CODING INDIA PUBLICATIONS

REDIS

The Complete Resource Book

From Your First Command to Production-Grade Systems

Every data type • Every core command • Version history from pre-3.2 to Redis 8
Django & FastAPI integration guides • 50 interview questions with answers

A Coding India Publication • 2026

Redis — The Complete Resource Book

First Edition, 2026

Published by Coding India

codingindia.co.in

This book is intended for learners, working developers, and interview candidates. All commands were written for Redis 7.x/8.x with version notes marking anything older or newer. Code samples may be freely used in your own projects.

Dedication

To my mother and father.

You never sat in a classroom long enough to read the books I read,
yet you understood the value of education better than anyone who did.

Papa, you spent your years behind a steering wheel so that I could spend mine behind a keyboard.
Every long day you drove, every comfort you gave up, every rupee you set aside —
it all lives inside every line of code I write.

You could not be given an education, so you gave me one instead.
A driver's son became a software engineer. That is your achievement before it is mine.

This book, and everything I build, is yours.

Table of Contents

(In Microsoft Word: right-click the list above → Update Field → Update entire table, to fill in the exact page numbers.)

Chapter 1 — What is Redis and Why Does Everyone Use It?

Redis stands for **REmote DIctionary Server**. It was created in 2009 by Salvatore Sanfilippo (known online as *antirez*). In the simplest words: Redis is a database that keeps all of its data in **RAM (memory)** instead of on a hard disk.

Why does that matter? Think of it like this. A normal database (MySQL, PostgreSQL) is like a big library. When you ask for a book, a librarian walks to the shelf, finds it, and brings it back. That walk takes time. Redis is like having the book already open on your desk. You just look down and read it.

How fast is it, really?

- A disk read takes a few **milliseconds** (thousandths of a second).
- A RAM read takes **nanoseconds to microseconds** (millionths of a second).
- A typical Redis GET command finishes in about **50–100 microseconds**.
- One Redis server on normal hardware can handle **100,000+ operations per second**.

The three design decisions that define Redis

1. Everything lives in RAM. Data is stored in memory. Disk is only used for backup (persistence), which we cover in Chapter 12. This is the source of Redis's speed and also its main cost: RAM is expensive, so you keep only hot, frequently-used data in Redis.

2. Commands run one at a time (single-threaded execution). Redis executes one command at a time, in order. This sounds slow, but it removes all locking and race-condition problems. Two clients incrementing the same counter can never corrupt it. Since Redis 6.0, network reading/writing uses multiple threads, but the actual command execution is still a single thread.

3. Simple text protocol (RESP). Clients talk to Redis using a simple protocol called RESP (REdis Serialization Protocol). It is so simple you can connect with telnet and type commands by hand.

What is Redis used for?

Use case	How Redis is used
Caching	Store results of slow database queries so repeated requests are instant
Session store	Keep logged-in user sessions for web apps
Queues / background jobs	Celery, RQ, Sidekiq all use Redis as their

Use case	How Redis is used
	job queue
Leaderboards / ranking	Sorted Sets keep scores automatically ordered
Rate limiting	Count requests per user per minute, block abusers
Real-time messaging	Pub/Sub for chat, notifications, live dashboards
Distributed locks	Make sure only one server runs a critical task at a time
Counting things	Page views, likes, unique visitors (HyperLogLog)

Redis vs a normal SQL database

	Redis	PostgreSQL / MySQL
Data lives in	RAM (disk only for backup)	Disk (RAM only for cache)
Speed	Microseconds	Milliseconds
Data model	Key → value (with rich value types)	Tables, rows, columns, joins
Query language	Simple commands (GET, SET…)	SQL
Best for	Hot data, speed-critical paths	Source of truth, complex queries
Durability	Configurable (may lose ~1s)	Fully durable by default

> 📌 Important mindset: Redis usually does not REPLACE your main database. It sits IN FRONT of it. PostgreSQL remains the source of truth; Redis serves the hot data fast.

Chapter 2 — Installing Redis and First Steps

Install with Docker (recommended)

```
docker run -d --name redis -p 6379:6379 redis:7-alpine

# open the Redis command line
docker exec -it redis redis-cli
```

Install on Ubuntu / Debian

```
sudo apt update
sudo apt install redis-server
sudo systemctl enable redis-server
sudo systemctl start redis-server
redis-cli ping     # should reply: PONG
```

Install on Windows

Redis does not officially support Windows. Use one of these: (1) WSL2 (Windows Subsystem for Linux) and install as on Ubuntu, or (2) Docker Desktop, or (3) Memurai (a third-party Windows port). WSL2 is the cleanest option for developers.

Your first session

```
127.0.0.1:6379> PING
PONG
127.0.0.1:6379> SET name "Coding India"
OK
127.0.0.1:6379> GET name
"Coding India"
127.0.0.1:6379> DEL name
(integer) 1
127.0.0.1:6379> GET name
(nil)
```

Three things to notice: **OK** means the write succeeded, **(integer) 1** means one key was deleted, and **(nil)** means the key does not exist. Redis never throws an error for a missing key on GET — it simply returns nil.

The configuration file (redis.conf)

Redis reads a file called redis.conf at startup. The settings you will touch most often:

Setting	What it does	Common production value
bind	Which network interfaces Redis listens on	127.0.0.1 or private IP

Setting	What it does	Common production value
port	TCP port	6379
requirepass	Password for clients	a long random string
maxmemory	RAM limit	e.g. 2gb
maxmemory-policy	What to evict when full (Ch. 13)	allkeys-lfu
appendonly	Turn on AOF persistence (Ch. 12)	yes
save	RDB snapshot schedule (Ch. 12)	3600 1 300 100 60 10000

You can also change settings live without restart:

```
CONFIG GET maxmemory
CONFIG SET maxmemory 2gb
CONFIG REWRITE     # write current config back to redis.conf
```

Useful redis-cli tricks

```
redis-cli --stat            # live stats every second
redis-cli --bigkeys         # find the largest keys
redis-cli --latency         # measure latency to the server
redis-cli MONITOR           # print every command hitting the server (NEVER
on busy prod!)
redis-cli -n 3 GET key      # talk to database number 3
```

📌 Redis has 16 numbered databases (0–15) by default, selected with SELECT n. This feature is considered legacy — Redis Cluster does not support it. Modern practice: use one database and separate data with key prefixes like app1:user:1.

Chapter 3 — Keys, Expiry (TTL) and Key Management

How to name keys

A Redis key is just a string (technically it can be any binary data up to 512 MB, but keep keys short). The universal convention is colon-separated namespaces:

```
user:1001                 # hash with user 1001's profile
user:1001:followers       # set of follower IDs
session:a94c8fe5          # session data
cache:api:/products?p=2   # cached API response
rate:login:192.168.1.5    # rate-limit counter per IP
```

Good key design is half of good Redis design. Keys should be predictable (you can construct them in code without searching) and self-describing (a human reading the key knows what it holds).

Basic key commands

```
EXISTS user:1001          # 1 if exists, 0 if not
TYPE user:1001            # string / list / hash / set / zset / stream
DEL user:1001             # delete, blocking
UNLINK user:1001          # delete in background thread (Redis 4.0+)
RENAME oldkey newkey
RANDOMKEY                 # a random key from the database
DBSIZE                    # how many keys exist
```

📌 Version note: UNLINK arrived in Redis 4.0. Before 4.0 you only had DEL, and deleting a huge key (say a list with 10 million items) would freeze the whole server while memory was reclaimed. UNLINK hands the memory-freeing work to a background thread. On modern Redis, prefer UNLINK for anything possibly large.

TTL — making keys expire automatically

Any key can be given a lifetime. When it runs out, Redis deletes the key by itself. This is the feature that makes Redis a great cache.

```
SET session:abc "data" EX 3600      # expire in 3600 seconds (1 hour)
SET session:abc "data" PX 5000      # expire in 5000 milliseconds
EXPIRE session:abc 600               # set/replace TTL on existing key
PEXPIRE session:abc 60000            # same but in milliseconds
EXPIREAT session:abc 1767225600      # expire at a Unix timestamp
TTL session:abc                      # seconds remaining (-1 = no expiry, -2
= key gone)
PTTL session:abc                     # milliseconds remaining
PERSIST session:abc                  # remove the expiry, key lives forever
```

How Redis actually expires keys (interview favourite)

Redis does NOT set a timer per key. That would be too expensive with millions of keys. Instead it uses two mechanisms working together:

- **Lazy expiry:** when a client reads a key, Redis first checks its TTL. If expired, Redis deletes it and answers as if it never existed.
- **Active expiry:** roughly 10 times per second, Redis samples 20 random keys that have TTLs. Expired ones are deleted. If more than 25% of the sample was expired, it immediately samples again. This keeps expired-but-not-yet-deleted keys to a small fraction of memory.

