The Redis Architecture: Memory-First Persistence, Core Data Structures, and Distributed Caching Roles

Date: 2026-01-08

Executive Summary

- Redis is an in-memory data platform with optional persistence; optimize for low latency first, then add durability as needed.
- Choose persistence mode based on role: pure cache (often no persistence), read-heavy cache with warm restarts (RDB), or primary-ish durability (AOF everysec or mixed).
- Leverage the right data structure to avoid O(N) work in hot paths; prefer operations with predictable time and memory overhead.
- For scale and HA, use replication + Sentinel for failover or Redis Cluster for sharding; pick cache-aside for most app caches.
- Control memory deterministically with `maxmemory` and an eviction policy aligned to access patterns (often `allkeys-lfu`).

Memory-First Design

- Single-threaded event loop per instance: predictable low-latency operations; parallelize with multiple instances/cores.
- Data in RAM: sub-millisecond access; persistence is asynchronous and tuned to avoid impacting the hot path.
- Copy-on-write (COW) on forks: snapshotting (RDB/AOF rewrite) duplicates pages until writes occur; budget extra memory during persistence.
- Pipelining and batching: reduce round trips; Lua/EVAL for atomic multi-key operations without intermediate RTTs.

Key implications:

- Latency sensitivity: avoid large blocking ops on hot keys (e.g., massive `LRANGE`); consider pagination, streaming, or summary data.
- Size carefully: RAM-bound; estimate object overhead (encoding, pointers) not just data size.

Persistence Models (Trade-offs and Defaults)

- RDB (snapshotting):
  - How: periodic full snapshots (`SAVE`/`BGSAVE`).
  - Pros: smallest files, fastest restarts, minimal write amplification during runtime.
  - Cons: can lose last N seconds/minutes of writes; fork COW memory cost during save.
  - Use when: cache that can tolerate warm-up gaps; need fast cold start.
- AOF (append-only file):
  - How: append each write; fsync policy `always | everysec | no`.

    - Pros: better durability (with `everysec` ~1s loss, `always` ~0 loss), append semantics are simple; can rewrite (compact) in background.
    - Cons: larger files than RDB; rewrite also uses fork COW; slightly slower steady-state writes vs RDB-only.
    - Use when: semi-durable requirements, event sourcing, or fast recovery with minimal loss.
- Mixed (RDB preamble + AOF tail):
    - How: rewrite to RDB snapshot then append newer commands.
    - Pros: fast recovery + near-AOF durability.
    - Use when: need quick restarts with near-current state.

Practical defaults:

- Pure cache: disable persistence or RDB every few minutes; set `maxmemory` + eviction.
- Cache that must survive restarts: RDB every 5–15 min or mixed AOF (`everysec`).
- Primary-like persistence: AOF `everysec` + monitored rewrite; ensure disks and COW memory headroom.

Operational pitfalls:

- Fork memory spikes: budget ~copy size of live set during RDB/AOF rewrite.
- Disk stalls: place AOF/RDB on performant storage; monitor fsync latency.
- Rewrite cadence: schedule off-peak; use `auto-aof-rewrite-percentage/size`.

Core Data Structures (When to Use What)

- Strings: values up to 512MB; counters, blobs, JSON (via modules) or serialized objects.
    - Tip: use `INCRBY`, `GETRANGE`, bit ops for efficiency; prefer atomic ops over read-modify-write in app.
- Hashes: field/value maps; memory-efficient for many small attributes.
    - Tip: store object-like records; partial updates without reading the whole object.
- Lists: ordered sequences with head/tail ops.
    - Tip: queues, recent activity; avoid unbounded growth and large range scans on hot lists.
- Sets: unique members; fast membership and set algebra.
    - Tip: tags, unique visitors; beware very large cardinality on hot keys.
- Sorted Sets (ZSET): members with scores; ranking, leaderboards, time-ordered events.
    - Tip: sliding windows (trim by score/time) to cap memory.
- Streams: append-only logs with consumer groups.
    - Tip: durable-ish queues with replay; more operational complexity than simple lists.
- Bitmaps/Bitfields: compact boolean/counter flags by offset.
    - Tip: daily activity tracking at scale with minimal memory.
- HyperLogLog: cardinality estimations (approximate).
    - Tip: unique counts with tiny memory; not for exact values.
- Geospatial: radius and bounding-box queries.
    - Tip: nearest-location lookups; back by sorted sets under the hood.

