Redis Cheatsheet

Redis Cheatsheet

Getting Started

```
# Connect to a local Redis instance
redis-cli

# Connect to a remote Redis instance
redis-cli -h 192.168.1.100 -p 6379 -a mypassword

# Connect with a specific database (0-15)
redis-cli -n 2

# Run a single command
redis-cli PING

# Pipe commands from stdin
echo -e "SET key value\nGET key" | redis-cli

# Run commands from a file
redis-cli --pipe < commands.txt

# Enable raw output (no formatting)
redis-cli --raw GET key

# Monitor all commands in real time
redis-cli MONITOR

# Latency doctor — analyze slow operations
redis-cli --latency-doctor
```

Server Management

```
# Ping the server
PING                          # PONG

# Get server information
INFO
INFO server
INFO memory
```

```
INFO replication
INFO persistence
INFO clients

# Get the number of keys
DBSIZE

# Flush all keys from the current database
FLUSHDB

# Flush all keys from all databases
FLUSHALL

# Get the Unix timestamp of the last DB save
LASTSAVE

# Change database at runtime
SELECT 1

# Debugging — inspect a key's internal encoding
DEBUG OBJECT mykey

# Slow log — find expensive commands
SLOWLOG GET 10                # Last 10 slow entries
SLOWLOG LEN                   # Total entries in slow log
SLOWLOG RESET                 # Clear the slow log

# Client management
CLIENT LIST                   # List all connected clients
CLIENT KILL <ip>:<port>       # Kill a specific client
CLIENT SETNAME myapp          # Name the current connection
CLIENT GETNAME                # Get the current connection name
CLIENT PAUSE 10000            # Pause all clients for 10s (ms)
```

Data Types

Strings

Strings are the most basic Redis type. A single key can hold a string value up to 512 MB. Binary-safe.

```
# Basic CRUD
SET user:1:name "Alice"
GET user:1:name               # "Alice"
DEL user:1:name

# Set with expiration (seconds)
SET session:abc123 "user_data" EX 3600

# Set with expiration (milliseconds)
SET lock:resource "locked" PX 5000

# Set only if key does not exist (NX) or does exist (XX)
SET user:2:name "Bob" NX      # Only set if not already present
SET cache:home "html..." XX   # Only update if already present

# Get and delete atomically
```

```
GETDEL temp:token

# Get and set a new value atomically
GETSET counter:visits 0       # Returns old value, sets new

# Append to a string
APPEND log:entries "line1\n"
APPEND log:entries "line2\n"

# Get substring
GETRANGE log:entries 0 10     # Characters 0 through 10

# Overwrite a substring
SETRANGE greeting 6 "World"   # "Hello World" (if greeting was "Hello ")

# Get string length
STRLEN user:1:name

# Increment / decrement
SET page:views 10
INCR page:views               # 11
INCRBY page:views 5           # 16
DECR page:views               # 15
DECRBY page:views 3           # 12
INCRBYFLOAT balance 0.50      # Floating point increment

# Multi-key operations
MSET user:3:name "Carol" user:3:email "carol@example.com"
MGET user:1:name user:2:name user:3:name
MGETNX key1 key2 key3 val1 val2 val3  # Set multiple only if none exist
```

Lists

Lists are linked lists of strings. Insertion and deletion at the head or tail are O(1).

```
# Push elements
LPUSH queue:tasks "task1" "task2" "task3"   # Push to head (left)
RPUSH queue:tasks "task4"                    # Push to tail (right)

# Pop elements
LPOP queue:tasks                    # Pop from head — "task3"
RPOP queue:tasks                    # Pop from tail — "task4"

# Pop and push (move between lists)
RPOPLPUSH source:queue dest:queue   # Atomically move last element
LMOVE source:queue dest:queue LEFT RIGHT  # Same with direction control (Redis 6.2+)

# Blocking pop (wait if list is empty)
BLPOP queue:tasks 5                 # Block up to 5 seconds
BRPOP queue:tasks 30                # Block up to 30 seconds
BLMOVE source:queue dest:queue LEFT RIGHT 30

# Get elements by index (0-based)
LINDEX queue:tasks 0                # First element
LINDEX queue:tasks -1               # Last element

# Get a range of elements
LRANGE queue:tasks 0 -1             # All elements
```

```
LRANGE queue:tasks 0 4              # First 5 elements

# Get list length
LLEN queue:tasks

# Trim list to a range
LTRIM recent:logs 0 99              # Keep only the newest 100 entries

# Set element at index
LSET queue:tasks 0 "new_task"

# Insert before or after a pivot element
LINSERT mylist BEFORE "pivot" "new_element"
LINSERT mylist AFTER "pivot" "new_element"

# Remove elements by value
LREM mylist -2 "value"              # Remove last 2 occurrences
LREM mylist 2 "value"               # Remove first 2 occurrences
LREM mylist 0 "value"               # Remove all occurrences
```

