REDIS 101

as cache =((

Agenda

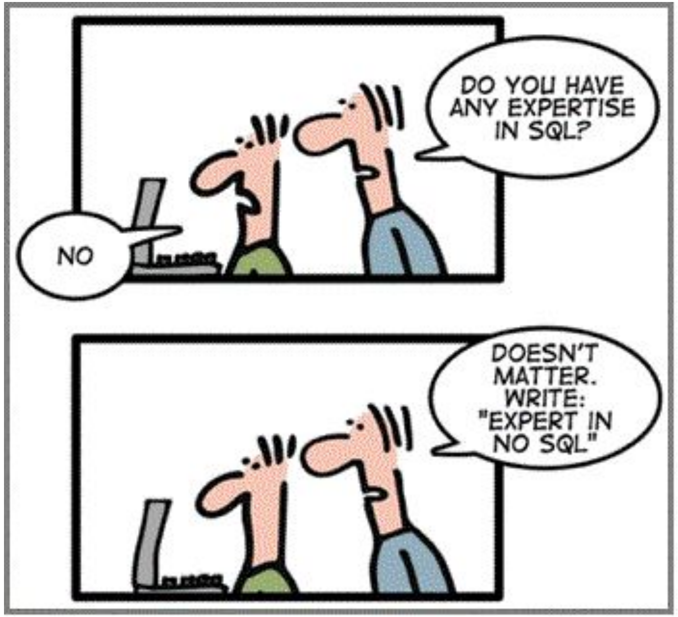
- What is Redis? Why Redis?
- Using Redis as cache, Redis caching strategies
- Redis data structures
- Back-up cache data with Redis
- Redis caching in Java

What is Redis?

- **In-memory data structure store**, used as a database, cache, message broker; that stores a mapping of keys to different value types
- Supports in-memory **persistent storage on disk**, replication, partitioning, …

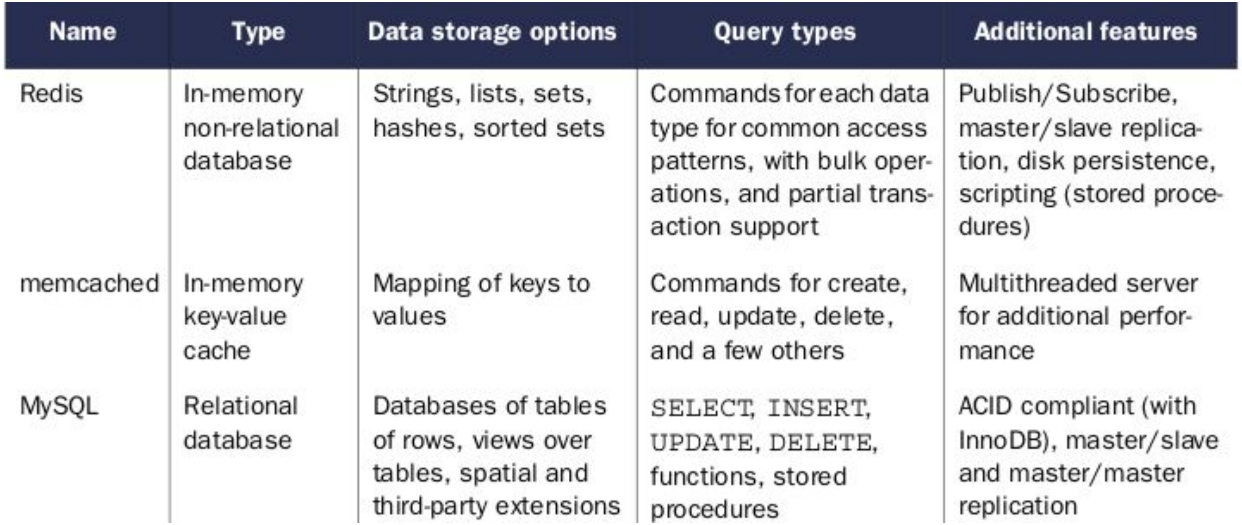
What is Redis?

- **Nosql** database
  - No tables or database-defined way of data relations
  - Do not use SQL to query data
- Redis compare to **memcached**?
  - In memory, key-value mappings, similar performance
  - Redis ~ single thread, while memcached ~ multithread
  - + Redis supports writing to disk automatically
  - + Multiple data structures beside plain string


DO YOU HAVE ANY EXPERTISE IN SQL?
NO
DOESN'T MATTER. WRITE: "EXPERT IN NO SQL"

What is Redis?

