Redis

* Inmemory: Caching, Rate limiting

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

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

* Key-Valued 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.

- * Key-value
- * Column
- * Document
- * Graph

Redis cli:

* SET orgID '625721321'

* GET orgID

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

* multiple sets:

  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 10 '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
```

EXEC → After this Run everything at once.

Sharding: Distribute dataset across multiple Redis instances

Redis:

- ★ OpenSource, BSD licensed, Advanced Key-Value Store
- ★ Data Structure server (String, hashes, list, Sets..)
- ★ Written in C
- ★ High performance, scalable web apps → In memory (use disk only for persistence) → Rich data types → Replicate to any no. of slaves.
- ★ Very fast → 110K SETS → 81K GETS
- ★ 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 'tutorialspoint'
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'
```

$\boxed{\text{HmSET}}$

```
* HGETALL Key
```

Store user objects : upto $2^{32}-1$ 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 JavatPoint Java
> LPush Javatpoint mongo
> Lrange Javatpoint 0 10
'mongo'
'redis'
```

```
>LPUSH users 1 2 3 4
>LINDEX users 0 → #1
>LRANGE users 0 -1 (slice)
```

$\boxed{2^{32}-1 \text{ elements}}$

4. Sets

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

```
> Sadd Key1 redis
> Sadd Key1 mongo
> Sadd Key1 mongo
> Smembers Key1 → 'mongo'
                  'redis'
```

5. Sorted sets:

* Non-repeating collection of String: Sorted. (Say small → 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. $\boxed{\text{Consistent}}$

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 closest 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 server app
- * Redis 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:
→ Datastorage, backup, End recovery
→ query, Index (query, use index: efficient query)
→ ACID (compliant, fault tolerant)
→ Integration, Advanced Security.

Usecase:

- * Auto complete, Result highlight
- * Real time analysis: Top score, cost, post, bidding
- * Fraud detection: spot 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 (Freeup dB)

*No DB, No tables

* SET ⟶ To create data

* Writes to disk at varying time interval (durability incase of failures)

```
* 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 usersfollowers 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: <Sup>*</Sup>
```

Geospatial Indexes:

* Latitude & Longitude data (distance)

```
> GEOADD 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, inmemory manipulation.
   (product catalogs, 3-rd party feeds).
3. Redis TimeSeries: Store time series data (Added timestamp)
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: Seperate (Isolation)

D: Durability: Ensure data persistence: Transaction complete, retrieved incase of failure.

Redis:

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: fast 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 |||ly provide more that 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 inconsistencies.

Conflict: which one to use (math rules)

Faster, fault tolerant.

Resources:

- ★ Time
- ★ Bit : no. of operations on bits required to run an algorithm
- ★ Space

Atomicity: Either occurs/not! (MULTI, EXEC) errors
- → Unable to queue (before starting)
- → Error in query!

- ★ 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:

- ★ Relies on RAM (fewer CPU instruction), 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
- → perfect for backup (everyday to 30days)
- → disaster recovery (DC), faster restart!
- → Not good if backup time long (data lost high as interval ↑)
- → Time consuming if done after affects performance.

Aof
- → Background (done), Append to old file
- → Bigger files than RDB! As fast as RDB
- → memory usage high!

Snapshot

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