Finding keys: KEYS vs SCAN

KEYS pattern returns every matching key — but it scans the ENTIRE keyspace in one blocking operation. On a database with 10 million keys, KEYS * freezes Redis for seconds. Every other client waits. This is one of the most common causes of production incidents.

```
# NEVER do this in production:
KEYS user:*

# Do this instead — cursor-based, non-blocking:
SCAN 0 MATCH user:* COUNT 1000
# returns a cursor + a batch of keys; repeat with the returned
# cursor until it comes back as 0
```

> 📌 Version note: SCAN (and HSCAN, SSCAN, ZSCAN for scanning inside big collections) was added in Redis 2.8. Before that, KEYS was the only option — one reason older Redis deployments had a reputation for mysterious freezes.

Chapter 4 — Strings: The Foundation Type

A Redis string is not just text. It is any sequence of bytes up to 512 MB: plain text, a number, JSON, or even an image. Strings are the type you will use most.

Set and get

```
SET user:1:name "Digamber"
GET user:1:name                # "Digamber"
SET counter 10
GETRANGE user:1:name 0 3       # "Diga" (substring)
STRLEN user:1:name             # 8
APPEND user:1:name " Jha"      # now "Digamber Jha"
MSET a 1 b 2 c 3               # set many at once
MGET a b c                     # get many at once
```

The powerful options of SET

```
SET key value EX 60        # with 60s expiry
SET key value PX 60000     # with 60000ms expiry
SET key value NX           # only set if key does NOT exist
SET key value XX           # only set if key DOES exist
SET key value KEEPTTL      # change value but keep old TTL (Redis 6.0+)
SET key value GET          # set new value, return the OLD one (Redis 6.2+)
```

SET ... NX is the single most important option in Redis. It is atomic: if two clients race, exactly one wins. This is the building block for distributed locks (Chapter 17).

> 📌 Version note: before Redis 2.6.12 you needed separate commands SETNX (set if not exists), SETEX (set with expiry) and PSETEX. Modern Redis folds all of these into options on SET, and the old commands are considered deprecated (they still work). GETSET is likewise replaced by SET ... GET, and GETDEL / GETEX were added in 6.2.

Atomic counters

```
SET views 0
INCR views          # 1
INCR views          # 2
INCRBY views 10     # 12
DECR views          # 11
DECRBY views 5      # 6
INCRBYFLOAT price 0.5   # works with decimals
```

INCR is atomic. If 1,000 clients call INCR at the same moment, the final value is exactly 1,000 — never less. No locks needed. This is why Redis is the standard tool for counting views, likes, and API requests.

Real example: simple page-view counter with daily reset

```
# key contains the date, so a new day = a new key
INCR views:home:2026-07-08
EXPIRE views:home:2026-07-08 172800   # keep 2 days, auto-clean
```

Chapter 5 — Lists: Ordered Data and Queues

A Redis list is an ordered sequence of strings. You can push and pop from both the **left** (head) and the **right** (tail) very fast. Reading by index in the middle is slower (the list is a linked structure, not an array).

Core commands

```
LPUSH tasks "task1"        # push to the LEFT:  [task1]
LPUSH tasks "task2"        # [task2, task1]
RPUSH tasks "task3"        # push to the RIGHT: [task2, task1, task3]
LRANGE tasks 0 -1          # read all: 0 = first, -1 = last
LLEN tasks                 # 3
LPOP tasks                 # remove+return "task2"
RPOP tasks                 # remove+return "task3"
LINDEX tasks 0             # read by position without removing
LINSERT tasks BEFORE "task1" "task0"
LREM tasks 1 "task1"       # remove 1 occurrence of value
LTRIM tasks 0 99           # keep only first 100 elements
```

Pattern 1: Recent activity feed (capped list)

```
LPUSH feed:user:1001 "liked post 42"
LTRIM feed:user:1001 0 49      # always keep only the 50 newest
LRANGE feed:user:1001 0 9      # show the 10 most recent
```

Pattern 2: A job queue (how Celery works underneath)

Producer pushes jobs on one side; workers pop from the other. The magic command is **BRPOP** — *blocking* pop. If the list is empty, the worker's connection simply waits (no busy polling) until a job arrives.

```
# producer
LPUSH queue:emails '{"to": "a@b.com", "subject": "Hi"}'

# worker (blocks up to 30s waiting for work; 0 = wait forever)
BRPOP queue:emails 30
```

For a **reliable** queue (job must not be lost if the worker crashes mid-processing), use LMOVE to atomically move the job to a per-worker 'processing' list, and remove it only after success:

```
LMOVE queue:emails processing:worker1 RIGHT LEFT   # take job, keep a copy
# ... process the job ...
LREM processing:worker1 1 <job>                     # done, remove the copy
```

📌 Version note: LMOVE and its blocking version BLMOVE arrived in Redis 6.2, replacing the older RPOPLPUSH / BRPOPLPUSH (which only supported right-to-left). Internals note: since Redis 3.2, lists are stored as a 'quicklist' — a linked list

> of compressed blocks — replacing the old dual ziplist/linkedlist encoding. You got better memory usage for free without changing any code.

Chapter 6 — Hashes: Objects Inside a Key

A hash is a collection of field → value pairs stored under one key. Think of it as a small object or a dictionary. Perfect for storing an entity like a user, a product, or a session.

Core commands

```
HSET user:1001 name "Digamber" email "d@x.com" plan "pro"
HGET user:1001 email            # "d@x.com"
HGETALL user:1001               # all fields and values
HMGET user:1001 name plan       # several specific fields
HDEL user:1001 plan             # delete one field
HEXISTS user:1001 email         # 1
HLEN user:1001                  # number of fields
HKEYS user:1001                 # field names only
HVALS user:1001                 # values only
HINCRBY user:1001 logins 1      # atomic counter INSIDE a hash
HSCAN user:1001 0 COUNT 100     # scan big hashes safely
HRANDFIELD user:1001 2          # random fields (Redis 6.2+)
```

Hash vs JSON-in-a-string: which one?

	Hash	JSON string
Update one field	HSET one field — cheap	Read whole JSON, change, write whole JSON back
Read one field	HGET — cheap	Must read and parse everything
Nested data	Not supported (flat only)	Full nesting
Atomic counters per field	HINCRBY — yes	No
Memory for small objects	Very efficient (listpack)	Less efficient

Rule of thumb: flat entity you update field-by-field → hash. Deeply nested blob you always read/write whole → JSON string (or the RedisJSON module, Chapter 16).

The memory trick: listpack encoding

When a hash is small (by default: at most 128 fields, each value at most 64 bytes), Redis stores it in a super-compact flat format called **listpack** instead of a real hash table. Millions of small hashes can use several times less memory than the same data as separate string keys. Instagram famously used this trick to cut a photo-ID mapping from 21 GB to 5 GB.

> 📌 Version note: the compact encoding was called 'ziplist' until Redis 7.0 replaced it with the safer, slightly faster 'listpack'. Behaviour is automatic in both cases.

> Also: HEXPIRE / field-level TTL inside hashes only arrived in Redis 7.4 — on anything older, TTL exists only per key, not per hash field. This is a classic interview trap question.

Chapter 7 — Sets: Unique, Unordered Collections

A set stores unique strings with no order. Adding the same member twice keeps only one copy. Membership checks are O(1) — instant regardless of size.

Core commands

```
SADD tags:post:42 "redis" "database" "cache"
SADD tags:post:42 "redis"        # ignored, already present
SMEMBERS tags:post:42            # all members (careful on huge sets)
SISMEMBER tags:post:42 "redis"   # 1
SMISMEMBER tags:post:42 "redis" "mysql"   # [1, 0]  (Redis 6.2+)
SCARD tags:post:42               # 3 (count)
SREM tags:post:42 "cache"
SPOP tags:post:42                # remove and return a RANDOM member
SRANDMEMBER tags:post:42 2       # random members without removing
SSCAN tags:post:42 0             # iterate a big set safely
```

Set algebra — the superpower

```
SADD followers:alice 1 2 3 4
SADD followers:bob   3 4 5 6

SINTER followers:alice followers:bob     # {3,4}  common followers
SUNION followers:alice followers:bob     # {1..6} all followers
SDIFF  followers:alice followers:bob     # {1,2}  alice-only
SINTERCARD 2 followers:alice followers:bob   # just the COUNT (Redis 7.0+)
SINTERSTORE common 2 followers:alice followers:bob  # save result to a key
```

"Show mutual friends", "users who bought A and B", "visitors who saw the ad but did not buy" — each is one command, no SQL join needed.

Pattern: unique visitor tracking per day

```
SADD visitors:2026-07-08 "user:1001"
SADD visitors:2026-07-08 "user:1002"
SCARD visitors:2026-07-08         # today's unique count
EXPIRE visitors:2026-07-08 604800 # auto-delete after 7 days
```

Caution: one member = real memory. Ten million visitors = a big set. If you only need the COUNT (not the actual members), HyperLogLog (Chapter 9) does it in 12 KB with ~0.8% error.

Chapter 8 — Sorted Sets (ZSET): The Most Powerful Type

A sorted set is like a set (unique members) where every member also carries a floating-point **score**. Redis keeps members permanently ordered by score. Insert, delete, and rank lookups are all O(log N). One structure powers leaderboards, priority queues, time-indexed data, and rate limiters.

