# Redis Notes

These are some notes about Redis that I thought were worth writing down. Far from being complete, far from being right. Just notes...

## Table of Contents

## What is Redis?

The name stands for: *REmote DIctionary Server*.

- It is an open source (BSD licensed), super fast, noSQL *in-memory cache/database/message broker*
- Also said to be a *Key -> Data Structure store* (vs. key-value store, more than String)
- In-memory, yes, but has *disk persistence*, so you survive a crash
- Clustering, HA supported
- Very fast paced open source development since 2009, huge active community
- Backed by commercial company *Redis Labs*
- Supports *pipelines* and *basic transactions*
- Supports LUA scripting (LUA compiles into memory) and modules to add flexibility
- Redis Cluster support for NAT / Docker in v4.0
- Very efficient but *very simple*, low level tool (compared to SQL db indexes, query language, high level clustering, etc.)
- Experiment with Redis in docker-compose!

### As a cache

- Configurable eviction, key expiration
- Optional data persistence (unusual for a cache system) so this cache can recover from a melt-down (does not need to start "cold")
- "Intelligent" cache, because the data can be analyzed/queried with a lots of commands
- Binary safe (no encoding)
- String value is 512MB!
- Ideal for:
  - Plain strings
  - Full JSON objects
  - Binary file content (e.g. for in-memory image manipulation, see BITOP)
  - Raw bits/flags/counters (e.g. real time metrics), see SETBIT, INCR, INCRBY

### As a Database

- In addition to String it also has: list, set, sorted set, hash (+bitmap +hyperloglog)
- 180 high performant commands (server side execution), see cheat-sheet
- Can be used as *1st class database* in some cases

### As a Pub/Sub messaging system

> https://redis.io/topics/pubsub

- Very simple, fast and scalable async messaging
- With some serious Limitations:
  - No persistence or value caching

    - No delivery guarantees
    - No cluster optimization... yet
- See this pubsub introduction for more details.
- This article shows the internals of how Redis pubsub is implemented.

### Very fast (that's the reputation)

- Written in C
- *Single-threaded server* (actually modern versions of Redis use threads for different things). It is not designed to benefit from multiple CPU cores, but most of the time CPU is not the bottleneck, memory and network are!.
- You can benchmark it yourself
- Runs on *ARM processors*
- Base binary is <4MB, very small memory/CPU footprint, can be *embedded in IoT* devices (think Raspberry Pi)

### "Traditional" Use Cases

Here is an early post by antirez, Redis' lead developer (from 2011!) which explains that Redis doesn't necessarily has to replace other databases, it can solve "old" problems, like these:

- Counting stuff
- User session store
- Recent visitor list
- Leader board, user voting (show top users/players based on score criteria)
- Expired items in list (like user session expiration)
- Unique items in a given amount of time
- Real time basic analysis for stats, like inbound traffic tracking from IPs for DDOS detection)
- PubSub
- Simple Queues
- Geolocation database
- Distributed locks (e.g. Redlock)
- Autocomplete word database

### More "Interesting" Use Cases

- A buffer or pre-processor *in front of Mongo, MySQL* for write-intensive operations (like incoming Big Data), see RedisLabs White Paper on the topic
- Database Caching Strategies Using Redis (Amazon)
- A real-time data update digesting and reporting for short lived live updates
- A IoT sensor data fed into Redis for live reports, archived later

## Data types and abstractions

> https://redis.io/topics/data-types-intro

**Strings**

> https://redis.io/commands#string

- Binary safe strings

**Lists**

> https://redis.io/commands#list

- Linked lists
- Max 4 billion in size
- Ideal for queues, stacks, top N, etc.

**Sets**

> https://redis.io/commands#set

- Unique values
- Union, Diff between multiple Sets, see SINTER
- Extract random members, see SPOP

**Sorted Set**

> https://redis.io/commands#sorted_set

- Adds a score to the set value
- Can be used as indexes of other data

**Hash**

> https://redis.io/commands#hash

- name value pairs inside of a single key
- used for maps (like tables)

**Bitmap**

> https://redis.io/commands/setbit

- Operations on binary data in memory, see BITOP

**Hyperloglog**

> https://redis.io/commands#hyperloglog

- Probabilistic cardinality estimator
- Counts unique things statically

**Geospatial**

> https://redis.io/commands#geo

- Members of a set representing lat/lon
- Functions that return distance between 2 members and a list members in a given radius

**PubSub**

> https://redis.io/topics/pubsub

- Simple pubsub messaging

## Clients

There is a dedicated page on Redis Clients. Some examples:

- Java: jedis - Supports connection pooling, pubsub, pipelines, transactions, LUA scripting, Sentinel, Cluster
- NodeJS: ioredis - Supports pipelines, pubsub, LUA scripting, Sentinel, Cluster
- C: hiredis - Super fast, minimalist client

## Scaling and HA

There are multiple overlapping solution to scaling and high availability, some developed by the open source Redis project (Sentinel and Redis Cluster), some by RedisLabs (RLEP) and some by independent projects (twemproxy) with different goals and features.

**Replication**

> https://redis.io/topics/replication

- Master to Slave replication and failover allows virtually no down time when Master goes down.
- In a common scenario Master can interact with clients, while slave stores replicated data to disk

**Redis Sentinel**

> https://redis.io/topics/sentinel

- Node monitoring, notification and failover management process, independent from Redis Cluster
- Sentinel need "smart" (sentinel aware) Redis clients, OR having HAProxy in front of Redis to redirect to slave in case of master failure, OR use virtual IPs

- Experiment with Redis Sentinel in docker-compose!

