Redis and Valkey

Version Redis 8.10 · Valkey 9.1 · verified 2026-11-02

Redis and Valkey data structures, TTL and eviction, cache-aside, stampedes, locks, streams, pipelines, persistence and the licence split.

Contents

Mental model 2

Data structures and their cost 2

TTL and eviction 4

Caching patterns and stampedes 6

Distributed locks 8

Pub/sub vs streams 10

Pipelines and transactions 12

Persistence 13

Redis vs Valkey 14

Further reading 15

Mental model

Redis and Valkey run every command on one main thread, one command at a time, against data structures held in memory. That makes each command atomic for free, and it makes every command's cost its time complexity times the size of the key it touches — one `O(N)` call on a big key stalls every other client. Choose the structure whose commands are cheap for your access pattern, and treat the server as a cache unless you have configured it to be more.

Data structures and their cost

Type	Core commands	Cost to watch
String	`SET`, `GET`, `INCR`	O(1)
Hash	`HSET`, `HGET`, `HGETALL`	`HGETALL` is O(N)
List	`LPUSH`, `RPOP`, `LRANGE`	`LRANGE` is O(S+N)
Set	`SADD`, `SISMEMBER`, `SMEMBERS`	`SMEMBERS` is O(N)

Type	Core commands	Cost to watch
Sorted set	`ZADD`, `ZRANGE … BYSCORE`	O(log N) per write
Stream	`XADD`, `XREADGROUP`, `XACK`	O(1) append

A key holds exactly one type, and commands for the wrong type fail with `WRONGTYPE`. Strings double as counters (`INCR` is atomic); hashes are objects; sorted sets are leaderboards, rate-limit windows and priority queues; streams are append-only logs with consumer groups.

```
redis-cli HSET user:42 name Ann plan pro  # 2
redis-cli HGET user:42 plan               # "pro"
redis-cli ZADD board 310 ann 120 bob      # 2
redis-cli ZRANGE board 0 0 REV WITHSCORES
# 1) "ann" 2) "310"
redis-cli INCR hits:2026-11-02            # 1
```

> **Gotcha:** `KEYS pattern` is O(N) over the whole database and blocks every client while it runs; the docs warn it "may ruin performance". Iterate with `SCAN 0 MATCH user:* COUNT 100` instead.

TTL and eviction

Setting	Effect
`SET k v EX 60`	Expire in 60 s
`HEXPIRE k 60 FIELDS 1 f`	Per-field TTL (Redis 7.4+, Valkey 9.0+)
`maxmemory-policy noeviction`	Default: writes fail when full
`allkeys-lru` / `allkeys-lfu`	Evict any key, approx. LRU / LFU
`volatile-*`	Evict only keys that have a TTL
`allkeys-lrm`	Least recently modified (Redis 8.6+)

Eviction only starts once `maxmemory` is reached. The default policy, `noeviction`, returns an error on writes instead of evicting, which is right for a primary store and wrong for a cache. For a pure cache, `allkeys-lru` is the documented “good default”.

```
redis-cli CONFIG SET maxmemory 2gb
redis-cli CONFIG SET maxmemory-policy allkeys-lru
redis-cli SET session:abc '{"u":42}' EX 1800
redis-cli TTL session:abc       # 1800
redis-cli PERSIST session:abc   # 1 — TTL removed
```

> **Gotcha:** the `volatile-*` policies “behave like `noeviction`” when no key has a TTL. A cache written without `EX` under `volatile-lru` fills up and then rejects every write with an OOM error.

Caching patterns and stampedes

Pattern	Write path	Risk
Cache-aside	App writes DB, deletes key	Brief stale reads
Write-through	App writes DB and cache	Cache holds unread data
Short lock on miss	One caller recomputes	Others wait or serve stale
TTL jitter	Random spread on expiry	None worth naming

Cache-aside is the default: read the cache, fall back to the database on a miss, then populate with a TTL. On write, delete the key rather than updating it, so a concurrent reader cannot overwrite fresh data with a stale value. A stampede is many callers missing the same hot key at once; a short `SET NX` lock lets one caller recompute.

```ts
import Redis from "ioredis";
const redis = new Redis();
async function cached(key: string,
  fn: () => Promise<string>): Promise<string> {
  for (;;) {
    const hit = await redis.get(key);
    if (hit !== null) return hit;
    if (await redis.set(`lock:${key}`, "1",
      "PX", 5000, "NX")) break;  // we recompute
    await new Promise((r) => setTimeout(r, 100));
  }
  const v = await fn();
  await redis.set(key, v, "EX", 300); // + jitter
  return v;
}
```

> **Gotcha:** if the loader takes longer than the lock's `PX`, the lock expires mid-load and a second caller starts recomputing too. Size the lock TTL above your slowest load, not your average one.

