    # fsync every second

aof-use-rdb-preamble yes

    # Optimize AOF loading in Redis 7.0+

RDB and AOF Example

Example: RDB + AOF Persistence with No fsync

In **redis.conf**, set the following options

save 3600 1

    # Save the DB if at least 1 key changed in 3600 seconds

appendonly yes

    # Enable AOF persistence

appendfsync no

    # Do not fsync, leave it to the OS

RDB advantages and disadvantages

- ✓ RDB is a very compact single file for backups and disaster recovery.
- ✓ RDB maximizes Redis's performance since the only work Redis's parent process needs to do to persist is forking a child who will do all the rest. The parent process will never perform disk I/O or similar work.
- ✓ RDB allows faster restarts with big datasets than AOF.

- **Data Loss**: RDB snapshots are taken periodically, which means that in case of a system crash, you could lose data that has not yet been included in the most recent snapshot.
- **Forking Overhead**: The Redis process needs to fork a child process to create the RDB snapshot, which can be resource-intensive for large datasets.

AOF advantages and disadvantages

- ✓ **Better Durability**: AOF provides better data durability, as it logs every write operation, reducing the risk of data loss.
- ✓ **Human-Readable Format**: AOF files store the commands in plain text, making them easy to inspect and understand.
- ✓ **Flexible Configuration**: You can configure the AOF fsync policy to balance durability and performance based on your requirements.

Disadvantages of AOF

- **Larger File Size**: AOF files can be significantly larger than RDB files, as they store every write operation.
- **Slower Recovery**: The recovery process for AOF can be slower than that of RDBs, as Redis needs to replay all the logged commands to reconstruct the dataset.

Persistence: RDB vs AOF

RDB (Redis Database File)	AOF (Append Only File)
Provides point in time snapshots	Logs every write
Creates complete snapshot at specified interval	Replays at server startup. If log gets big, optimization takes place
File is in binary format	File is easily readable
On crash minutes of data can be lost	Minimal chance of data loss
Small files, fast (mostly)	Big files, 'slow'

Source: https://www.slideshare.net/MaartenSmeets1/introduction-redis-93365594

RDB related commands

BGSAVE

Save the DB in the background. Redis forks, the parent continues serving the clients, and the child saves the dataset on disk and exits.

SAVE

Perform a synchronous save of the dataset. Other clients are blocked – never use in production!

LASTSAVE

Return the Unix time of the last successful DB save.

AOF related commands

BGREWRITEAOF

Instruct Redis to start an AOF rewrite process. The rewrite will create a small optimized version of the current AOF log.

If **BGREWRITEAOF** fails, no data gets lost, as the old AOF will be untouched

Read more about persistence:
https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/

What persistence is used for?

- ✓ **Backups**
- ✓ **Disaster Recovery**
- ✓ **Performance Maximization**
- ✓ **Faster Restarts with Big Datasets**
- ✓ **Replicas**

Summary. Why Redis?

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