Sets

Sets are collections of unique, unsorted strings. O(1) add/remove/membership test.

```
# Add / remove members
SADD tags:article:1 "redis" "database" "nosql" "caching"
SREM tags:article:1 "nosql"

# Check membership
SISMEMBER tags:article:1 "redis"     # 1 (true)
SISMEMBER tags:article:1 "mongodb"  # 0 (false)

# Get all members
SMEMBERS tags:article:1

# Get member count
SCARD tags:article:1

# Random member
SRANDMEMBER tags:article:1          # Return a random member (does not remove)
SRANDMEMBER tags:article:1 3        # Return 3 random members

# Pop a random member (removes it)
SPOP tags:article:1

# Move a member between sets
SMOVE source:set dest:set "redis"

# Set operations
SADD set:a 1 2 3 4 5
SADD set:b 4 5 6 7 8
SDIFF set:a set:b                  # Elements in a but not in b → {1, 2, 3}
SINTER set:a set:b                  # Elements in both → {4, 5}
SUNION set:a set:b                  # All elements → {1, 2, 3, 4, 5, 6, 7, 8}

# Store set operation results
SDIFFSTORE result:set set:a set:b
SINTERSTORE result:set set:a set:b
```

```
SUNIONSTORE result:set set:a set:b

# Scan through set members (cursor-based, non-blocking)
SSCAN tags:article:1 0 COUNT 100
```

Sorted Sets (ZSETs)

Sorted sets map unique string members to floating-point scores. Members are ordered by score.

```
# Add members with scores
ZADD leaderboard 100 "player1" 200 "player2" 150 "player3"

# Add or update with options
ZADD leaderboard NX 300 "player4"            # Only add if not exists
ZADD leaderboard XX 250 "player2"            # Only update if exists
ZADD leaderboard CH 50 "player1"             # Return number of changed elements
ZADD leaderboard GT 120 "player1"            # Only update if new score > current

# Get score
ZSCORE leaderboard "player1"                # 100

# Get rank (0-based, by ascending score)
ZRANK leaderboard "player3"                 # 1
ZREVRANK leaderboard "player3"              # 1 (by descending score)

# Count members in a score range
ZCOUNT leaderboard 100 200                  # 3

# Get members by score range
ZRANGE leaderboard 0 -1                     # All members (ascending)
ZRANGEBYSCORE leaderboard 100 200           # Members with score 100-200
ZREVRANGE leaderboard 0 -1                  # All members (descending)
ZREVRANGEBYSCORE leaderboard 200 100        # Members with score 200-100 (desc)

# With scores included
ZRANGE leaderboard 0 -1 WITHSCORES
ZRANGEBYSCORE leaderboard 100 200 WITHSCORES

# Remove members
ZREM leaderboard "player1"
ZREMRANGEBYRANK leaderboard 0 2             # Remove bottom 3
ZREMRANGEBYSCORE leaderboard 0 50           # Remove all with score <= 50

# Increment a member's score
ZINCRBY leaderboard 10 "player1"            # player1 score → 110

# Get cardinality
ZCARD leaderboard

# Union and intersection (aggregate scores)
ZADD daily:2026-04-19 100 "user1" 200 "user2"
ZADD daily:2026-04-20 150 "user1" 300 "user3"
ZUNIONSTORE weekly:scores 2 daily:2026-04-19 daily:2026-04-20 WEIGHTS 1 1 AGGREGATE SUM
ZINTERSTORE common:scores 2 daily:2026-04-19 daily:2026-04-20

# Lexicographic range (when all scores are the same)
ZRANGE lex:set [a [z BYLEX
```

```
# Scan through sorted set members
ZSCAN leaderboard 0 COUNT 100
```

