Redis

Redis Overview

Rakesh raj · Follow
7 min read · Jan 11, 2025

Redis (Remote Dictionary Server) is an open-source, in-memory, key-value data store. It is designed for high performance, offering low-latency read and write operations. Redis is classified as a NoSQL database and supports **advanced data types** beyond just strings, making it a **data structure server**.

Redis allows storing and manipulating data in **binary-safe strings**, **lists**, **sets**, **sorted sets**, **hashes**, **bitmaps**, and **hyperloglogs**.

Key Data Types in Redis

- **Strings** are used for simple key-value pairs.
- **Lists** allow for ordered collections and efficient operations at both ends.
- **Sets** provide unordered collections of unique elements.
- **Sorted Sets** allow elements to be stored with scores and automatically sorted.
- **Hashes** are ideal for storing field-value pairs, such as user profiles.

1. Working with Strings

Command: `SET`

- **Syntax:** `SET key value`

```
SET user:1000 "Alice"
```

- Stores the value “Alice” under the key `user:1000`.

Command: `GET`

- **Syntax:** `GET key`

```
GET user:100
```

- Returns: `"Alice"`

Command: `SETEX`

- **Syntax:** `SETEX key seconds value`

```
SETEX session:12345 300 "user123"
```

- Sets the key `session:12345` with value `"user123"`, which expires in 300 seconds (5 minutes).

Command: `PSETEX`

- **Syntax:** `PSETEX key milliseconds value`

```
PSETEX session:12345 5000 "user123"
```

- Same as `SETEX` but expiration time is in milliseconds (5 seconds).

Command: `SETNX`

- **Syntax:** `SETNX key value`

```
SETNX user:1001 "Bob"
```

- Sets the key `user:1001` to `"Bob"` only if the key doesn't already exist.

Command: `STRLEN`

- **Syntax:** `STRLEN key`

```
STRLEN user:1000
```

- Returns: `5` (the length of "Alice").

2. Working with Lists

Command: `LPUSH`

- **Syntax:** `LPUSH key value`

```
LPUSH fruits "apple"
LPUSH fruits "banana"
```

- Adds `"apple"` and `"banana"` to the `fruits` list. `"banana"` will be at the head of the list.

Command: `RPUSH`

- **Syntax:** `RPUSH key value`

```
RPUSH fruits "orange"
```

- Adds `"orange"` to the tail of the `fruits` list.

Command: `LPOP`

- **Syntax:** `LPOP key`

```
LPOP fruits
```

- Removes and returns the first element in the list (`"banana"`).

Command: `RPOP`

- **Syntax:** `RPOP key`

```
RPOP fruits
```

- Removes and returns the last element in the list (`"orange"`).

Command: `LRANGE`

- **Syntax:** `LRANGE key start end`

```
LRANGE fruits 0 -1
```

- Returns all elements in the list: `["banana", "apple"]`.

Command: `LLEN`

- **Syntax:** `LLEN key`

```
LLEN fruits
```

- Returns: `2` (the number of elements in the list).

3. Working with Sets

Command: `SADD`

- **Syntax:** `SADD key value`

```
SADD colors "red" "green" "blue"
```

- Adds `"red"`, `"green"`, and `"blue"` to the `colors` set.

Command: `SMEMBERS`

- **Syntax:** `SMEMBERS key`

```
SMEMBERS colors
```

- Returns: `["red", "green", "blue"]`.

Command: `SISMEMBER`

- **Syntax:** `SISMEMBER key value`

```
SISMEMBER colors "green"
```

- Returns: `1` (because `"green"` is present in the set).

Command: `SCARD`

- **Syntax:** `SCARD key`

```
SCARD colors
```

- Returns: `3` (number of elements in the set).

Command: `SREM`

- **Syntax:** `SREM key value`

```
SREM colors "blue"
```

- Removes `"blue"` from the `colors` set.

Command: `SPOP`

- **Syntax:** `SPOP key`

```
SPOP colors
```

- Removes and returns a random element from the `colors` set, e.g., `"green"`.

4. Working with Sorted Sets (ZSets)

Command: `ZADD`

- **Syntax:** `ZADD key score value`

```
ZADD leaderboard 100 "Alice" 150 "Bob"
```

- Adds `"Alice"` with score `100` and `"Bob"` with score `150` to the `leaderboard` sorted set.

Command: `ZRANGE`

- **Syntax:** `ZRANGE key start end`

```
ZRANGE leaderboard 0 -1
```

- Returns: `["Alice", "Bob"]` (sorted by score, ascending).