**Partitioning**

There are usually three different avenues for partitioning data with Redis—client-side partitioning, proxy assisted partitioning, and query routing.

- In **client-side partitioning**, the partitioning logic is contained in the client code that selects the correct partition or Redis node based on either an algorithm, storing extra information, or some combination of the two.
- With **proxy-assisted** partitioning, Redis clients connect to a proxy middleware (like twemproxy ) that then routes the client's requests to the correct Redis node.
- The final implemented avenue for Redis partitioning is **query routing** where any client querying a random node in the cluster will be routed to the correct node containing the key, the approach taken in the current implementation of Redis cluster.

**twemproxy (nutcracker)**

> https://github.com/twitter/twemproxy

Twemproxy is an open source project released by Twitter for creating a caching proxy between a client and backend made up of either Memecache or Redis instances. Twemproxy separates the client calls, in our case any suitable Redis client, from the datastore backend through the use of an intermediary middleware. This middleware then implements a sharding strategy based on preferences that are set in a configuration YAML file.

- Written in C.
- Lightweight proxy for Memcached and Redis
- Connection pooling (even per server)
- Pipelining of requests and responses
- Automatic sharding across multiple servers, without either the client or the servers knowing about it (!)

**Redis Cluster**

> https://redis.io/topics/cluster-spec

- Data sharding and replication strategy with re-sharding between nodes while the nodes are running, with failover support.
- Cluster needs "smart" (cluster aware) Redis clients

**Redis(e) Pack from RedisLabs**

> https://redislabs.com/products/redis-pack/

- Offers a proxy based independent clustering (sharding) solution for $$.

- Experiment with Redis(e) Pack in docker-compose!

### Why you shouldn't READ from Slaves

...if you care about failover related downtimes.

Source: https://www.youtube.com/watch?v=wdPqFa3ru6U&list=PL83Wfqi-zYZF1MDKLr5djmLYUI0woy1wi&index=48 and in writing http://lolpack.me/rediswhitepaper.pdf

**Case Study - What happens during failover (using Sentinel)?**

Steps of failover -- 1GB in memory

1. Master is unreachable
2. Sentinels (as they discover that they haven't heard from Master for a while) will reach quorum and initiate failover -- 30 sec, configurable
3. A Slave is elected as the new Master
4. Master starts serving requests, but the Slave(s) not yet!
5. The new Master does a full `BGSAVE` (dumps last known state to disk) -- ~9 sec
6. The new Master syncs data to discoverable Slaves. During this time the Slaves are **NOT** serving traffic -- ~40 sec, over very fast Internet connection
7. Slaves load data from disk into memory -- ~8 sec
8. Slaves start serving traffic

Downtime for Slaves:

- 1GB of data: ~1.5 min.
- 5GB of data: ~3 min.
- 20GB of data: ~12.5 min!
- 40GB of data: ~18 min!!!

That's when people start asking "Why can't you just restart it??"... Well, restarting would do nothing, the loading from disk would happen there too...

**Conclusion**

Do NOT interact with Slaves if you care about failover related down times. Interact with Masters exclusively, for both READs and WRITEs. The Slaves will sync up in the background during failover.

## Tips for operations

- `redis-cli monitor` - this command, instead of going into the interactive mode, it will output commands received by that redis server, live, kinda like tail. Good for basic (low traffic) monitoring.

- `INFO replication` - This Redis command tells you if the server you are connected to is a Master or slave and other replication details.

- SAVE and BGSAVE - save the dump.rdb (or whatever file name is configured in redis.conf) to disk.
- Simple data recovery: dump.rdb can be copied into the filesystem of a stopped Redis instance, and when Redis starts, it will restore its state from it.
- **NEVER** use even number of sentinels. They might not be able to pick a new master when half of the sentinels vote for one and the other half for another node. **ALWAYS** use odd number of sentinels, at least 3.

## Further Reading

### Self learning

- https://redis.io/documentation
- https://www.youtube.com/watch?v=Hbt56gFj998
- https://www.youtube.com/watch?v=jTTlBc2-T9Q
- https://gemalto.udemy.com/learn-redis/learn/v4/overview
- https://redislabs.com/resources/ebook/
- http://openmymind.net/redis.pdf
- https://try.redis.io/
- https://www.youtube.com/watch?v=qHkXVY2LpwU - Redis complementing MongoDb

### Clustering/Sentinel (Redis in Production)

- https://redis.io/topics/sentinel
- https://redis.io/topics/cluster-tutorial
- https://scalegrid.io/blog/high-availability-with-redis-sentinels-connecting-to-redis-masterslave-sets/
- http://code.flickr.net/2014/07/31/redis-sentinel-at-flickr/
- http://www.programcreek.com/java-api-examples/index.php?source_dir=wint-master/wint-framework/src/main/java/wint/help/redis/SentinelRedisClient.java
- Upgrading or restarting Redis without downtime: https://redis.io/topics/admin
- Clustering alternative: https://github.com/twitter/twemproxy - Proxy in front of Redis nodes, client sees a single Redis instance. Supports fail-over and limited sharding. See https://www.youtube.com/watch?v=3zxYaI3RQyM
- Redis Sentinel failover case study: https://www.youtube.com/watch?v=wdPqFa3ru6U&list=PL83Wfqi-zYZF1MDKLr5djmLYUI0woy1wi&index=48 . Very useful!