- Compare to other data storage?
    - **Redis** is prefer when the performance or functionality of Redis is necessary
    - **Other data storage** when slower performance is acceptable, data is too large to fit in-memory



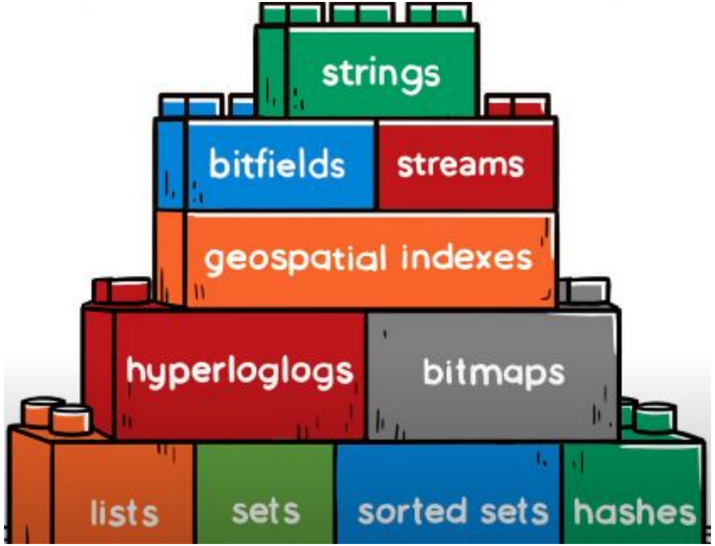
Name	Type	Data storage options	Query types	Additional features
Redis	In-memory non-relational database	Strings, lists, sets, hashes, sorted sets	Commands for each data type for common access patterns, with bulk operations, and partial transaction support	Publish/Subscribe, master/slave replication, disk persistence, scripting (stored procedures)
memcached	In-memory key-value cache	Mapping of keys to values	Commands for create, read, update, delete, and a few others	Multithreaded server for additional performance
MySQL	Relational database	Databases of tables of rows, views over tables, spatial and third-party extensions	`SELECT`, `INSERT`, `UPDATE`, `DELETE`, functions, stored procedures	ACID compliant (with InnoDB), master/slave and master/master replication

Why Redis?

- **High performance and low latency**
  - Written in C -> helps compiler optimize code better
  - Runs entirely in memory (> 1.5m ops/s & <1ms latency)
  - Single thread, atomic operations

- **Support pre build-in data structures**
  - Simplify application, allows data processed on db level

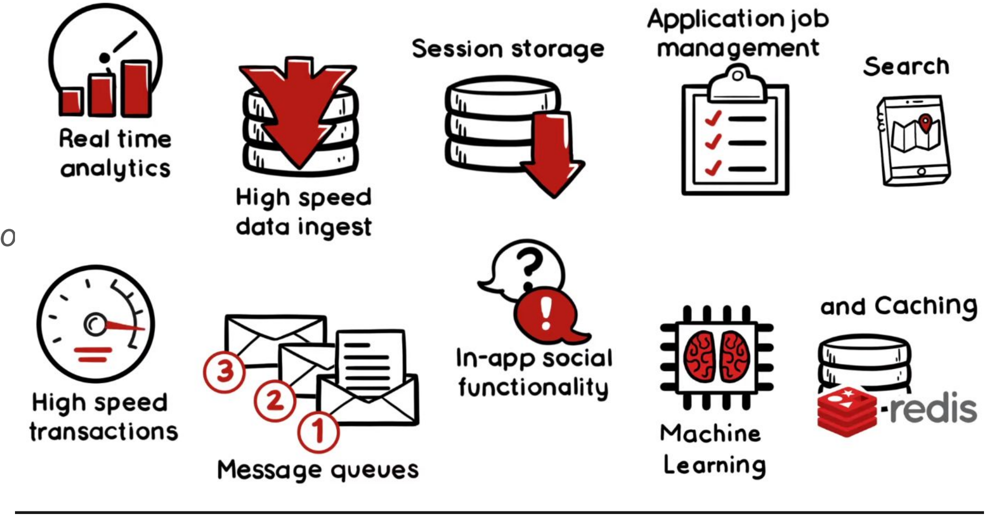
  → Reduce code complexity, bandwidth requirements



Why Redis?