Hashes

Hashes map string fields to string values. Perfect for representing objects.

```
# Set a single field
HSET user:1001 name "Alice"
HSET user:1001 email "alice@example.com"
HSET user:1001 age 30

# Set multiple fields at once
HMSET user:1001 name "Alice" email "alice@example.com" age 30

# Set only if field does not exist
HSETNX user:1001 role "admin"             # Only sets if "role" doesn't exist

# Get a single field
HGET user:1001 name                       # "Alice"

# Get multiple fields
HMGET user:1001 name email age

# Get all fields and values
HGETALL user:1001

# Delete fields
HDEL user:1001 age

# Check if a field exists
HEXISTS user:1001 email                   # 1 (true)

# Get number of fields
HLEN user:1001

# Get all field names
HKEYS user:1001

# Get all values
HVALS user:1001

# Increment a numeric field
HINCRBY user:1001 login_count 1
HINCRBYFLOAT user:1001 balance 10.50

# Scan through hash fields
HSCAN user:1001 0 COUNT 100
```

Streams

Streams are a log data structure with consumer groups. Ideal for event sourcing and messaging.

```
# Add an entry to a stream
XADD events:user:1001 * name "login" ip "10.0.0.1" timestamp 1745145600000
```

```
# Add with a specific ID or auto-generate with *
XADD mystream 0-1 field1 value1 field2 value2

# Cap stream to a maximum number of entries
XADD mystream MAXLEN ~ 10000 * field value   # ~ means approximate trimming

# Read entries from a stream
XREAD COUNT 10 STREAMS mystream 0-0           # All entries
XREAD COUNT 10 STREAMS mystream $             # Only new entries (block)

# Block until new entries arrive
XREAD BLOCK 5000 STREAMS mystream $            # Wait up to 5 seconds
XREAD BLOCK 0 STREAMS mystream $               # Block indefinitely

# Read a range of entries by ID
XRANGE mystream - +                            # All entries
XRANGE mystream 1672531200000-0 1672617600000-0

# Read in reverse
XREVRANGE mystream + -

# Get stream info and length
XINFO STREAM mystream
XLEN mystream

# Trim a stream
XTRIM mystream MAXLEN ~ 1000

# Remove entries
XDEL mystream 1672531200000-0

# Consumer groups
XGROUP CREATE mystream mygroup $ MKSTREAM      # Create group, start from new entries
XGROUP CREATE mystream mygroup 0-0 MKSTREAM   # Create group, read from the beginning
XGROUP DESTROY mystream mygroup                # Delete the group

# Read as a consumer in a group
XREADGROUP GROUP mygroup consumer1 COUNT 5 STREAMS mystream >

# Acknowledge messages
XACK mystream mygroup 1672531200000-0 1672531200000-1

# Claim pending messages (for crashed consumers)
XCLAIM mystream mygroup consumer2 3600000 1672531200000-0 1672531200000-1

# View pending entries
XPENDING mystream mygroup
XPENDING mystream mygroup - + 10              # Detailed pending info

# List consumer groups on a stream
XINFO GROUPS mystream

# List consumers in a group
XINFO CONSUMERS mystream mygroup

# Autoclaim — claim and return pending messages
XAUTOCLAIM mystream mygroup consumer2 3600000 0-0 COUNT 10
```

Bitmaps

Bitmaps are string-based bit arrays. Perfect for flags, presence sets, and analytics.

```
# Set a bit (offset is 0-based)
SETBIT user:1001:logins 0 1                  # Day 0: logged in
SETBIT user:1001:logins 1 1                  # Day 1: logged in
SETBIT user:1001:logins 2 0                  # Day 2: did not log in

# Get a bit
GETBIT user:1001:logins 0                    # 1

# Count set bits
BITCOUNT user:1001:logins                    # Total days logged in
BITCOUNT user:1001:logins 0 1                # Count bits in byte range 0-1

# Bitwise operations (store result in destkey)
SETBIT flags:a 0 1
SETBIT flags:a 3 1
SETBIT flags:b 1 1
SETBIT flags:b 3 1
BITOP AND result:flags flags:a flags:b       # {0,1,0,1} AND {0,1,0,1} → {0,1,0,1}
BITOP OR result:flags flags:a flags:b
BITOP XOR result:flags flags:a flags:b
BITOP NOT result:flags flags:a

# Find first bit set
BITPOS user:1001:logins 1                    # First day with login
BITPOS user:1001:logins 0                    # First day without login

# Real-world: tracking daily active users for a given date
SETBIT active:2026-04-20 <user_id> 1
BITCOUNT active:2026-04-20                   # Total active users that day
```