Distributed locks

Step	Command
Acquire	`SET lock:k <token> NX PX 30000`
Release (Redis 8.4+)	`DELEX lock:k IFEQ <token>`
Release (Valkey 9.0+)	`DELIFEQ lock:k <token>`
Release (older)	Lua: `GET` then `DEL` if equal

Acquire with a random token, a TTL and `NX`; release only if the key still holds your token, atomically. A plain `DEL` can delete a lock another client acquired after yours expired. The two engines added different commands for compare-and-delete, so check which one you run.

```
import Redis from "ioredis";
import { randomUUID } from "node:crypto";
const redis = new Redis();

const token = randomUUID();
const ok = await redis.set("lock:invoice:9",
token,
  "PX", 30000, "NX");
if (ok === "OK") {
  try { /* critical section */ }
  finally {
    await redis.call("DELEX", "lock:invoice:9",
      "IFEQ", token);  // Valkey: DELIFEQ key
token
  }
}
```

> **Warning:** a Redis lock is a lease, not a mutex. A GC pause longer than the TTL lets two holders act at once, and replication is asynchronous, so a failover can hand the lock out twice. The Redis docs tell you to “implement fencing tokens” when correctness matters.

Pub/sub vs streams

	Pub/sub	Streams
Delivery	At-most-once	At-least-once with groups
Offline subscriber	Misses messages	Reads from its last ID
Storage	None	Persisted, trimmed by `MAXLEN`
Ack and retry	None	`XACK`, `XAUTOCLAIM`

Pub/sub fans a message out to whoever is connected right now and keeps nothing. Streams append entries with IDs; a consumer group tracks each consumer's pending entries until it acknowledges them, so a crashed worker's messages can be claimed by another.

```
redis-cli XGROUP CREATE orders billing '$'
MKSTREAM
redis-cli XADD orders MAXLEN '~' 100000 '*' id 9
redis-cli XREADGROUP GROUP billing w1 COUNT 10 \
  BLOCK 5000 STREAMS orders '>'
# ack the entry ID that XREADGROUP returned
redis-cli XACK orders billing 1762128000000-0
redis-cli XPENDING orders billing   # unacked
```

> **Gotcha:** with `>` you only ever get new entries. After a crash, a consumer must read its own backlog with ID `0` (or another worker must `XAUTOCLAIM`), or those messages sit in the pending list forever.

Pipelines and transactions

Tool	Atomic	Round trips
Pipeline	No	1 for N commands
`MULTI` … `EXEC`	Yes, no rollback	1 if pipelined
`WATCH` + `MULTI`	Optimistic check-and-set	2+
Lua / `FUNCTION`	Yes	1

A pipeline only batches network round trips; other clients' commands can interleave. `MULTI` / `EXEC` queues commands and runs them back to back with nothing in between. `WATCH` aborts the `EXEC` (returns null) if a watched key changed, so you retry.

```
import Redis from "ioredis";
const redis = new Redis();

const res = await redis.multi()
  .incr("stock:sku1:sold")
```

```
  .decrby("stock:sku1", 1)
  .exec();
// [[null, 1], [null, 99]] — [error, result] pairs
```

> **Gotcha:** “Redis does not support rollbacks”. If one queued command fails at run time (say `INCR` on a hash), the others in the transaction still execute. `exec()` reports it per command.

Persistence

Mode	Default	Data at risk on crash
RDB snapshots	On: 3600 1, 300 100, 60 10000	Since last snapshot
AOF	`appendonly no`	~1 s with `appendfsync everysec`
Both off	`save ""`	Everything

RDB forks and writes a compact point-in-time file; the default rules save after an hour with 1 change, 5 minutes with 100, or a minute with 10000. AOF logs every write and is replayed on restart; with the default `everysec` fsync you can lose about a second of writes. Use both when the data matters; disable both for a pure cache.

```
redis-cli BGSAVE                      # fork + RDB
redis-cli INFO persistence | grep rdb_last_bg
redis-cli CONFIG SET appendonly yes  # starts AOF
redis-cli CONFIG GET appendfsync    # everysec
```

Redis vs Valkey

	Redis 8.10	Valkey 9.1
Licence	RSALv2, SSPLv1 or AGPLv3	BSD 3-Clause
Origin	Redis Ltd.	Fork from before the licence change

	Redis 8.10	Valkey 9.1
Built in	Query engine, JSON, time series	Core data structures
Lock release	`DELEX … IFEQ`	`DELIFEQ`
LRM eviction	Yes (8.6+)	No

Redis 8 moved to a choice of three licences, AGPLv3 among them; 7.2 and earlier stay BSD. Valkey forked “right before the transition” and stays BSD; Amazon ElastiCache now makes it the default engine for new clusters. The core commands, protocol and clients are shared, so `ioredis` and `redis-cli` work against both; newer commands diverge.

```
redis-cli INFO server | grep _version
redis-cli COMMAND INFO DELEX DELIFEQ  # nil = absent
```

Further reading

- Redis — Commands reference

- Redis — Distributed locks and the Redlock analysis
- Redis — Streams
- Valkey — Commands
- ioredis on GitHub