Core commands

```
ZADD leaderboard 850 "alice" 920 "bob" 780 "carol"
ZSCORE leaderboard "alice"           # 850
ZINCRBY leaderboard 30 "alice"       # add 30 → 880, position updates
automatically
ZRANK leaderboard "carol"            # 0 (lowest score = rank 0)
ZREVRANK leaderboard "bob"           # 0 (highest first)
ZCARD leaderboard                    # 3
ZCOUNT leaderboard 800 900           # how many scores in [800, 900]
ZREM leaderboard "carol"
```

Reading ranges

```
# Top 10 with scores (highest first)
ZRANGE leaderboard 0 9 REV WITHSCORES

# All members with score between 800 and 900
ZRANGE leaderboard 800 900 BYSCORE

# Old style (still everywhere in tutorials and codebases):
ZREVRANGE leaderboard 0 9 WITHSCORES
ZRANGEBYSCORE leaderboard 800 900
```

📌 Version note: Redis 6.2 unified the six old range commands (ZRANGEBYSCORE, ZREVRANGE, ZRANGEBYLEX...) into one flexible ZRANGE with REV / BYSCORE / BYLEX options, plus ZRANGESTORE to save results into a new key. The old commands still work but new code should use the unified form. ZADD gained NX / XX / GT / LT / CH options in 3.0–6.2: e.g. ZADD lb GT 900 alice updates alice only if 900 is GREATER than her current score — perfect for 'best score' leaderboards.

Pattern 1: game leaderboard

```
ZINCRBY lb:weekly 120 "player:77"        # player earned points
ZREVRANK lb:weekly "player:77"           # their live rank
ZRANGE lb:weekly 0 99 REV WITHSCORES     # top-100 page
```

Pattern 2: delayed job scheduler

Score = the Unix timestamp when the job should run. A worker polls for 'jobs whose time has come':

```
ZADD delayed_jobs 1767290400 '{"task":"send_reminder","user":1001}'

# worker loop, every second:
ZRANGEBYSCORE delayed_jobs 0 <now> LIMIT 0 10   # due jobs
ZREM delayed_jobs <job>                          # claim it
```

Pattern 3: sliding-window rate limiter

```
# allow 100 requests per 60 seconds per user
ZADD rate:u1001 <now_ms> <unique_request_id>
ZREMRANGEBYSCORE rate:u1001 0 <now_ms - 60000>   # drop old entries
ZCARD rate:u1001                                  # if > 100 → reject
EXPIRE rate:u1001 61
```

Pattern 4: autocomplete with ZRANGEBYLEX

If every member has the SAME score (e.g. 0), Redis sorts members alphabetically. ZRANGEBYLEX then gives you prefix search:

```
ZADD cities 0 "delhi" 0 "dehradun" 0 "dhanbad" 0 "mumbai"
ZRANGEBYLEX cities [de (df       # everything starting with "de"
# 1) "dehradun"  2) "delhi"
```

Chapter 9 — Advanced Types: Streams, Bitmaps, HyperLogLog, Geo, Bitfields

9.1 Streams — Redis's answer to Kafka (Redis 5.0+)

A stream is an **append-only log**. Every entry gets an auto-generated ID like 1720400000000-0 (millisecond timestamp + sequence). Unlike Pub/Sub (Chapter 11), stream entries are **stored** — a consumer that was offline can catch up later. Streams support **consumer groups**: several workers share one stream, each message is delivered to exactly one worker in the group, and workers must ACK messages so crashed work can be re-delivered.

```
# produce ("*" = auto-generate the ID)
XADD orders * customer 1001 amount 500
XLEN orders
XRANGE orders - +                    # read everything (- = min, + = max)
XREAD COUNT 10 STREAMS orders 0      # read from the beginning

# consumer groups
XGROUP CREATE orders processors $     # $ = only new messages from now
XREADGROUP GROUP processors worker1 COUNT 10 BLOCK 5000 STREAMS orders >
XACK orders processors 1720400000000-0        # mark done
XPENDING orders processors                     # what is delivered but not
ACKed?
XAUTOCLAIM orders processors worker2 60000 0   # steal messages stuck >60s
(6.2+)
XTRIM orders MAXLEN ~ 100000                   # cap stream length
```

When to choose Streams over a real Kafka: you already run Redis, throughput is below roughly 50k messages/second, and you do not need multi-datacenter log replication. For a startup's background processing, Streams is usually enough and one less system to operate.

9.2 Bitmaps — millions of yes/no flags in tiny memory

A bitmap is just a string treated as an array of bits. Bit number N can be 0 or 1. 10 million users' daily-active flags = 10,000,000 bits ≈ 1.25 MB.

```
SETBIT active:2026-07-08 1001 1     # user 1001 was active today
GETBIT active:2026-07-08 1001       # 1
BITCOUNT active:2026-07-08          # how many users active today
# users active BOTH today and yesterday:
BITOP AND both active:2026-07-08 active:2026-07-07
BITCOUNT both
```

9.3 HyperLogLog — count unique things in 12 KB

HyperLogLog is a probabilistic counter. It answers "how many DISTINCT items have I seen?" using at most 12 KB of memory regardless of whether you add one thousand or one billion items, with a standard error of about 0.81%. You cannot get the members back — only the count.

```
PFADD visitors:today "ip1" "ip2" "ip3"
PFCOUNT visitors:today                       # ≈ 3
PFMERGE visitors:week visitors:mon visitors:tue   # union of days
```

9.4 Geo — location queries (Redis 3.2+)

Geo commands store longitude/latitude pairs (internally inside a sorted set using a geohash as the score) and answer "what is near me?" questions.

```
GEOADD drivers 77.2090 28.6139 "driver:1"    # lon lat member (Delhi)
GEOADD drivers 77.1025 28.7041 "driver:2"
GEODIST drivers driver:1 driver:2 km
GEOSEARCH drivers FROMLONLAT 77.20 28.61 BYRADIUS 10 km ASC WITHCOORD
```

> 📌 Version note: GEO commands were introduced in Redis 3.2 — before that, doing 'nearby drivers' in Redis required manual geohash tricks in application code. GEOSEARCH (6.2) replaced the older GEORADIUS, which is now deprecated.

9.5 BITFIELD — packed integers (Redis 3.2+)

BITFIELD lets you treat one string as an array of small integers of any bit width — e.g. store 1,000 players' 8-bit levels in a single 1,000-byte key, with atomic per-slot increments:

```
BITFIELD players SET u8 0 42          # slot 0 = 42
BITFIELD players INCRBY u8 0 5        # slot 0 → 47
BITFIELD players GET u8 0
```

Chapter 10 — Pipelining, Transactions, and Lua Scripting

10.1 Pipelining — cut network round-trips

A Redis command itself takes microseconds, but the network round-trip between your app and Redis takes maybe half a millisecond. If you send 100 commands one by one, you pay 100 round-trips. Pipelining sends them all in one batch:

```
# Python (redis-py)
pipe = r.pipeline(transaction=False)
for uid in user_ids:
    pipe.hgetall(f"user:{uid}")
results = pipe.execute()      # ONE round-trip, list of results
```

Pipelining routinely makes bulk operations 10–50x faster. It gives no atomicity — other clients' commands may interleave on the server — it only saves network time.

10.2 Transactions — MULTI / EXEC

```
MULTI
INCR balance:from -500       # queued
INCR balance:to 500          # queued
EXEC                         # both run back-to-back, nothing in between
```

Redis transactions guarantee **isolation** (no other client's command runs in the middle) but NOT rollback. If the second command fails at runtime (e.g. wrong type), the first one has still happened. People coming from SQL find this shocking — say it confidently in interviews.

WATCH adds optimistic locking: watch a key, then MULTI/EXEC. If anyone modified the watched key before your EXEC, the EXEC fails (returns nil) and you retry.

```
WATCH stock:item42
val = GET stock:item42        # read, decide in app code
MULTI
DECR stock:item42
EXEC                          # nil if someone changed stock meanwhile → retry
```

10.3 Lua scripting — true atomic logic (Redis 2.6+)

A Lua script runs on the server as ONE atomic unit. No other command can interleave. This is how you express "read, check condition, then write" safely:

```
-- rate limiter: allow ARGV[2] requests per ARGV[1] seconds
local current = redis.call('INCR', KEYS[1])
```

```
if current == 1 then
    redis.call('EXPIRE', KEYS[1], ARGV[1])
end
if current > tonumber(ARGV[2]) then
    return 0
end
return 1
```

```
# Python
allowed = r.eval(script, 1, "rate:user:1001", 60, 100)
# better: register once, call by hash
sha = r.script_load(script)
r.evalsha(sha, 1, "rate:user:1001", 60, 100)
```

📌 Version note: EVAL/Lua arrived in Redis 2.6 (2012) — before that, atomic multi-step logic required WATCH-retry loops or was impossible. Redis 7.0 added Functions (FUNCTION LOAD): named, server-stored libraries of Lua code that survive restarts and replicate properly, fixing EVAL's operational weaknesses (scripts living only in app code, cache-miss NOSCRIPT errors).