Rule of thumb: choose the structure that lets Redis do the heavy lifting (membership checks, ranges, ranks) instead of the app.

Distributed Caching Roles and Patterns

- Topologies:
  - Primary + Replicas: scale reads; `Replica-Read` for heavy read traffic; failover via Sentinel.
  - Redis Cluster: sharding across nodes; auto-resharding and multi-master; clients must be cluster-aware.
  - Client-side sharding: simple and fast; you manage resharding/failover logic.
- Caching patterns:
  - Cache-aside (most common): app reads cache, falls back to DB on miss, then populates cache. Simple and reliable.
  - Read-through: cache layer loads on miss automatically (proxy/sidecar); centralizes loading logic.
  - Write-through: writes go to cache then backing store; strong consistency but higher write latency.
  - Write-back (write-behind): batch async DB writes; fastest writes but risk of data loss on cache failure.
- Invalidation strategies:
  - TTL-based: set per-key expirations; easiest way to bound staleness and memory.
  - Event-driven: delete/update keys on source-of-truth changes; requires discipline to avoid drift.
- Eviction policies (with `maxmemory`):
  - `allkeys-lfu`: great general-purpose for skewed traffic; favors hot keys.
  - `allkeys-lru`: simpler alternative; common default for caches.
  - `volatile-ttl`/`volatile-lru/lfu`: evict only keys with TTL; useful when only some data should be evictable.

Consistency notes:

- Replication is async; read-your-own-writes may not hold on replicas. For strictness, pin reads to primary or use `WAIT` for acked replicas.
- Cluster multi-key operations require keys in same hash slot (use hash tags like `{user:123}`).

Performance Playbook (Quick Wins)

- Use pipelining/batching to cut RTTs in hot paths.
- Prefer atomic Redis ops over round-trip sequences; consider Lua for small, critical multi-step logic.
- Right-size object lifetimes: set TTLs on cache keys; prevent unbounded growth.
- Cap collection sizes: trim lists/zsets; partition very large datasets.
- Monitor key metrics: ops/sec, hits/misses, memory used, evictions, fork time, AOF fsync latency.

Configuration Cheatsheet (Safe Starting Points)

- Pure cache:
  - `maxmemory <bytes>` and `maxmemory-policy allkeys-lfu`
  - Persistence: off, or RDB `save 900 1` for occasional warm restarts
- Warm, read-heavy cache:
  - RDB every few minutes or AOF `everysec` with rewrite
- Semi-durable store:
  - AOF `appendfsync everysec`, enable automatic rewrite; ensure SSD and memory headroom for COW
- HA:

    - Replication + Sentinel for failover; or Redis Cluster for sharding at scale

Common Pitfalls to Avoid

- Running snapshots/rewrite without enough free RAM for COW → OOM/latency spikes.
- Large blocking operations on hot keys (full list scans, giant `SMEMBERS`): redesign with paging or summaries.
- Forgetting TTLs/eviction on caches → memory bloat and sudden evictions under pressure.
- Treating replicas as strongly consistent → serve stale reads unexpectedly.
- Ignoring fsync/storage performance for AOF → latency hiccups and rewrite stalls.

Final Take

Start with the role: cache vs store. Pick persistence that matches loss tolerance, enforce `maxmemory` with an LFU/LRU policy, and select the data structure that turns your workload into O(1) or efficient range ops. Scale reads with replicas, scale capacity with Cluster, and keep an eye on fork/COW and storage latency.