- **Compatibility**
  - Open source and stable, used by tech-giants (GitHub, Weibo, Pinterest, Snapchat, StackOverflow, Flickr, ...)
  - Large supporting community for most of the languages
  - Cover most popular use cases

- ***Facts***:
  - *Fastest growing database 2013*
  - *Most loved database by developers (stacko*
  - *No.1 Nosql data store, also in containers*
  - *World fastest database*



Redis caching strategies

- **Cache aside (Lazy loading) -** most common strategy
  - Application first checks the cache
  - cache hit & cache miss
  - (+) Keep cache size cost effective
  - (+) Cache fault tolerance
  - (-) Overhead when cache miss
- **Write through**
  - Write is done synchronously both to cache and db
  - (+) Data persistence guaranteed
  - (-) Extra write latency

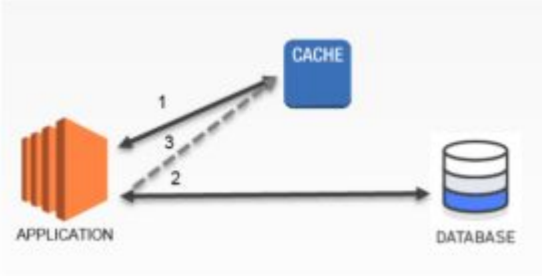


Figure 2: A cache-aside cache

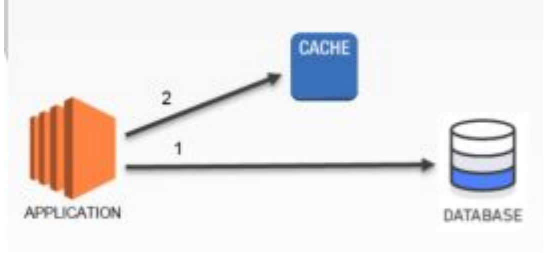
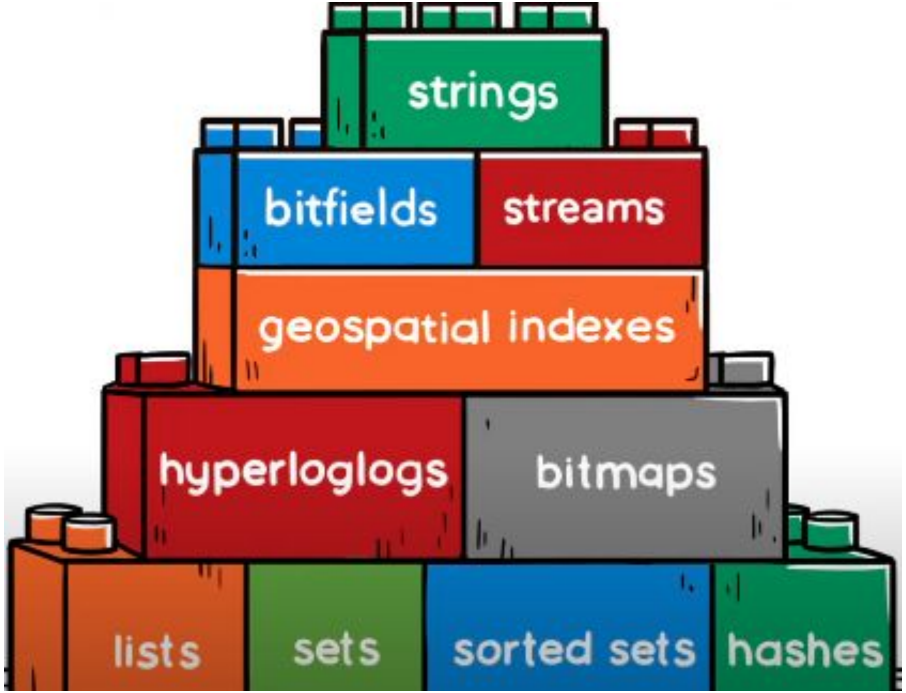