Chapter 11 — Pub/Sub and Keyspace Notifications

Pub/Sub basics

Publish/Subscribe is fire-and-forget messaging. Subscribers listen to channels; publishers send. If nobody is listening at that moment, the message is **gone forever** — nothing is stored.

```
# terminal 1
SUBSCRIBE news:tech

# terminal 2
PUBLISH news:tech "Redis 8 released"     # returns: number of receivers

# pattern subscription
PSUBSCRIBE news:*
```

	Pub/Sub	Streams
Message stored?	No — lost if no listener	Yes — persistent log
Offline consumer catches up?	Never	Yes, from any position
Delivery	All current subscribers	Consumer groups: each msg to one worker
Acknowledgement	None	XACK, re-delivery of failed work
Good for	Live tickers, chat presence, cache-invalidation broadcast	Job queues, event sourcing, audit logs

> 📌 Version note: in Redis Cluster, classic PUBLISH is broadcast to every node — wasteful at scale. Redis 7.0 added Sharded Pub/Sub (SPUBLISH / SSUBSCRIBE) where a channel lives on one shard only.

Keyspace notifications

Redis can publish an event whenever a key changes or expires (disabled by default, enable with notify-keyspace-events). The classic use: react to expiring keys, e.g. "session ended → mark user offline":

```
CONFIG SET notify-keyspace-events Ex
SUBSCRIBE __keyevent@0__:expired      # receives the name of every expired key
```

Warning: expiry notifications fire when Redis actually deletes the key (lazy/active expiry, Chapter 3), which can be seconds late. Do not build precise timers on this.

Chapter 12 — Persistence: RDB, AOF, and Surviving a Crash

RAM disappears on restart. Redis offers two ways to survive: snapshots (RDB) and a write log (AOF). Understanding both — and their trade-offs — is mandatory interview material.

RDB — point-in-time snapshots

Redis forks a child process; the child writes the ENTIRE dataset to a compact binary file (dump.rdb) while the parent keeps serving traffic (copy-on-write memory). Configured as:

```
save 3600 1      # snapshot if ≥1 change in 3600s
save 300 100     # or ≥100 changes in 300s
save 60 10000    # or ≥10000 changes in 60s

BGSAVE           # trigger manually in background
SAVE             # blocking snapshot — avoid in production
LASTSAVE         # timestamp of last successful snapshot
```

- **Pros:** compact single file, fastest restart, great for backups, minimal steady-state overhead.
- **Cons:** crash between snapshots loses EVERYTHING since the last one (minutes of data). Fork of a huge dataset can cause a latency spike and briefly doubles memory in the worst case.

AOF — append-only file

Every write command is appended to a log. On restart Redis replays the log to rebuild the data. The critical setting is how often Redis calls fsync (force the OS to actually write to disk):

```
appendonly yes
appendfsync everysec    # default: at most ~1 second of data lost. Best
balance.
appendfsync always      # fsync every write. Safest, slowest.
appendfsync no          # let the OS decide (~30s risk). Fastest.
```

The AOF grows forever, so Redis periodically **rewrites** it: instead of the full history ("INCR x" a million times), it writes the minimal commands to recreate current data ("SET x 1000000"). Triggered automatically or with BGREWRITEAOF.

Which one in production?

Both. RDB for fast restarts and off-site backups; AOF (everysec) for durability. Since Redis 4.0 the hybrid mode (aof-use-rdb-preamble yes) writes the AOF as an

RDB snapshot + recent commands: fast to load AND durable. Redis 7.0 further split the AOF into multiple files (base + increments) in a folder, making rewrites cheaper.

> 📌 Version notes: (1) 'Hybrid' RDB+AOF persistence arrived in 4.0. (2) Multi-part AOF arrived in 7.0 — before that, AOF rewrite could double disk I/O painfully. (3) Diskless replication (master streams RDB straight to replica sockets, skipping disk) became stable around 6.0. Interview answer for 'Can Redis be a primary database?': with AOF everysec + replication, yes for data where losing ≤1 second is acceptable (sessions, carts, counters) — not for financial ledgers.

Chapter 13 — Memory Management and Eviction

maxmemory and eviction policies

Without a limit, Redis grows until the OS kills it. Always set maxmemory in production. When the limit is reached, maxmemory-policy decides what happens:

Policy	Meaning	Use when
noeviction	New writes fail with an error	Redis is a primary store; losing data silently is worse than erroring
allkeys-lru	Evict least-recently-used key, any key	Pure cache — the safe default
allkeys-lfu	Evict least-frequently-used (4.0+)	Cache where some keys are always hot; survives one-off scans better
volatile-lru / volatile-lfu	LRU/LFU but only among keys WITH a TTL	Mixed cache + permanent data in one instance (avoid this setup anyway)
volatile-ttl	Evict keys closest to expiring	Rarely
allkeys-random / volatile-random	Random eviction	Almost never

Redis's LRU/LFU is **approximate**: it samples a few keys (maxmemory-samples, default 5) and evicts the best candidate among them, instead of maintaining a perfect global ordering — a deliberate trade of accuracy for speed and memory.

> 📌 Version note: LFU did not exist before Redis 4.0 — old deployments had only LRU/random/ttl. LFU is usually the better cache policy because one full table-scan (someone exports a report) does not flush your genuinely hot keys.

Hunting memory problems

```
INFO memory                    # used_memory, fragmentation ratio
MEMORY USAGE user:1001         # bytes used by one key (4.0+)
MEMORY DOCTOR                  # plain-English advice (4.0+)
redis-cli --bigkeys            # sample keyspace, report biggest key per type
redis-cli --memkeys            # sample by memory usage
redis-cli --hotkeys            # most-accessed keys (needs LFU policy)
OBJECT ENCODING mykey          # listpack? hashtable? quicklist?
```

Big keys and hot keys

- **Big key** = one key holding a huge value (a 5M-item list, a 500k-field hash). Problems: DEL blocks (use UNLINK), migration in Cluster stalls, one query returns megabytes. Fix: split into buckets (user:1001:feed:0..9), trim with LTRIM/XTRIM, delete collections in SCAN-sized batches.
- **Hot key** = one key receiving a huge share of traffic. Since command execution is single-threaded, one celebrity's profile key can max out a CPU core. Fixes: replicate reads to replicas, duplicate the key with suffixes (profile:42:0..4, pick randomly), cache it in the application process itself, or use Redis 6 client-side caching (the server tracks what you cached and pings you on invalidation).

Memory fragmentation: if INFO memory shows mem_fragmentation_ratio well above 1.5, the allocator is wasting RAM. Redis 4.0+ has activedefrag yes to compact memory live.

Chapter 14 — Replication, Sentinel, and Cluster

14.1 Replication — copies of your data

One primary (master) accepts writes; any number of replicas mirror it. Replication is **asynchronous**: the master answers the client without waiting for replicas, so a crashing master can lose the last few writes. Replicas serve reads to scale read traffic.

```
# on the replica
REPLICAOF 10.0.0.5 6379
REPLICAOF NO ONE          # promote to standalone/master
INFO replication           # role, lag, connected replicas
WAIT 1 500                 # block until ≥1 replica confirms, max 500ms
```

📌 Version note: the command was SLAVEOF before Redis 5.0 renamed it REPLICAOF. Also: Redis 2.8 introduced partial resynchronization (PSYNC) — before that, ANY network blip forced a full copy of the entire dataset to the replica. Redis 4.0's PSYNC2 kept partial resync working even across failovers.

14.2 Sentinel — automatic failover

Sentinel is a separate small process (run at least 3 of them) that monitors your master and replicas. If the master stops responding, Sentinels vote (quorum), elect one replica as the new master, reconfigure the others, and tell client libraries the new address. Sentinel gives **high availability** without sharding — your data still fits on one machine.

```
# sentinel.conf
sentinel monitor mymaster 10.0.0.5 6379 2   # quorum of 2
sentinel down-after-milliseconds mymaster 5000
sentinel failover-timeout mymaster 60000
```

14.3 Cluster — sharding for datasets bigger than one machine

Redis Cluster splits the keyspace into **16384 hash slots**. Each master node owns a range of slots; each master has replicas for failover. A key's slot is CRC16(key) mod 16384. Cluster-aware clients cache the slot map and send each command straight to the right node (following MOVED/ASK redirects when slots migrate).

```
# quick 6-node cluster (3 masters + 3 replicas)
redis-cli --cluster create 10.0.0.1:6379 10.0.0.2:6379 10.0.0.3:6379 \
  10.0.0.4:6379 10.0.0.5:6379 10.0.0.6:6379 --cluster-replicas 1
CLUSTER INFO
CLUSTER SLOTS
```

Hash tags — the detail interviewers love. Multi-key commands (MGET, SINTER, transactions, Lua) only work when all keys are in the SAME slot. Wrap the shared part in braces: {user:1001}:profile and {user:1001}:settings hash only on "user:1001", guaranteeing co-location.

- SELECT / multiple databases do not exist in Cluster — database 0 only.
- Transactions and Lua scripts cannot span nodes.
- Don't reach for Cluster early: one instance + replicas + Sentinel handles most workloads. Cluster is for when data or write throughput genuinely exceeds one machine.

