

Redis

* Inmemory: Caching, Rate limiting

* Traditional: Part of DB (Inmemory: Faster access), Remaining: Disk

* Redis: Completely on RAM (not on HDD/SSD)

* Key-Value DB, Powerful.

* eg: Stateless app (Central Storage System), Share data by multiple apps (bruteforce attack info): Block them at every service/app!

* Redis: Connect disconnected/loosely conn. Services share a

Common DB.

Redis cli:

* Key-Value

* Column

* Document

* Graph

* SET orgID '625721321'

* GET orgID

* Don't overwrite: SETNX orgID '625721322' → No effect as already there!

* Multiple keys:

MSET Key1 'value1' Key2 'value2' Key3 'value3'

MGET Key1 Key2 Key3 Key4

* Delete a Key:

DEL Key1

GET Key1 → #null

* INCR, DECR:

SET number 10

INCR number → #11

DECR number → #10

* Expire:

SET number 1

EXPIRE number 1 → after 1 second number is deleted

* TTL: Time to live:

SET number 1

TTL number (-1: Key exists, no expiration set)

(-2: Key not exists)

Returns expiry time remaining in sec.

SET and EXPIRE:

SETEX Key to 'Value'

Redis Info:

- * Single threaded process: even on multicore system
- * Intended to safeguard read/writes!
- * If multithreaded: Need locking (Performance will be worse)
- * Supports append only file persistence.

Blocks:

MULTI

SET hello 'world'

INCR number

EXPIRE hello 10

EXEC

—————> After this Run everything at once.

Sharding: Distribute dataset across multiple Redis Instances

Redis:

- * Open Source, BSD licensed, Advanced Key-Value Store
- * Data Structure Server (String, hashes, list, Sets...)
- * Written in C
- * High performance, Scalable web apps
- * Very fast
 - 110K SETS
 - 81K GETS
- In memory (use disk only for persistence)
- Rich data types
- Replicate to any no. of Slaves.
- * All Redis operations are atomic (update value will be received)
- * Multi utility tool: Cache, messaging queue (AS Redis supports Subscribe/publish)
- * Any short lived data (web app sessions, hit count)

Redis Config:

* CONFIG GET *

* CONFIG GET loglevel

* CONFIG SET loglevel 'SEVERE'

Data types:

1. String: (512 Mb)

SET name 'kukorialepoint'

GET name

2. Hashes:

* Collection of Key-value pairs (map b/w String fields & String values): represent objects.

* HMSET Key 'username' 'password' 'hello' 'hi' 'how'

HMSET

* HGETALL Key

Store user objects: up to $2^{32} - 1$ field value pairs
(more than 4 billion)

> HMSET hello hi how are you

> HGETALL hello

'hi'

'how'

'are'

'you'

3. List (List of strings: Sorted by Insertion order)

> LPUSH javakpoint java

> LPUSH javakpoint mongo

> LRANGE javakpoint 0 10

'mongo'

'redls'

LPUSH users 1 2 3 4

LINDEX users 0 → #1

LRANGE users 0 -1 (slice)

$2^{32} - 1$ elements

4. Sets

* Unordered Collection of String: 0(1) add, Remove, Search for existence.

> SADD Key1 redls

> SADD Key1 mongo

> SADD Key1 mongo

> SMEMBERS Key1 → 'mongo'

'redls'

5. Sorted Sets:

* Non-repeating Collection of String: Sorted. (say Small \rightarrow great)

Score
→ zadd key1 0 redis
→ zadd key1 0 mongo
→ zadd key1 0 rabbit
→ ZRANGEBYSCORE key1 0 1000

redis
mongo
rabbit

when to use NoSQL databases:

* Large amount of data:

* Highly replicable: No need for primary read/write & only

Secondary read-only nodes. **Consistent**

Horizontal Scaling: more machines

Vertical Scaling: more CPU, RAM

* while horizontal Scaling: RDBMS takes effort!

Sharding: Split/partition resources into smaller pieces & distribute to different computing resources. [use closer DB: ↓ latency]

* Read/write: higher in NoSQL (unstructured data).

* Traditional DB: Takes effort to change Schema

* Resource intensive app: don't scale well.

NoSQL: Scale at ease!

Redis:

* Data Storage (Durability guaranteed by Redis)

* Data Structures

- String
- List
- Set, Sorted Set
- Hash
- Bit array
- Hyperlog logs
- Streams
- Geospatial Indexes

* Supports publish/subscribe pattern.

multimodel databases:

- * Single Integrated backend serves app
- * Rich modules: pre built (JSON support, SQL, Image processing, Linear algebra, Indexes)

Why not single model:

* Bottleneck in access & represent data!

* multimodel databases handles

1. Relational
2. object-oriented
3. Key value
4. Wide Column
5. Document
6. Graph models.

* Store Structured & Semi Structured data

* Eliminates Fragmentation problem.

Top multimodel DB offers:

- Data storage, backup, End recovery
- Query, Index (query, use index: efficient query)
- ACID (compliant, fault tolerant)
- Integration, Advanced security.

Use case:

- * Auto complete, Result highlight
- * Real time analysis: Top score, Cost, post, bidding
- * Fraud detection: Spok trends
- * Gaming & leaderboards
- * Session management
- * Social apps
- * Recommendation manager
- * Cache, publish/subscribe pattern for incoming data
- * Job/queue management
- * Built in analysis
- * Native JSON handling.

Search:

- * Index, query: Search with high performances
- * Secondary indexes

Cache: B/w Server & app. (Firestore DB)

* No DB, No cables

* SET → To create data

* Writes to disk at varying time interval (durability in case of failure)

* `redis-cli -h <hostname> -p <port>`

* Keys can be anything (As Binary safe): Also use an image as key.