HyperLogLog

HyperLogLog provides probabilistic cardinality estimation with ~0.81% standard error using minimal memory (12 KB per key).

```
# Add elements
PFADD unique:visitors:page:home "user1" "user2" "user3" "user1"

# Get estimated cardinality
PFCOUNT unique:visitors:page:home            # 3 (estimated)

# Merge multiple HLLs
PFADD unique:visitors:day1 "user1" "user2"
PFADD unique:visitors:day2 "user2" "user3"
PFMERGE unique:visitors:combined unique:visitors:day1 unique:visitors:day2
PFCOUNT unique:visitors:combined             # ~3

# Real-world: monthly unique page views
PFADD uv:2026-04-20 "user1" "user2" "user3"
PFADD uv:2026-04-21 "user2" "user4" "user5"
PFMERGE uv:2026-04 uv:2026-04-20 uv:2026-04-21
PFCOUNT uv:2026-04                           # ~5 unique users
```

Key Management

Expiration and TTL

```
# Set expiration (seconds)
EXPIRE session:abc 3600                      # Expire in 1 hour
EXPIREAT session:abc 1745232000              # Expire at Unix timestamp

# Set expiration (milliseconds)
PEXPIRE session:abc 3600000
PEXPIREAT session:abc 1745232000000

# Get remaining time to live
TTL session:abc                              # Seconds remaining (-1 = no expiry, -2 = key go
PTTL session:abc                             # Milliseconds remaining

# Persist a key (remove expiration)
PERSIST session:abc

# Set with built-in expiration
SET cache:product:42 "data" EX 300           # Expire in 5 minutes
SET lock:order:99 "held" PX 10000            # Expire in 10 seconds
```

Key Scanning and Pattern Matching

```
# KEYS pattern (WARNING: blocks the server, use in dev only)
KEYS user:*                                  # All keys starting with "user:"
KEYS session:???:*                           # 3-char ID after "session:"

# SCAN — cursor-based, non-blocking alternative
SCAN 0 MATCH user:* COUNT 100                # First batch
SCAN 34 MATCH user:* COUNT 100               # Next batch (cursor = 34)
SCAN 0 MATCH session:* COUNT 100 COUNT 1000  # Larger batches

# Type inspection
TYPE mykey                                   # string, list, set, zset, hash, stream, etc.

# Check existence
EXISTS mykey                                 # 1 if exists, 0 if not
EXISTS key1 key2 key3                        # Count of existing keys

# Rename
RENAME oldkey newkey                         # Overwrites newkey if it exists
RENAMENX oldkey newkey                       # Only rename if newkey doesn't exist

# Copy a key (Redis 6.2+)
COPY sourcekey destkey

# Dump and restore (binary serialization)
DUMP mykey
RESTORE newkey 0 "<serialized-data>"

# Serialize key to see encoding
DEBUG OBJECT mykey

# Sort keys (works on lists, sets, sorted sets)
```

```
SORT mylist ALPHA
SORT mylist BY weight:* DESC LIMIT 0 10
SORT mylist BY weight:* DESC GET object:*->name
```

Transactions

Redis transactions group multiple commands into a single atomic operation. All commands are queued and executed sequentially.

```
# Basic transaction
MULTI
SET user:1:balance 1000
DECRBY user:1:balance 200
INCRBY user:2:balance 200
EXEC

# Discard a transaction
MULTI
SET key1 value1
DISCARD                                  # Cancels the transaction

# Optimistic locking with WATCH
WATCH account:1001                       # Watch key for changes
balance = GET account:1001               # Read current balance
MULTI
SET account:1001 <new_balance>
EXEC                                      # Succeeds only if account:1001 wasn't changed

# UNWATCH — cancel all watched keys
UNWATCH
```

Transaction Gotchas

```
# Errors inside a transaction: syntax errors cause EXEC to fail entirely,
# but type errors (e.g., INCR on a string) are skipped silently.

MULTI
SET foo bar
INCR foo                                 # Type error: foo is a string
SET baz qux
EXEC                                      # foo="bar", INCR is skipped, baz="qux"

# Redis does NOT support rollback — handle errors in your application.
```

Pub/Sub

Redis publish/subscribe enables real-time messaging between clients.

```
# Subscribe to channels
SUBSCRIBE channel:notifications
SUBSCRIBE channel:notifications channel:alerts channel:logs

# Subscribe to a pattern
```

```
PSUBSCRIBE channel:user:*                    # Matches channel:user:1001, etc.

# Publish a message
PUBLISH channel:notifications "Order #1234 has been shipped"

# Unsubscribe
UNSUBSCRIBE channel:notifications
PUNSUBSCRIBE channel:user:*

# List active channels
PUBSUB CHANNELS                              # All active channels
PUBSUB CHANNELS channel:user:*               # Channels matching pattern
PUBSUB NUMSUB channel:notifications          # Subscriber count
PUBSUB NUMPAT                                # Number of pattern subscriptions
```

Pub/Sub Limitations

- Messages are **fire-and-forget**: if no subscriber is listening, the message is lost.
- Pub/Sub does **not persist** messages — use Streams for durable messaging.
- Subscribers only receive messages published **after** they subscribe.