> 📌 Version note: Redis Cluster shipped in 3.0 (2015). Before that, sharding meant client-side hashing (Twemproxy from Twitter, or manual consistent hashing in app code) with no automatic failover or resharding. The redis-trib.rb Ruby tool managed clusters until 5.0 moved everything into redis-cli --cluster.

Chapter 15 — Security and a Production Checklist

The uncomfortable history

Old Redis was famously insecure by default: it listened on all interfaces with NO password. Thousands of databases were wiped or ransomed by bots scanning port 6379. Redis 3.2 responded with **protected mode**: if no password and no explicit bind address are set, Redis refuses connections from other machines.

Modern security layers

```
# 1. Password (legacy, still fine for simple setups)
requirepass a-very-long-random-string
AUTH a-very-long-random-string

# 2. ACLs — real users with permissions (Redis 6.0+)
ACL SETUSER cacheuser on >secret ~cache:* +get +set +del
ACL SETUSER admin on >adminpass ~* +@all
ACL LIST
# per-user: which commands (+get, +@read) and which keys (~cache:*)

# 3. Disable/rename dangerous commands
rename-command FLUSHALL ""
rename-command CONFIG "cfg-8f2a1"

# 4. TLS encryption in transit (Redis 6.0+)
tls-port 6380
tls-cert-file /etc/redis/redis.crt
tls-key-file /etc/redis/redis.key
```

📌 Version note: before 6.0 there was ONE shared password for everyone and no TLS (people ran stunnel in front). ACLs + TLS in 6.0 made Redis enterprise-acceptable. If an interviewer asks 'how do you give the analytics team read-only access?' — the answer is an ACL user with +@read.

Production checklist

- bind to private interfaces only; never expose 6379 to the internet
- requirepass or ACLs always on, even inside a VPC
- maxmemory set + allkeys-lfu (cache) or noeviction (store)
- appendonly yes with everysec, plus RDB snapshots shipped off-server
- Replicas + Sentinel (or a managed service: AWS ElastiCache, Azure Cache, GCP Memorystore)

- Monitoring: INFO stats (hit rate = keyspace_hits/(hits+misses)), latency, memory, connected_clients; alert on fragmentation and evicted_keys
- SLOWLOG GET regularly; investigate anything over a few milliseconds
- Never KEYS, never FLUSHALL, never MONITOR on production; use SCAN/UNLINK
- Set client timeouts and use connection pooling in the app

Chapter 16 — Redis Version History: Before 3.2 vs Everything After

This chapter is your time machine. Interviewers with grey hair remember pre-3.2 Redis; codebases still contain deprecated commands from that era. Knowing what appeared when lets you read old code, upgrade safely, and sound like you have used Redis for a decade.

The old world: Redis before 3.2 (2009 – early 2016)

- **1.0 (2009):** strings, lists, sets. The original cache-and-queue tool.
- **2.0 (2010):** hashes; MULTI/EXEC transactions; Pub/Sub.
- **2.2 (2011):** first maxmemory eviction policies (LRU era begins).
- **2.6 (2012):** Lua scripting (EVAL) — atomic multi-step logic finally possible; expanded SET options begin (2.6.12 folds NX/XX/EX/PX into SET); millisecond expiry (PEXPIRE).
- **2.8 (2013):** SCAN family — finally a safe alternative to KEYS; partial resync (PSYNC) — replicas no longer re-copy everything after a blip; keyspace notifications; Sentinel became genuinely usable (v2).
- **3.0 (2015):** Redis Cluster GA — official sharding after years of Twemproxy and client-side hashing.

What life was like: single shared password (or none), KEYS in cron jobs freezing servers, DEL on big keys causing timeouts, no streams (people faked queues with lists), no LFU, no modules, no TLS, lists stored as ziplist-or-linkedlist with sudden memory jumps.

The turning point: Redis 3.2 (May 2016)

- **GEO commands** (GEOADD, GEORADIUS) — location features without PostGIS
- **BITFIELD** — packed integer arrays with atomic ops
- **Protected mode** — the answer to the mass-wiping of open Redis servers
- **Quicklist** — lists re-implemented as linked chains of compressed blocks; big memory savings, no more encoding cliff
- SPOP with count; Lua debugger (redis-cli --ldb)

Redis 4.0 (2017)

- **Modules API** — load C extensions: this later enabled RedisJSON, RediSearch, RedisTimeSeries, RedisBloom
- **UNLINK** — non-blocking delete; lazy freeing (lazyfree-*) in general

- **LFU eviction** — allkeys-lfu / volatile-lfu
- **MEMORY USAGE / MEMORY DOCTOR** — introspection at last
- **Hybrid RDB+AOF persistence**; PSYNC2 (partial resync survives failover); active defragmentation; SWAPDB

Redis 5.0 (2018)

- **Streams (XADD, XREADGROUP, XACK…)** — the headline feature; Kafka-style logs with consumer groups
- ZPOPMIN/ZPOPMAX and blocking BZPOPMIN/BZPOPMAX
- SLAVEOF renamed REPLICAOF (terminology cleanup)
- redis-cli --cluster absorbed the old redis-trib.rb Ruby tool

Redis 6.0 (2020)

- **ACLs** — real users, per-command and per-key-pattern permissions
- **TLS** built in — encrypted connections without stunnel
- **Threaded I/O** — network read/write on multiple threads (command execution still single-threaded); big throughput gains
- **Client-side caching (RESP3 tracking)** — server tells clients when their locally-cached keys change
- RESP3 protocol; diskless replication matured

Redis 6.2 (2021) — the quality-of-life release

- GETDEL, GETEX, COPY, OBJECT FREQ, ZRANDMEMBER, HRANDFIELD, SMISMEMBER, LMOVE/BLMOVE (replacing RPOPLPUSH), ZRANGESTORE and the unified ZRANGE (REV/BYSCORE/BYLEX), GEOSEARCH (replacing GEORADIUS), XAUTOCLAIM for streams

Redis 7.0 (2022)

- **Functions** — server-stored, replicated Lua libraries; the fix for EVAL's operational problems
- **Sharded Pub/Sub** (SPUBLISH/SSUBSCRIBE) for Cluster
- **Multi-part AOF** — cheaper rewrites, AOF now a directory
- **Listpack** fully replaced ziplist encodings
- ACL v2 (selectors), command introspection (COMMAND DOCS), cluster-wide improvements

Redis 7.2 / 7.4 (2023–2024)

- 7.2: performance work, WAITAOF (wait for AOF fsync), client-eviction controls
- **7.4: hash field expiration (HEXPIRE/HPEXPIRE/HTTL)** — TTL per hash field, a feature requested for a decade
- 2024 licensing storm: Redis Inc. moved 7.4 to dual RSALv2/SSPLv1 (no longer OSI open source). The community forked 7.2 as **Valkey** under the Linux Foundation; AWS and Google backed Valkey for their managed services.

Redis 8.0 (2025)

- License changed again: **AGPLv3 added** — Redis is open source again (with copyleft strings attached)
- The former paid 'Redis Stack' modules folded into core: **JSON documents, the Redis Query Engine (search + secondary indexes), time series, and probabilistic types (Bloom/Cuckoo filters, t-digest, top-k)**
- **Vector sets** (beta) — native vector similarity search for AI/RAG workloads
- Substantial per-command performance improvements

Cheat table: deprecated → modern replacement

Old (pre-3.2 era habit)	Modern replacement	Since
SETNX / SETEX / PSETEX	SET with NX / EX / PX options	2.6.12
KEYS pattern	SCAN cursor MATCH pattern	2.8
DEL bigkey	UNLINK bigkey	4.0
Lists as job queues (no ACK)	Streams + consumer groups	5.0
SLAVEOF	REPLICAOF	5.0
Single shared requirepass	ACL users	6.0
stunnel for encryption	Built-in TLS	6.0
RPOPLPUSH / BRPOPLPUSH	LMOVE / BLMOVE	6.2
ZREVRANGE / ZRANGEBYSCORE / ZRANGEBYLEX	ZRANGE with REV / BYSCORE / BYLEX	6.2
GEORADIUS	GEOSEARCH	6.2
GETSET	SET ... GET	6.2
EVAL scripts in app code	FUNCTION LOAD (server-side libraries)	7.0
Whole-key TTL only for	HEXPIRE per-field TTL	7.4

Old (pre-3.2 era habit)	Modern replacement	Since
hashes		
Separate RedisJSON/RediSearch modules	Built into core Redis	8.0

Chapter 17 — Production Caching Patterns (Stampedes, Locks, Invalidation)

17.1 Cache-aside — the pattern you'll write 90% of the time

```
def get_user(user_id):
    key = f"user:{user_id}"
    cached = r.get(key)
    if cached:
        return json.loads(cached)          # cache HIT
    user = db.fetch_user(user_id)          # cache MISS → real DB
    r.setex(key, 300, json.dumps(user))    # store with 5-min TTL
    return user
```

The application owns the logic: check cache → miss → read DB → fill cache. Simple, resilient (Redis down = slower app, not broken app), and only caches what is actually requested.