Command: `ZRANGEBYSCORE`

- **Syntax:** `ZRANGEBYSCORE key min max`

```
ZRANGEBYSCORE leaderboard 100 150
```

- Returns: `["Alice", "Bob"]` (elements within the score range 100 to 150).

Command: `ZREM`

- **Syntax:** `ZREM key value`

```
ZREM leaderboard "Alice"
```

- Removes `"Alice"` from the `leaderboard` sorted set.

Command: `ZCARD`

- **Syntax:** `ZCARD key`

```
ZCARD leaderboard
```

- Returns: `1` (only `"Bob"` remains).

5. Working with Hashes

Command: `HMSET`

- **Syntax:** `HMSET key field1 value1 field2 value2`

```
HMSET user:1000 name "Alice" age 30
```

- Sets two fields (`name` and `age`) for the `user:1000` hash.

Command: `HGETALL`

- **Syntax:** `HGETALL key`

```
HGETALL user:1000
```

- Returns: `["name", "Alice", "age", "30"]`.

Command: `HGET`

- **Syntax:** `HGET key field`

```
HGET user:1000 name
```

- Returns: `"Alice"` (the value of the `name` field).

Command: `HDEL`

- **Syntax:** `HDEL key field`

```
HDEL user:1000 age
```

- Removes the `age` field from the `user:1000` hash.

Command: `HEXISTS`

- **Syntax:** `HEXISTS key field`

```
HEXISTS user:1000 name
```

- Returns: `1` (because the `name` field exists).

6. Additional Redis Commands

Command: `DEL`

- **Syntax:** `DEL key`

```
DEL user:1000
```

- Deletes the `user:1000` key from Redis.

Command: `INCR`

- **Syntax:** `INCR key`

```
INCR counter
```

- Increments the `counter` key by 1.

Command: `INCRBY`

- **Syntax:** `INCRBY key increment`

```
INCRBY counter 10
```

- Increments `counter` by 10.

Command: `EXPIRE`

- **Syntax:** `EXPIRE key seconds`

```
EXPIRE session:12345 600
```

- Sets a TTL of 600 seconds (10 minutes) for the key `session:12345`.

Command: `FLUSHALL`

- **Syntax:** `FLUSHALL`

```
FLUSHALL
```

- Removes all keys in the Redis instance.

Summary of Key Commands by Data Type

Data Type	Key Command	Description
String	`SET`, `GET`, `INCRBY`	Simple key-value storage
List	`LPUSH`, `RPUSH`, `LPOP`	Ordered collection, add/remove at both ends
Set	`SADD`, `SREM`, `SMEMBERS`	Unordered unique elements
Sorted Set	`ZADD`, `ZRANGE`, `ZSCORE`	Elements with scores, sorted automatically
Hash	`HMSET`, `HGETALL`, `HSETNX`	Field-value pairs for structured data

Redis Publisher/Subscriber Model (Pub/Sub)

The **Pub/Sub** model in Redis allows clients to send (publish) and receive (subscribe) messages in real-time, enabling communication between different systems or applications. Redis handles this model using **channels**, where messages are sent and received.

Message Queue:

- A **message queue** is like a temporary storage that holds messages sent by a publisher until a subscriber is ready to process them.
- **Publisher** sends a message to the queue.
- **Subscriber** retrieves and processes messages from the queue.

Use Case: For example, in a task-processing system, a publisher sends jobs to the queue, and subscribers pick them up one by one to process.

How Redis is Used as a Message Queue

Redis can function as a message queue through its **Pub/Sub** system. Here's how it works:

1. **Publisher:** Sends a message to a specific **channel.**
2. **Subscriber:** Listens for messages from one or more **channels** and processes the messages when they arrive.

Subscribing to a Channel

- **Command:** `SUBSCRIBE channel`
- **Description:** A client subscribes to a channel and listens for incoming messages.

```
SUBSCRIBE mychannel
```

- The client will now listen for messages sent to `mychannel`.

Publishing to a Channel

- **Command:** `PUBLISH channel message`
- **Description:** A client sends a message to a channel. All subscribers to that channel will receive the message.

```
PUBLISH mychannel "Hello, Subscribers!"
```

Multiple Subscribers to a Channel

- If there are **multiple subscribers** to a channel, they will all receive the message when a publisher sends it.

- **Example:** If two clients subscribe to `mychannel`, both will receive `"Hello, Subscribers!"` when published.