Lua Scripting

Lua scripts execute atomically on the server, ensuring no other commands run during execution.

```
# EVAL — run a Lua script
EVAL "return redis.call('SET', KEYS[1], ARGV[1])" 1 mykey myvalue

# EVALSHA — run by SHA1 digest (cached on server)
SCRIPT LOAD "return redis.call('SET', KEYS[1], ARGV[1])"
# Returns: sha1_digest
EVALSHA sha1_digest 1 mykey myvalue

# Script management
SCRIPT EXISTS <sha1> <sha2>                  # Check if scripts are cached
SCRIPT FLUSH                                 # Remove all cached scripts
SCRIPT KILL                                  # Kill a currently running script

# Example: atomic rate limiter
EVAL "
local current = redis.call('GET', KEYS[1])
if current and tonumber(current) >= tonumber(ARGV[1]) then
    return 0
end
redis.call('INCR', KEYS[1])
redis.call('EXPIRE', KEYS[1], ARGV[2])
return 1
" 1 rate:user:1001 10 60
```

Lua Scripting Rules

- All Redis commands executed inside a script are **atomic**.
- Lua scripts should be **fast** — they block the server while running.
- Use `redis.call()` for commands and `redis.pcall()` for non-throwing calls.

- Scripts are **read-only** by default on replicas; use `redis.replicate_commands()` for write scripts.
- Use `SCRIPT LOAD` + `EVALSHA` in production to avoid transmitting the script body on every call.

Persistence

RDB (Redis Database) Snapshots

Point-in-time snapshots of the dataset stored as binary `.rdb` files.

```
# redis.conf — RDB settings
save 900 1            # Save after 900 seconds if at least 1 key changed
save 300 10           # Save after 300 seconds if at least 10 keys changed
save 60 10000         # Save after 60 seconds if at least 10000 keys changed

save ""               # Disable RDB saves entirely

rdbcompression yes    # Compress RDB files with LZF
rdbchecksum yes       # Append CRC64 checksum
dbfilename dump.rdb   # RDB filename
dir /var/lib/redis    # RDB directory

# Manual save
redis-cli BGSAVE      # Fork a child process to save (non-blocking)
redis-cli SAVE        # Block until save completes (avoid in production)
```

AOF (Append Only File)

Logs every write operation sequentially. More durable than RDB but larger files.

```
# redis.conf — AOF settings
appendonly yes                    # Enable AOF
appendfilename "appendonly.aof"   # AOF filename
appendfsync everysec              # Sync once per second (recommended)
# appendfsync always               # Sync every write (safest, slowest)
# appendfsync no                   # Let OS decide (fastest, least safe)

auto-aof-rewrite-percentage 100   # Rewrite when AOF is 100% larger than last rewrite
auto-aof-rewrite-min-size 64mb    # Minimum size to trigger rewrite

no-appendfsync-on-rewrite no      # Continue fsync during rewrite

# Manual rewrite
redis-cli BGREWRITEAOF
```

RDB vs AOF Comparison

Feature	RDB	AOF
Durability	Point-in-time snapshots	Every write logged
File size	Compact binary	Larger, text-based
Recovery speed	Fast	Slower
Write performance	Lower impact on writes	Can impact write latency

Feature	RDB	AOF
Data loss window	Minutes (based on save rules)	~1 second (everysec)
Best for	Backups, disaster recovery	Production durability

Mixed Persistence (Redis 4.0+)

```
# Use both RDB and AOF together
# RDB for base file + AOF incremental changes
aof-use-rdb-preamble yes
```

Replication

Redis replication creates one or more read replicas of a master node.

```
# On the replica — connect to a master
redis-cli REPLICAOF 192.168.1.100 6379

# On a replica — promote to master
redis-cli REPLICAOF NO ONE

# Check replication status
redis-cli INFO replication

# Configure master password (if required)
masterauth <password>

# Replica read-only (default: yes)
replica-read-only yes

# Delayed replication (for disaster recovery)
replica-serve-stale-data yes
repl-backlog-size 1mb
```