17.2 The other patterns (know them for interviews)

- **Write-through:** every write updates DB and cache together. Reads never miss, writes cost more.
- **Write-behind:** write to Redis first, flush to DB asynchronously. Fastest writes; risk of loss if Redis dies before the flush. Rare, used for metrics-like data.
- **Read-through:** the cache layer itself fetches from DB on miss (the app only talks to the cache). Needs library support.

17.3 Cache invalidation

"There are only two hard things in Computer Science: cache invalidation and naming things." Your three options, weakest to strongest:

- **TTL only:** data may be stale up to TTL seconds. Fine for most read-heavy data. Start here.
- **Delete-on-write:** when the user updates their profile, DEL user:1001 right after the DB commit. Next read repopulates. (Delete, don't SET — two concurrent writers SETting can race and leave stale data.)
- **Versioned keys:** include a version in the key (user:1001:v7); bump the version on change; old entries die by TTL. No delete races at all.

17.4 Cache stampede (thundering herd)

A hot key expires. 10,000 requests miss simultaneously. All 10,000 hit PostgreSQL. PostgreSQL falls over. Three defences, use them together:

- **TTL jitter:** ttl = 300 + random(0, 60) so a fleet of keys never expires in sync.

- **Mutex on miss:** first misser takes a lock (SET lock:key token NX EX 10) and rebuilds; everyone else waits briefly or serves stale data.
- **Probabilistic early refresh:** each request has a small, rising chance to refresh the key BEFORE expiry, so the rebuild happens while old data still serves traffic.

```
def get_with_stampede_protection(key, rebuild, ttl=300):
    val = r.get(key)
    if val is not None:
        return val
    if r.set(f"lock:{key}", "1", nx=True, ex=10):   # I am the rebuilder
        try:
            val = rebuild()
            r.setex(key, ttl + random.randint(0, 60), val)
            return val
        finally:
            r.delete(f"lock:{key}")
    time.sleep(0.05)                                  # someone else is
rebuilding
    return get_with_stampede_protection(key, rebuild, ttl)
```

17.5 Distributed locks done properly

```
import uuid

token = str(uuid.uuid4())
# acquire: NX = only if free, EX = auto-release if I crash
got = r.set("lock:report:daily", token, nx=True, ex=30)

# release: ONLY if I still own it — must be atomic, so Lua:
RELEASE = """
if redis.call('GET', KEYS[1]) == ARGV[1] then
    return redis.call('DEL', KEYS[1])
else
    return 0
end"""
r.eval(RELEASE, 1, "lock:report:daily", token)
```

Why the token? Suppose your job stalls for 35 seconds. The lock expired at 30s; another worker acquired it. A plain DEL would now delete THEIR lock. The token check prevents that. Also know the limits: single-instance Redis locks are for efficiency (don't run the same cron twice), not for absolute correctness under network partitions — for that, use a consensus store (etcd/ZooKeeper) or Redlock with its documented caveats (the famous Kleppmann vs antirez debate).

17.6 Caching LLM responses (2026-relevant)

LLM API calls cost real money. Cache key = hash of (model + normalized prompt + temperature + max_tokens); value = the completion; TTL = hours or days for deterministic (temperature 0) prompts. Namespace by tenant (llm:{tenant}:{hash}) so one customer never receives another's cached answer. With shared system prompts and repeated user queries, hit rates of 30–50% — and equivalent bill reduction — are realistic.

Chapter 18 — Redis with Django: The Complete Guide

18.1 Setup

```
pip install django-redis redis

# settings.py
CACHES = {
    "default": {
        "BACKEND": "django_redis.cache.RedisCache",
        "LOCATION": "redis://127.0.0.1:6379/1",
        "OPTIONS": {
            "CLIENT_CLASS": "django_redis.client.DefaultClient",
            "CONNECTION_POOL_KWARGS": {"max_connections": 50},
        },
        "KEY_PREFIX": "myapp",   # avoids collisions if Redis is shared
        "TIMEOUT": 300,           # default TTL in seconds
    }
}
```

Django 4.0+ also ships a built-in django.core.cache.backends.redis.RedisCache. django-redis remains the common choice because it exposes the raw client, better pooling options, and pattern deletes.

18.2 The cache API

```
from django.core.cache import cache

cache.set("greeting", "hello", timeout=60)
cache.get("greeting")                    # "hello"
cache.get("missing", "default-value")
cache.get_or_set("stats", compute_stats, timeout=600)   # cache-aside in one
line
cache.delete("greeting")
cache.incr("api_calls")                  # atomic (key must exist)
cache.set_many({"a": 1, "b": 2}, timeout=120)
cache.get_many(["a", "b"])

# django-redis extras:
cache.delete_pattern("user:1001:*")      # uses SCAN internally
from django_redis import get_redis_connection
r = get_redis_connection("default")      # raw redis-py client
r.zadd("leaderboard", {"alice": 850})    # full Redis power
```

18.3 Caching views and template fragments

```
from django.views.decorators.cache import cache_page
```

```
@cache_page(60 * 5)                       # whole view cached 5 minutes
def product_list(request):
    ...

# per-user variant caching:
from django.views.decorators.vary import vary_on_cookie
@cache_page(60)
@vary_on_cookie
def dashboard(request): ...

# in templates:
{% load cache %}
{% cache 300 sidebar request.user.id %}
   ...expensive sidebar...
{% endcache %}
```

18.4 Sessions in Redis

```
# settings.py — sessions become fast and share across app servers
SESSION_ENGINE = "django.contrib.sessions.backends.cache"
SESSION_CACHE_ALIAS = "default"
# safer variant (survives cache eviction, falls back to DB):
# SESSION_ENGINE = "django.contrib.sessions.backends.cached_db"
```

18.5 Invalidation with signals

```
from django.db.models.signals import post_save, post_delete
from django.dispatch import receiver
from django.core.cache import cache

@receiver([post_save, post_delete], sender=Product)
def invalidate_product(sender, instance, **kwargs):
    cache.delete(f"product:{instance.pk}")
    cache.delete_pattern("product_list:*")
```

18.6 Celery with Redis as broker

```
pip install celery redis

# settings.py
CELERY_BROKER_URL = "redis://127.0.0.1:6379/0"
CELERY_RESULT_BACKEND = "redis://127.0.0.1:6379/0"

# tasks.py
from celery import shared_task
@shared_task
def send_welcome_email(user_id):
    ...

# call from a view — returns instantly, worker does the job
```

```
send_welcome_email.delay(user.id)
```

Under the hood Celery LPUSHes the task onto a Redis list and workers BRPOP it — exactly the queue pattern from Chapter 5. Use a DIFFERENT Redis database (or instance) for the broker vs the cache: a cache under memory pressure with allkeys-lru could otherwise evict your pending jobs.

18.7 A rate-limited API view (raw client)

```
from django_redis import get_redis_connection
from django.http import JsonResponse

def rate_limited_api(request):
    r = get_redis_connection("default")
    key = f"rl:{request.user.id or request.META['REMOTE_ADDR']}"
    current = r.incr(key)
    if current == 1:
        r.expire(key, 60)
    if current > 100:
        return JsonResponse({"error": "rate limit exceeded"}, status=429)
    ...  # normal handling
```

Chapter 19 — Redis with FastAPI: The Complete Guide

19.1 Async setup with lifespan

FastAPI is async, so use redis-py's asyncio client (redis.asyncio — the old separate aioredis package was merged into redis-py 4.2+). Create ONE connection pool at startup and share it:

```
from contextlib import asynccontextmanager
from fastapi import FastAPI, Request
import redis.asyncio as redis

@asynccontextmanager
async def lifespan(app: FastAPI):
    app.state.redis = redis.Redis(
        host="127.0.0.1", port=6379, db=0,
        decode_responses=True, max_connections=50,
    )
    yield
    await app.state.redis.aclose()

app = FastAPI(lifespan=lifespan)

async def get_redis(request: Request) -> redis.Redis:
    return request.app.state.redis        # dependency
```

19.2 Cache-aside endpoint

```
import json
from fastapi import Depends

@app.get("/products/{pid}")
async def product(pid: int, r: redis.Redis = Depends(get_redis)):
    key = f"product:{pid}"
    if (cached := await r.get(key)) is not None:
        return json.loads(cached)
    data = await fetch_product_from_db(pid)      # slow path
    await r.set(key, json.dumps(data), ex=300)
    return data
```

19.3 fastapi-cache2 — caching as a decorator

```
pip install fastapi-cache2

from fastapi_cache import FastAPICache
from fastapi_cache.backends.redis import RedisBackend
from fastapi_cache.decorator import cache
```

```
# in lifespan, after creating the client:
FastAPICache.init(RedisBackend(app.state.redis), prefix="fc")

@app.get("/expensive-report")
@cache(expire=600)
async def expensive_report():
    return await build_report()
```

19.4 Rate-limiting dependency

```
from fastapi import HTTPException

LUA = """
local c = redis.call('INCR', KEYS[1])
if c == 1 then redis.call('EXPIRE', KEYS[1], ARGV[1]) end
if c > tonumber(ARGV[2]) then return 0 end
return 1"""

async def rate_limit(request: Request, r: redis.Redis = Depends(get_redis)):
    ok = await r.eval(LUA, 1, f"rl:{request.client.host}", 60, 100)
    if not ok:
        raise HTTPException(429, "Too many requests")

@app.get("/api/data", dependencies=[Depends(rate_limit)])
async def data(): ...
```