Unsubscribing from a Channel

- **Command:** `UNSUBSCRIBE channel`
- **Description:** A client can unsubscribe from a channel and stop receiving messages from it.

```
UNSUBSCRIBE mychannel
```

Subscribing to Multiple Channels

- If you want to subscribe to a **group of channels**, you can use **pattern matching**.
- **Command:** `PSUBSCRIBE pattern`
- **Description:** Subscribes to all channels that match the given pattern (e.g., all channels starting with “news”).

```
PSUBSCRIBE news*
```

Unsubscribing from Multiple Channels

- **Command:** `PUNSUBSCRIBE pattern`
- **Description:** Unsubscribes from all channels that match the pattern.

```
PUNSUBSCRIBE news*
```

Redis Security

Redis supports basic security features to prevent unauthorized access and restrict harmful operations.

Authentication in Redis

By default, Redis does not require authentication. However, you can enable authentication by setting a password.

1. **Check if authentication is enabled:**

- **Command:** `CONFIG GET requirepass`
- This will return the current password setting.

Set a password for Redis:

1. **Command:** `CONFIG SET requirepass yourpassword`
2. This will enable authentication and require the password for further access.

Authenticate a client:

- After setting a password, any client trying to connect will need to authenticate using the following command:
- **Command:** `AUTH yourpassword`

- If not authenticated, clients will get the error: `NOAUTH Authentication required.`

Restricting Specific Commands

Sometimes, we want to allow users to interact with Redis, but we may want to **restrict** some dangerous commands like `FLUSHALL`, which can delete all data.

1. **Renaming commands to restrict access:**
2. You can rename specific commands to make them harder to execute for unauthorized users.
3. For example, renaming `FLUSHALL` to `PURGEALL` allows only authorized users who know the new name to execute this command.
4. **How to rename a command:**

- Open the Redis configuration file (`redis.conf`) and add the following line to rename a command:

```
rename-command flushall purgeall
```

- This will disable the `FLUSHALL` command and make it available as `PURGEALL` instead.
- **Restart Redis:** After modifying the configuration file, restart the Redis server for the changes to take effect.

Important Redis Security Tips

- **Use strong passwords** to protect your Redis instance.
- **Limit access to Redis** by restricting network access and using firewalls.
- **Rename dangerous commands** like `FLUSHALL`, `FLUSHDB`, `CONFIG`, and `SHUTDOWN` to prevent accidental or malicious data loss.
- **Monitor and log activity** to ensure unauthorized access attempts are detected.

Redis Transactions

Redis transactions allow you to group multiple commands together to be executed atomically. A transaction ensures that all the commands are executed in order, without interruptions. Redis transactions are a form of **Atomicity** in the ACID properties (though Redis transactions are not fully isolated).

Key Commands for Redis Transactions

1. `MULTI`

The `MULTI` command marks the beginning of a transaction in Redis. After calling `MULTI`, all subsequent commands are queued up and not executed immediately.

```
MULTI
```

2. `EXEC`

The `EXEC` command is used to execute all the commands that were queued after the `MULTI` command. Once `EXEC` is executed, all queued commands are

executed in the order in which they were received, and the transaction is complete.

```
EXEC
```

3. DISCARD

If you decide not to execute the queued commands and abort the transaction, you can use the `DISCARD` command. This will discard all the commands that were queued after the `MULTI` command.

```
DISCARD
```

4. `WATCH`

The `WATCH` command is used to monitor one or more keys for changes. If any of the watched keys are modified by another client before executing the transaction, Redis will abort the transaction

```
WATCH key1 key2 ...
```

5. `UNWATCH`

The `UNWATCH` command is used to stop watching the keys that were previously monitored with the `WATCH` command. If you no longer need to watch for changes, you can issue this command.

```
UNWATCH
```

- **Atomicity**: Redis transactions are atomic in that either all commands will be executed, or none will be. However, Redis transactions are not fully isolated, and concurrent modifications by other clients can still affect the state.
- **Concurrency**: Redis transactions allow other clients to modify data between the time the transaction is started (`MULTI`) and the transaction is executed (`EXEC`).
- **Optimistic Concurrency Control**: Redis allows "optimistic" concurrency control using the `WATCH` command. You can monitor the keys for changes and abort the transaction if another client modifies the data.
- **No Rollback**: Redis does not support rolling back a transaction if something goes wrong, except by aborting the transaction entirely.

Redis Pub Sub

R

Written by Rakesh raj

4 Followers · 3 Following

Software developer | Scalable system | HLD & LLD

Follow

No responses yet

Jake

What are your thoughts?

More from Rakesh raj

Rakesh raj

Dependency injection using Google Guice with examples

What is dependency?

Feb 11, 2024 1

Rakesh raj

Stream of bytes internal concept — Java

What is a byte array?

Mar 1, 2024 1

Rakesh raj

Polling vs Long polling — Java

Polling and Long polling is a way to request data from server. It is bidirectional...