Replication Architecture

```
# redis.conf on master
bind 0.0.0.0
protected-mode no
requirepass your_strong_password
masterauth your_strong_password

# redis.conf on replica
replicaof <master_ip> <master_port>
masterauth your_strong_password
replica-read-only yes

# Partial resynchronization (PSYNC)
# Replicas maintain a replication backlog to handle reconnections
repl-backlog-size 10mb
repl-backlog-ttl 3600
```

Sentinel

Redis Sentinel provides automatic failover, monitoring, and notification for Redis masters.

```
# sentinel.conf (minimum viable config)
sentinel monitor mymaster 127.0.0.1 6379 2     # Name, host, port, quorum
sentinel down-after-milliseconds mymaster 5000  # Mark master down after 5s
sentinel failover-timeout mymaster 60000        # Failover timeout: 60s
sentinel parallel-syncs mymaster 1              # Replicate 1 replica at a time

# Start Sentinel
redis-sentinel /path/to/sentinel.conf

# Sentinel commands
redis-cli -p 26379 SENTINEL masters                    # List monitored masters
redis-cli -p 26379 SENTINEL get-master-addr-by-name mymaster  # Current master
redis-cli -p 26379 SENTINEL replicas mymaster           # List replicas
redis-cli -p 26379 SENTINEL ckquorum mymaster           # Check quorum
redis-cli -p 26379 SENTINEL failover mymaster           # Force failover
```

Sentinel Client Connection

```
# Applications should connect through Sentinel to discover the master
# Sentinel returns the current master's IP and port dynamically
redis-cli -p 26379 SENTINEL get-master-addr-by-name mymaster
# 1) "192.168.1.100"
# 2) "6379"
```

Clustering

Redis Cluster shards data automatically across multiple nodes (minimum 3 masters + 3 replicas for production).

Cluster Setup

```
# Create a cluster from nodes
redis-cli --cluster create \
  192.168.1.101:6379 \
  192.168.1.102:6379 \
  192.168.1.103:6379 \
  192.168.1.104:6379 \
  192.168.1.105:6379 \
  192.168.1.106:6379 \
  --cluster-replicas 1

# Add a new node
redis-cli --cluster add-node 192.168.1.107:6379 192.168.1.101:6379

# Add a node as a replica
redis-cli --cluster add-node \
  192.168.1.108:6379 192.168.1.101:6379 \
  --cluster-slave

# Rebalance slots across nodes
```

```
redis-cli --cluster rebalance 192.168.1.101:6379

# Remove a node
redis-cli --cluster del-node 192.168.1.107:6379 <node-id>

# Check cluster state
redis-cli --cluster check 192.168.1.101:6379

# Cluster info
redis-cli CLUSTER INFO
redis-cli CLUSTER NODES
redis-cli CLUSTER SLOTS
redis-cli CLUSTER KEYSLOT "mykey"           # Which slot a key hashes to
```

Cluster Configuration

```
# redis.conf for cluster mode
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
cluster-announce-ip 192.168.1.101
cluster-announce-port 6379

# Hash tags — force related keys to the same slot
# Keys inside {} use only the content inside braces for hashing
SET {user:1001}:name "Alice"
SET {user:1001}:email "alice@example.com"
SET {user:1001}:age 30
# All three keys hash to the same slot → can be used in multi-key operations
```

Cluster Limitations

- **No cross-slot multi-key operations** (use hash tags `{...}` to group related keys).
- **Transactions (MULTI/EXEC)** only work on keys in the same slot.
- **Pub/Sub** messages are broadcast to all nodes; subscribers must connect to the correct node.
- **Lua scripts** must declare all keys accessed via `redis.call()` so Redis can verify they're on the same slot.
- **Database count is fixed** at 16 databases (but cluster mode only supports database 0).