19.5 WebSockets + Pub/Sub: real-time across multiple servers

One FastAPI process can push to its own WebSocket clients — but with 4 replicas behind a load balancer, a message published on server A must reach a user connected to server B. Redis Pub/Sub is the bridge:

```
from fastapi import WebSocket

@app.websocket("/ws/{room}")
async def ws_endpoint(ws: WebSocket, room: str):
    await ws.accept()
    r = ws.app.state.redis
    pubsub = r.pubsub()
    await pubsub.subscribe(f"room:{room}")
    try:
        async for msg in pubsub.listen():
            if msg["type"] == "message":
                await ws.send_text(msg["data"])
    finally:
        await pubsub.unsubscribe(f"room:{room}")

@app.post("/rooms/{room}/send")
```

```
async def send(room: str, text: str, r: redis.Redis = Depends(get_redis)):
    await r.publish(f"room:{room}", text)     # reaches EVERY server's
sockets
```

19.6 Background jobs: FastAPI producer, Streams consumer

```
# producer — endpoint returns instantly
@app.post("/orders")
async def create_order(order: dict, r: redis.Redis = Depends(get_redis)):
    await r.xadd("orders", {"payload": json.dumps(order)})
    return {"status": "queued"}

# consumer — separate worker process
async def worker():
    r = redis.Redis(decode_responses=True)
    try:
        await r.xgroup_create("orders", "workers", id="0", mkstream=True)
    except redis.ResponseError:
        pass                                    # group already exists
    while True:
        msgs = await r.xreadgroup("workers", "w1", {"orders": ">"},
                                  count=10, block=5000)
        for _stream, entries in msgs or []:
            for msg_id, fields in entries:
                await process(json.loads(fields["payload"]))
                await r.xack("orders", "workers", msg_id)
```

This gives you at-least-once delivery with ACKs — a lightweight Celery alternative with zero extra infrastructure.

Chapter 20 — 50 Interview Questions with Answers

Section A: Fundamentals (Q1–Q15)

Q1. What is Redis?

An in-memory key-value data store used as a cache, database, message broker, and queue. Data lives in RAM, giving microsecond reads/writes; optional disk persistence provides durability.

Q2. Why is Redis so fast?

Four reasons: (1) all data in RAM, no disk seeks; (2) single-threaded command execution, so no locking overhead; (3) highly optimized C data structures; (4) a simple protocol (RESP) with minimal parsing cost.

Q3. Redis is single-threaded — isn't that a weakness?

Command execution is single-threaded, which eliminates race conditions and lock contention; one instance still does 100k+ ops/sec because operations are memory-only and O(1)/O(log N). Since 6.0, network I/O uses multiple threads. To use more cores you run replicas or Cluster.

Q4. Name Redis's core data types.

String, List, Hash, Set, Sorted Set (ZSET); plus Streams, Bitmaps, Bitfields, HyperLogLog, Geo. Redis 8 adds JSON, time series, and probabilistic types into core.

Q5. Difference between Redis and Memcached?

Memcached: multi-threaded, strings only, no persistence, no replication — a pure cache. Redis: rich data types, persistence, replication, Lua, transactions, Pub/Sub, Streams. Memcached can edge out Redis for flat string caching at extreme thread counts; for everything else Redis wins.

Q6. What is a TTL and how do you set one?

Time-to-live: the key auto-deletes after the given time. SET key val EX 60, or EXPIRE key 60 on an existing key. TTL key shows remaining seconds (-1 = none, -2 = key doesn't exist).

Q7. How does Redis delete expired keys internally?

Two mechanisms: lazy (checked when a key is accessed) and active (a background job samples ~20 keys with TTLs 10x/second, deleting expired ones; it repeats immediately if >25% of the sample was expired).

Q8. Why is KEYS * dangerous and what should you use?

KEYS scans the entire keyspace in one blocking O(N) call — on millions of keys it freezes the server for every client. Use SCAN, which iterates in small non-blocking chunks with a cursor.

Q9. DEL vs UNLINK?

Both remove a key. DEL reclaims memory synchronously — a huge key blocks the server. UNLINK (4.0+) unlinks the key instantly and frees memory in a background thread. Prefer UNLINK for possibly-large keys.

Q10. What are the 16 databases in Redis? Should you use them?

Numbered logical namespaces (SELECT 0–15) in one instance. Considered legacy: no per-DB security, no per-DB memory limits, and Redis Cluster supports only DB 0. Use key prefixes or separate instances instead.

Q11. When would you choose a Hash over a JSON string?

Hash: flat object updated field-by-field (HSET one field, HINCRBY counters), memory-efficient listpack encoding for small hashes. JSON string: nested data always read/written whole. RedisJSON (core in Redis 8) covers nested-with-partial-update.

Q12. What is special about INCR?

It is atomic. Concurrent INCRs never lose updates — 1,000 simultaneous INCRs yield exactly +1,000. This makes Redis the default tool for counters and rate limiting without any locking.

Q13. How do you get the top 10 of a leaderboard?

Store scores in a sorted set: ZADD lb score member (or ZINCRBY for increments), then ZRANGE lb 0 9 REV WITHSCORES. Rank of one player: ZREVRANK lb member. All O(log N).

Q14. SMEMBERS vs SSCAN?

SMEMBERS returns the whole set in one call — fine for small sets, dangerous for millions of members (blocks and returns megabytes). SSCAN iterates in cursor-based batches safely.

Q15. What does SET key value NX do and why does it matter?

Sets the key only if it does not already exist — atomically. It's the primitive behind distributed locks, 'only-once' initialization, and deduplication. Combined with EX it gives a self-expiring lock.

Section B: Persistence, memory, architecture (Q16–Q30)

Q16. Explain RDB vs AOF.

RDB: periodic full snapshots via a forked child; compact, fast restart, but you lose everything since the last snapshot on crash. AOF: log of every write, fsync configurable (everysec loses ≤1s); bigger files, slower restart. Production best practice: both, with the 4.0+ hybrid format (RDB preamble inside the AOF).

Q17. What does appendfsync everysec mean?

The AOF buffer is fsynced to disk once per second. Crash cost: at most about one second of writes. It's the default because 'always' (fsync per write) costs significant throughput and 'no' risks ~30 seconds.

Q18. Can Redis be a primary database?

Yes, when the durability contract fits: AOF everysec + replication + failover means worst case ~1s of loss. Acceptable for sessions, carts, counters, feature flags; not for money movement or anything legally requiring zero loss.

Q19. What happens when Redis hits maxmemory?

The maxmemory-policy decides: noeviction returns errors on writes; allkeys-lru/lfu evicts across all keys; volatile-* evicts only keys that have TTLs. For a cache use allkeys-lru or allkeys-lfu; for a store use noeviction and alert on memory.

Q20. LRU vs LFU eviction — which and why?

LRU evicts what was least RECENTLY used; LFU (4.0+) what is least FREQUENTLY used. LFU is usually better for caches: a one-time bulk scan touches many cold keys once, which under LRU would evict genuinely hot keys.

Q21. What is a big key and why is it a problem?

A single key with a huge value (multi-million-item list/hash/set). Problems: DEL blocks the server, Cluster slot migration stalls, single reads return megabytes, replication spikes. Detect with redis-cli --bigkeys; fix by bucketing into smaller keys, trimming (LTRIM/XTRIM), and deleting with UNLINK or in SCAN batches.

Q22. What is a hot key and how do you handle it?

One key receiving a disproportionate share of requests (celebrity profile, global config). It can saturate one core or one cluster shard. Fixes: read from replicas, duplicate the key with random suffixes, cache in-process, or Redis 6 client-side caching.

Q23. How does Redis replication work?

Asynchronous primary→replica streaming. The replica first receives a full RDB sync (or diskless stream), then a continuous command stream. Async means the primary doesn't wait — so an abrupt primary failure can lose the last moments of writes. WAIT can require N replica ACKs for critical writes.

Q24. What is Redis Sentinel?

A distributed monitoring system (run ≥3 nodes) that detects primary failure, holds a quorum vote, promotes a replica, reconfigures the topology, and informs clients — automatic failover without sharding.

Q25. How does Redis Cluster distribute data?

16384 hash slots; slot = CRC16(key) mod 16384; slots are assigned to master nodes, each with replicas. Cluster-aware clients cache the slot→node map and follow MOVED/ASK redirects during resharding.

Q26. What are hash tags in Cluster?

Braces select the hashed substring: {user:1}:a and {user:1}:b hash on 'user:1' and land in the same slot — required for multi-key ops (MGET, SINTER, MULTI, Lua) which cannot span slots.

Q27. Do transactions support rollback?

No. MULTI/EXEC guarantees the queued commands run contiguously (isolation), but a runtime failure of one command does not undo the others. Use Lua when you need check-then-act atomicity.

Q28. What does WATCH do?

Optimistic locking: WATCH key, read it, then MULTI/EXEC. If the watched key changed before EXEC, EXEC aborts (nil) and you retry. Compare-and-set without blocking.