Jan 27, 2024

Rakesh raj

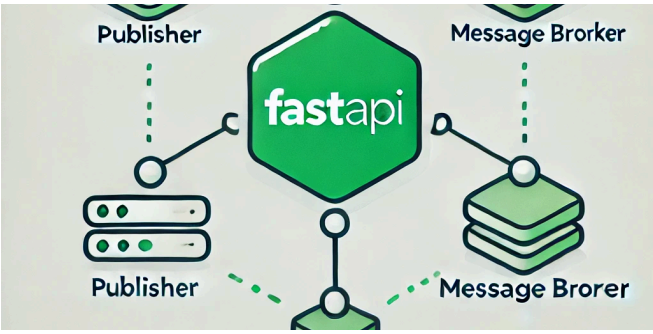
Sever sent events — Java

Server sent events are used to send events in realtime, the communication happens...

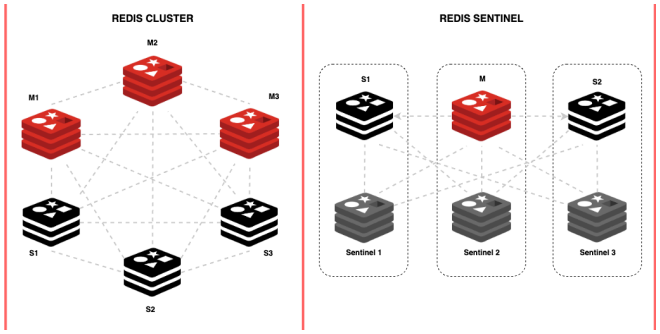
Jan 27, 2024

See all from Rakesh raj

Recommended from Medium



In Mindful Engineering by Rahul Suthar



Praful Khandelwal

Understanding Message Brokers and Message Backends: Redis,...

In this article, we'll explore message queues and message brokers, including some of the...

Dec 17, 2024 386

Understanding Redis High Availability: Cluster vs. Sentinel

Redis is a leading in-memory database that can be used as a key-value store, cache or...

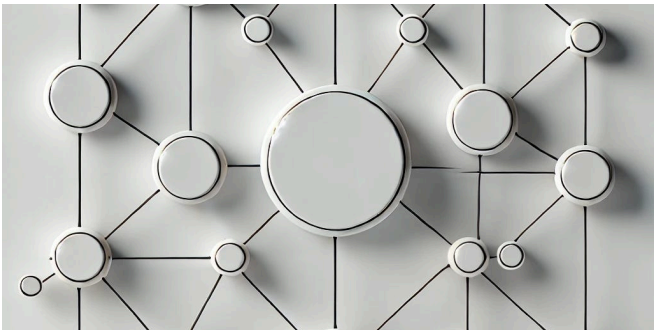
Nov 26, 2024 5

Suganthi

Building a REST API with FastAPI and Redis Caching

When building web applications, optimizing performance is essential, especially for APIs...

Sep 24, 2024 7

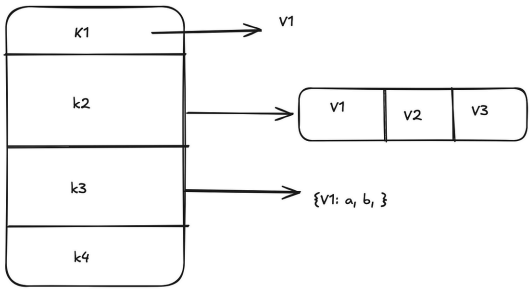


Emmanuel Davidson

RedisGraph

RedisGraph is a graph database module built on top of Redis, designed for storing,...

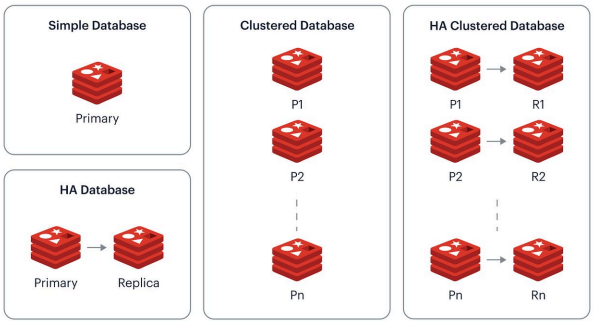
Sep 30, 2024



Parvathi Pai

Redis

Redis is an in-memory, single-threaded data store that primarily utilizes an append-only...



ABHISHEK PANDEY (SUBHAM)

Redis System Design: An In-Depth Technical Analysis

Mastering Redis: From Data Structures to Distributed Systems — All the Secrets to...

Dec 29, 2024 Oct 10, 2024

See more recommendations