* But mostly strings (Common)

Hashes:

HSET house:5100 numBed 3 size 5000 hvac 'forced'

HGET house:5100 numBed → #3

Sorted Sets:

* Leaderboard (Score: Each member)

> ZADD users 31 Steve 2 Owen 13 Jake

> ZRANGE users 0 -1 (0 to end)

> ZRANGE users 0 -1 WITHSCORES

> ZREVRANGE users 0 -1 WITHSCORES

> ZINCRBY users 20 Jakob → Increment Jakob by 20 # 33

Hyperlog:

* Keep an Estimate Count of unique items (eg: Track Count of unique visitors to a website), maintains internal hash. (determine already there: If yes: Not entered)

> PFADD visitors 127.0.0.1 → 1 (new)
0 (If already exists)

> PFCOUNT → No. of unique hyperlogs

Pub/Sub:

* Redis can act as a fast & efficient means to exchange messages in a publisher/subscriber pattern.

* publisher creates key value pair: 0/more subscribers to receive messages.

> PUBLISH weather temp:85f

* The message is published on the channel weather

The client subscribed will receive

'message'

'weather'

'temp:85f'

PUBLISH weather: 54481 temp: 85f

PSUBSCRIBE weather: ^{*}

Geospatial Indexes:

* Latitude & Longitude data (distance)

- > GEODIST towers -89.500 44.500 tower1
- > GEODIST towers tower1 tower2 # calculate distance
- > GEODIST towers tower1 tower2 mi

Redis Streams:

* data is appended like a log file (So only stream)

XADD

XRANGE

Can view pending messages & do powerful operations'

Redis modules:

1. Redis Search: Full text search engine (with secondary indexing) powerful querying weighted search
2. RedisJSON: Store JSON, in-memory manipulation. (product catalogs, 3-rd party feeds).
3. Redis Timeseries: Store time series data (Added Timescale)
4. Redis Graph: Graph DB
5. Redis Bloom: Support additional probabilistic DS
6. Redis AI
7. Redis Gears: Batch/event driven processing.

* shard: takes care of a subset of data

* proxy (zero latency): proxy to appropriate shard (each node of cluster uses proxy).

* cluster manager: manages cluster health, monitoring (balance, shard, provision/deprovision)

A: write full / don't (no partials): Atomic

C: Data correct (Before/after write): Consistent

I: Each process: Separate (Isolation)

D: Durability: Ensure data persistence: Transaction Complete, retrieved in case of failure.

Reqs:

A: MULTI, WATCH, EXEC [Indivisible & Irreducible]

C: only permitted writes

I: Single threaded (only multi/Exec: executed)

D: Durability

Data persistence: (2 methods)

1. AOF (Append only file):

* Redis replies for each successful operation write

* AOF: applies to every shard

* write every second: safe but not safe (slower performance)

* Redis enterprise handles this differently! (writing): optimized

* performance unaffected as master shard unaffected.

2. Snapshot:

* point in time copy (For durability rather than as backup)

CAP theorem:

* Impossible for a network based service (server) to 111ly provide more than 2 out of

* Consistency (Conflict free replications)

* Availability (make copies across DCS)

* partition tolerance

Layers:

management layer: Administer cluster, placement of shards, Failure detection & mitigation

Data access layer: manage connections (with clients, pri/sec shard)

In memory replication using WAN: Sync!

Conflict free replicated Datatypes

* CRDTs: multiple copies stored across locations (Independently)

* update, Resolve in consistencies.

Conflict: which one to use (math rules)

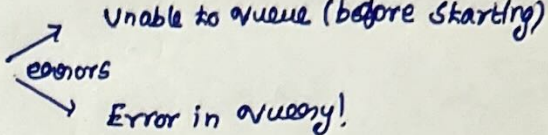
Faster, Fault tolerant.

Resources:

* Time

* Bk: No. of operations on bks generated to run an algorithm

* Space

Atomicity: Either occur/not! (MULTI, EXEC)
 

* EXEC: All done, All failed.

* Redis: doesn't support rollbacks!

WATCH myKey:

* Conditional 'EXEC': perform only when watched keys are unmodified.

* modification: client, Redis (expiration/eviction)

* when EXEC is called: all keys are unwatched!

'Optimized locking'

Inmemory DB:

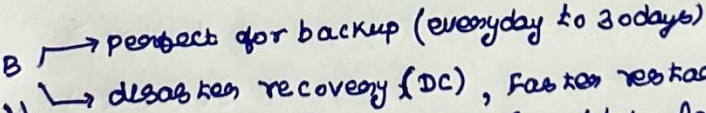
* Runs on RAM (fewer CPU instructions), volatile.

* Split data into multiple Redis instances.

Redis persistence:

* Redis DB: Snapshot in different time interval

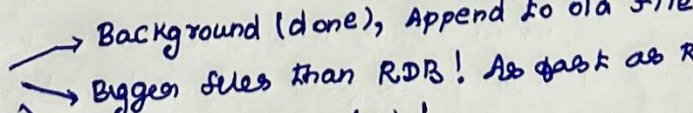
* AOF: write in logs (every operation): Do everything to restore!

RDB 

↳ disables recovery (DC), faster restore!

↳ Not good if backup time long (data lost high as interval ↑)

↳ Time consuming if done often affects performance.

AOF 

↳ Bigger files than RDB! As fast as RDB

↳ Memory usage high!

Snapshot

Save 60 1000 [every 60s, 1000 keys changed atleast]