Memory Management

```
# Memory info
redis-cli INFO memory

# Max memory policy
maxmemory 2gb
maxmemory-policy allkeys-lru     # Evict least recently used keys
# maxmemory-policy volatile-lru   # Evict LRU among keys with TTL set
# maxmemory-policy allkeys-lfu    # Evict least frequently used keys (Redis 4.0+)
# maxmemory-policy volatile-lfu   # Evict LFU among keys with TTL set
# maxmemory-policy allkeys-random # Evict random keys
# maxmemory-policy volatile-random # Evict random keys with TTL set
# maxmemory-policy volatile-ttl   # Evict keys with shortest TTL first
# maxmemory-policy noeviction     # Return errors when memory limit reached
```

```
# Sampled LRU — Redis uses an approximation of LRU for efficiency
# It samples N keys and evicts the least recently used among the sample
maxmemory-samples 5              # Default: 5 (higher = more accurate, slower)

# LFU tuning (Redis 4.0+)
lfu-log-factor 10                # Counter logarithmic factor (higher = slower decay)
lfu-decay-time 1                 # Decay time in minutes

# Inspect memory usage of a key
redis-cli MEMORY USAGE mykey

# Memory allocator stats
redis-cli MEMORY STATS

# Memory doctor — get optimization advice
redis-cli MEMORY DOCTOR

# Purge expired keys (usually automatic)
redis-cli --scan | xargs -L 1000 redis-cli UNLINK
```

Memory Optimization Tips

```
# Use hashes to save memory for related fields
# Small hashes (< hash-max-ziplist-entries) are encoded as ziplists
# 100 hashes with 1 field each → ~100 keys (wasteful)
# vs 1 hash with 100 fields → compact ziplist encoding

# Configure encoding thresholds
hash-max-ziplist-entries 512
hash-max-ziplist-value 64
list-max-ziplist-size -2
set-max-intset-entries 512
zset-max-ziplist-entries 128
zset-max-ziplist-value 64
```

Connection Management

```
# Connection pool settings (in redis.conf)
maxclients 10000                    # Max simultaneous clients
timeout 300                         # Close idle clients after 300 seconds
tcp-keepalive 60                    # TCP keepalive interval

# TLS (Redis 6.0+)
tls-port 6379
tls-cert-file /path/to/redis.crt
tls-key-file /path/to/redis.key
tls-ca-cert-file /path/to/ca.crt
tls-auth-clients optional
redis-cli --tls --cert ./client.crt --key ./client.key --cacert ./ca.crt

# Unix socket (faster than TCP for local connections)
unixsocket /var/run/redis/redis.sock
unixsocketperm 700
redis-cli -s /var/run/redis/redis.sock
```

```
# Authentication
requirepass your_strong_password
redis-cli -a your_strong_password    # Connect with password
CONFIG SET requirepass "newpass"     # Change password at runtime
CONFIG GET requirepass               # Retrieve password (shows as asterisks in logs)

# ACL (Access Control Lists — Redis 6.0+)
ACL SETUSER developer on +@all -@dangerous >dev_password ~*
ACL SETUSER readonly on +@read ~* >readonly_pass
ACL SETUSER appuser on +@all -DEBUG -CONFIG -SHUTDOWN +SET +GET +DEL ~app:*
ACL LIST                             # List all ACL rules
ACL WHOAMI                           # Show current user
ACL CAT                              # List all command categories
ACL GENPASS                          # Generate a random password
ACL DELUSER developer                # Delete a user
```

Common Patterns

Rate Limiting

```
# Fixed-window rate limiter using INCR + EXPIRE
# Allow 10 requests per minute per user

# Lua script for atomicity
EVAL "
local key = KEYS[1]
local limit = tonumber(ARGV[1])
local window = tonumber(ARGV[2])
local current = redis.call('INCR', key)
if current == 1 then
    redis.call('EXPIRE', key, window)
end
if current > limit then
    return 0
end
return current
" 1 rate:user:1001 10 60

# Sliding-window rate limiter using sorted sets
# Allow 100 requests per hour per IP
ZADD rate:ip:10.0.0.1 <timestamp_ms> <unique_id>
ZREMRANGEBYSCORE rate:ip:10.0.0.1 -inf <timestamp_ms - 3600000>
ZCARD rate:ip:10.0.0.1
# If ZCARD >= 100 → reject
```

Caching

```
# Cache-aside pattern (application-side)
# 1. Try cache first
GET cache:user:1001
# 2. If miss, fetch from database, then:
SET cache:user:1001 "<json_data>" EX 300

# Cache with refresh-ahead (renew before expiry)
```

```
GET cache:product:42
# If TTL < 60 seconds, trigger async refresh
TTL cache:product:42

# Cache invalidation on write
SET cache:user:1001 "<new_data>" EX 300     # Write-through
# Or on update:
DEL cache:user:1001                          # Invalidate

# Distributed lock with SET NX
SET lock:order:99 "unique_token" NX PX 10000
# Returns OK if lock acquired, nil if already held
# Release with a Lua script (verify ownership):
EVAL "
if redis.call('GET', KEYS[1]) == ARGV[1] then
    return redis.call('DEL', KEYS[1])
else
    return 0
end
" 1 lock:order:99 unique_token
```