Q29. Pipelining vs transaction — the difference?

Pipelining batches commands into one network round-trip purely for speed; other clients may interleave server-side. A transaction (MULTI/EXEC) guarantees no interleaving but doesn't reduce round-trips by itself. redis-py pipelines are transactional by default (transaction=True).

Q30. Why Lua scripts (or Functions) instead of transactions?

A Lua script can read a value and make decisions on it mid-execution, atomically — MULTI/EXEC cannot branch on intermediate results. Rate limiters, token buckets, and safe lock release all need Lua. Redis 7 Functions add server-side storage and clean replication of that logic.

Section C: Patterns and production (Q31–Q42)

Q31. Explain the cache-aside pattern.

App checks Redis first; on miss it queries the DB, writes the result to Redis with a TTL, and returns it. The DB stays the source of truth; Redis failure degrades performance, not correctness.

Q32. What is a cache stampede and how do you prevent it?

Many concurrent misses on the same expired hot key all hit the database at once. Prevent with TTL jitter (randomized expiry), a rebuild mutex (SET NX so one request rebuilds while others wait/serve stale), and probabilistic early refresh.

Q33. How do you build a distributed lock in Redis, correctly?

Acquire: SET lock:res <random-token> NX EX 30. Release: a Lua script that deletes only if the stored token matches yours — otherwise a slow client could delete someone else's lock after its own expired. For strict correctness under partitions, prefer a consensus system; Redis locks are efficiency locks.

Q34. Design a rate limiter for 100 requests/minute per user.

Fixed window: INCR rl:{uid}:{minute}, EXPIRE 60, reject if >100 — simple, allows 2x bursts at boundaries. Sliding window: ZADD timestamps, ZREMRANGEBYSCORE old, ZCARD — accurate, more memory. Token bucket via Lua — smooth bursts; what most API gateways use.

Q35. Pub/Sub vs Streams — when each?

Pub/Sub: ephemeral fan-out (live dashboards, chat presence, invalidation broadcast); offline subscribers miss messages. Streams: persistent log with consumer groups, ACKs, replay, pending-entry recovery — use for job queues and event pipelines.

Q36. How do sessions in Redis work for a multi-server web app?

Session data keyed as session:<id> with a TTL equal to the session lifetime; every app server reads/writes the same Redis, so users stay logged in regardless of which server handles a request. Django: SESSION_ENGINE=cache backend.

Q37. How would you count unique daily visitors for 100M-user scale?

HyperLogLog: PFADD visitors:<date> <user-or-ip>, PFCOUNT for the count — 12 KB per day, ~0.8% error, PFMERGE for weekly/monthly rollups. Exact alternatives (sets, bitmaps keyed by user ID) cost real memory.

Q38. How does Celery use Redis?

As broker: tasks are LPUSHed to a list per queue, workers BRPOP them; results (if enabled) go into celery-task-meta-<id> keys with TTL. Keep the broker on a separate DB/instance from the cache so eviction can never drop jobs.

Q39. How do you find what is slow in a Redis deployment?

SLOWLOG GET for commands over threshold; redis-cli --latency and latency history for round-trip health; INFO commandstats for per-command usage; MONITOR only on non-prod. Then check for big keys, KEYS usage, missing pipelining, and swap/fragmentation.

Q40. Your cache hit rate is 40%. What do you look at?

keyspace_hits vs misses in INFO stats. Causes: TTLs too short, cache filled but evicting (check evicted_keys and maxmemory), key names not matching between writer and reader, caching data that's never re-read, or cold start after deploy. Fix TTLs, memory, and key discipline before adding RAM.

Q41. How do you secure a production Redis?

Private network only (bind), ACL users with least privilege (6.0+), TLS for transit, rename/disable FLUSHALL/CONFIG/DEBUG, protected-mode on, no internet exposure of 6379, and OS-level firewalling. Managed services (ElastiCache etc.) handle much of this.

Q42. ElastiCache/managed Redis vs self-hosted?

Managed: patching, failover, backups, scaling handled; slightly higher cost; some commands/CONFIG restricted. Self-hosted: full control, cheaper at scale, but you

own 3am failovers. Default to managed unless you have a platform team and a reason.

Section D: Versions and modern Redis (Q43–Q50)

Q43. What major features arrived in Redis 3.2?

GEO commands, BITFIELD, protected mode (response to mass attacks on open instances), and the quicklist encoding for lists. The dividing line between 'old' and 'modern' Redis.

Q44. What did Redis 4.0 bring?

Modules API (enabling RedisJSON/RediSearch), UNLINK and lazy freeing, LFU eviction, MEMORY commands, hybrid RDB+AOF persistence, PSYNC2, active defragmentation.

Q45. What is the headline feature of Redis 5.0?

Streams — the append-only log type with consumer groups (XADD/XREADGROUP/XACK), bringing Kafka-style durable messaging into Redis. Also ZPOPMIN/MAX and the SLAVEOF→REPLICAOF rename.

Q46. What did Redis 6.0 change for security and performance?

ACLs (per-user command/key permissions), built-in TLS, threaded I/O for network handling, RESP3, and client-side caching with server-driven invalidation.

Q47. What are Redis Functions (7.0) and why were they added?

Server-stored, named libraries of Lua code, persisted and replicated like data. They fix EVAL's weaknesses: scripts living only in application memory, NOSCRIPT cache misses, and awkward replication of script effects.

Q48. What is hash field expiration and when did it arrive?

Per-field TTL inside a hash (HEXPIRE/HPEXPIRE/HTTL), Redis 7.4. Before that, TTL applied only to whole keys — a decade-old feature request and a classic trick question.

Q49. Explain the Redis licensing saga and Valkey.

In 2024 Redis Inc. relicensed 7.4+ to RSALv2/SSPL (not OSI-approved), prompting the Linux Foundation-backed fork Valkey (from 7.2), adopted by AWS/GCP. In 2025 Redis 8 added AGPLv3, returning Redis to open source, and folded the Stack modules (JSON, query engine, time series, probabilistic) into core.

Q50. What's in Redis 8 for AI workloads?

Vector sets (beta) — native vector similarity search — plus JSON and the Query Engine in core, letting a RAG pipeline store documents, metadata filters, and embeddings in Redis without extra modules. Combined with Redis's classic role as an LLM response cache and rate limiter, one Redis can serve an entire AI backend's hot path.

Appendix A — Quick Command Reference

Category	Commands
Keys	SET GET DEL UNLINK EXISTS TYPE RENAME EXPIRE PEXPIRE EXPIREAT TTL PTTL PERSIST SCAN RANDOMKEY DBSIZE
Strings	SET (NX XX EX PX KEEPTTL GET) GETRANGE SETRANGE STRLEN APPEND MSET MGET INCR INCRBY INCRBYFLOAT DECR GETDEL GETEX
Lists	LPUSH RPUSH LPOP RPOP LRANGE LLEN LINDEX LSET LINSERT LREM LTRIM LMOVE BLMOVE BLPOP BRPOP LPOS
Hashes	HSET HGET HGETALL HMGET HDEL HEXISTS HLEN HKEYS HVALS HINCRBY HINCRBYFLOAT HSCAN HRANDFIELD HEXPIRE(7.4) HTTL(7.4)
Sets	SADD SREM SMEMBERS SISMEMBER SMISMEMBER SCARD SPOP SRANDMEMBER SINTER SUNION SDIFF SINTERCARD S*STORE SSCAN SMOVE
Sorted sets	ZADD ZINCRBY ZSCORE ZRANK ZREVRANK ZRANGE (REV BYSCORE BYLEX) ZRANGESTORE ZCARD ZCOUNT ZREM ZREMRANGEBYSCORE ZPOPMIN ZPOPMAX BZPOPMIN ZRANDMEMBER ZSCAN
Streams	XADD XLEN XRANGE XREVRANGE XREAD XGROUP XREADGROUP XACK XPENDING XCLAIM XAUTOCLAIM XTRIM XDEL XINFO
Bit/HLL/Geo	SETBIT GETBIT BITCOUNT BITOP BITPOS BITFIELD PFADD PFCOUNT PFMERGE GEOADD GEODIST GEOSEARCH GEOPOS
Pub/Sub	SUBSCRIBE UNSUBSCRIBE PSUBSCRIBE PUBLISH PUBSUB SPUBLISH(7.0) SSUBSCRIBE(7.0)
Tx/Scripting	MULTI EXEC DISCARD WATCH UNWATCH EVAL EVALSHA SCRIPT LOAD FUNCTION LOAD(7.0) FCALL(7.0)
Server/ops	INFO CONFIG GET/SET/REWRITE CLIENT LIST/KILL SLOWLOG MEMORY USAGE/DOCTOR DEBUG SLEEP MONITOR SHUTDOWN BGSAVE BGREWRITEAOF LASTSAVE REPLICAOF

Category	Commands
	WAIT ACL CLUSTER COMMAND

Full per-command documentation with complexity notes lives at redis.io/commands. Everything in this book was written to be typed, tested, and broken on your own machine — that is the only way it becomes yours. Happy hacking. — Coding India