Leaderboards

```
# Add or update a player's score
ZADD leaderboard:global 1500 "player:1001"
ZINCRBY leaderboard:global 50 "player:1001"     # +50 points

# Top N players (highest scores first)
ZREVRANGE leaderboard:global 0 9 WITHSCORES     # Top 10
ZREVRANGE leaderboard:global 0 99 WITHSCORES    # Top 100

# Get a player's rank
ZREVRANK leaderboard:global "player:1001"        # 0-based rank
ZSCORE leaderboard:global "player:1001"          # 1550

# Players near a given rank (for pagination)
# Get the score of the player at the boundary, then:
ZREVRANGEBYSCORE leaderboard:global 1550 1400 WITHSCORES

# Top N players within a specific range
ZREVRANGEBYSCORE leaderboard:global 2000 1500 WITHSCORES LIMIT 0 10

# Remove a player
ZREM leaderboard:global "player:1001"
```

Job Queues

```
# Simple FIFO queue with lists
RPUSH queue:emails "send_welcome user@example.com"
RPUSH queue:emails "send_invoice order:99"
BLPOP queue:emails 30                          # Worker: block up to 30s for next job

# Reliable queue with sorted sets (priority queue)
ZADD queue:jobs <priority> <job_id>
# Worker: claim the highest-priority job
ZPOPMIN queue:jobs                              # Pop lowest score (highest priority if lower
```

```
# Delayed queue
ZADD queue:delayed <execute_at_timestamp> <job_id>
# Worker: poll for due jobs
ZRANGEBYSCORE queue:delayed -inf <current_timestamp> LIMIT 0 10
ZREM queue:delayed <job_id>

# Reliable processing with Streams (preferred for production)
XADD stream:orders * event "created" order_id "1234" customer "1001"
XREADGROUP GROUP workers worker-1 COUNT 1 BLOCK 5000 STREAMS stream:orders >
# Process job...
XACK stream:orders workers <message_id>
```

Distributed Counters

```
# Page view counter
INCR page:views:/about
GET page:views:/about

# Daily counter with auto-expiry
INCR stats:pageviews:2026-04-20
EXPIRE stats:pageviews:2026-04-20 172800       # Expire in 2 days

# Per-user action counter
HINCRBY user:1001:stats logins 1
HINCRBY user:1001:stats purchases 1
HGET user:1001:stats logins
```

Configuration

```
# Runtime configuration
CONFIG GET maxmemory                          # Get a config value
CONFIG GET "*"                                # Get all config
CONFIG SET maxmemory 2gb                      # Set a config value at runtime
CONFIG REWRITE                                # Persist runtime config to redis.conf

# Common configuration
bind 127.0.0.1                                # Listen on specific interfaces
port 6379                                     # Listen port
daemonize yes                                 # Run as daemon
pidfile /var/run/redis/redis-server.pid
logfile /var/log/redis/redis.log
loglevel notice                               # debug, verbose, notice, warning
databases 16                                  # Number of databases (default: 16)

# Slow log
slowlog-log-slower-than 10000                 # Log commands slower than 10ms
slowlog-max-len 128                           # Keep last 128 slow entries
```

Best Practices

1. **Use key naming conventions** with colons as separators: `object-type:id:field` (e.g., `user:1001:profile`).

2. **Always set TTLs** on cache keys to prevent memory leaks from stale data.
3. **Use `SCAN` instead of `KEYS`** in production — `KEYS` blocks the server.
4. **Use connection pooling** in application clients to avoid connection overhead.
5. **Enable `appendonly yes`** for durability in production, or use mixed persistence.
6. **Use `UNLINK` instead of `DEL`** for large keys — it deletes asynchronously in a background thread.
7. **Use Lua scripts** for atomic multi-step operations (rate limiting, distributed locks).
8. **Use hash tags `{...}`** in cluster mode to keep related keys on the same slot.
9. **Set `maxmemory-policy`** to `allkeys-lru` or `volatile-lru` to handle memory pressure gracefully.
10. **Use Sentinel or Redis Cluster** for high availability — never rely on a single Redis instance in production.
11. **Use ACLs (Redis 6.0+)** instead of a single `requirepass` for fine-grained access control.
12. **Monitor with `INFO`**, `SLOWLOG`, and `MEMORY DOCTOR` regularly in production.

In-Memory Data Structure Store Reference