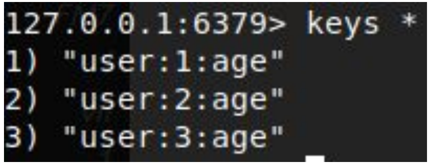
Figure 3: A write-through cache

Redis data structures

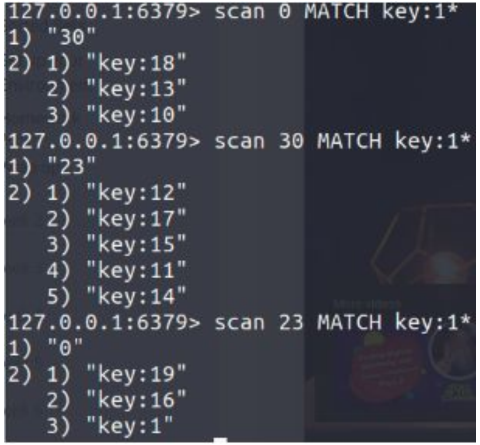


Keys

- **Binary safe** (sequence of bytes), maximum size = 512MB
- Key spaces:
    - Flat key space → need naming convention
    - No automatic namespace (ex: bucket, collection…)
    - e.g: “user:10134:friends-of-friends”
- Commands
    - **KEYS** (O(N))
        - Blocks until complete → Not safe for production
    - **SCAN** (O(1) for each)
        - Cursor based iterator, terminates when 0 is returned
        - (-) A given element may be returned multiple times
    - **Time To Live (TTL)** mechanism
        - **EXPIREAT** → seconds, milliseconds or Unix time epoch



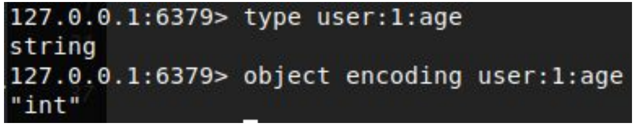
```
127.0.0.1:6379> keys *
1) "user:1:age"
2) "user:2:age"
3) "user:3:age"
```



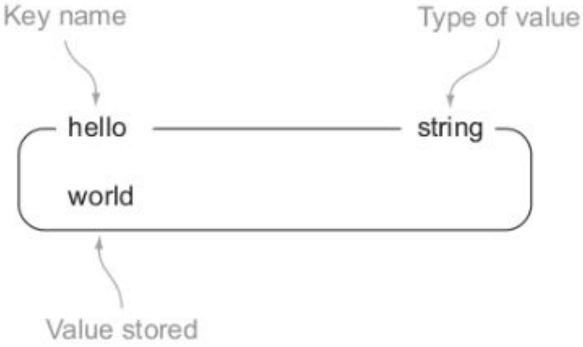
```
127.0.0.1:6379> scan 0 MATCH key:1*
1) "30"
2) 1) "key:18"
   2) "key:13"
   3) "key:10"
127.0.0.1:6379> scan 30 MATCH key:1*
1) "23"
2) 1) "key:12"
   2) "key:17"
   3) "key:15"
   4) "key:11"
   5) "key:14"
127.0.0.1:6379> scan 23 MATCH key:1*
1) "0"
2) 1) "key:19"
   2) "key:16"
   3) "key:1"
```

String

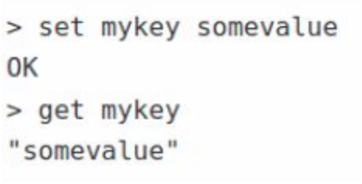
- Redis string ~ **Simple Dynamic String (SDS)** → binary safe, max size = 512Mb
- Can be manipulated as Text, Float, Integer (INCR and INCRBY)



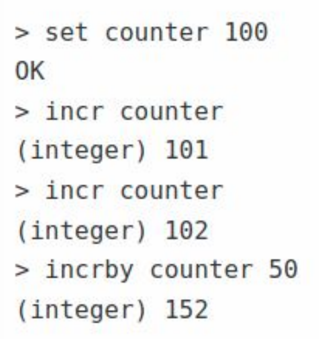
```
127.0.0.1:6379> type user:1:age
string
127.0.0.1:6379> object encoding user:1:age
"int"
```



- **Commands**



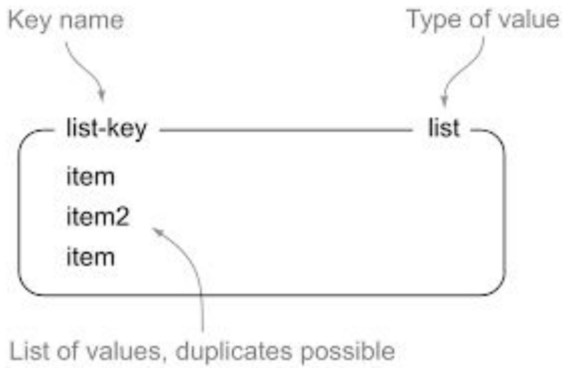
```
> set mykey somevalue
OK
> get mykey
"somevalue"
```



```
> set counter 100
OK
> incr counter
(integer) 101
> incr counter
(integer) 102
> incrby counter 50
(integer) 152
```

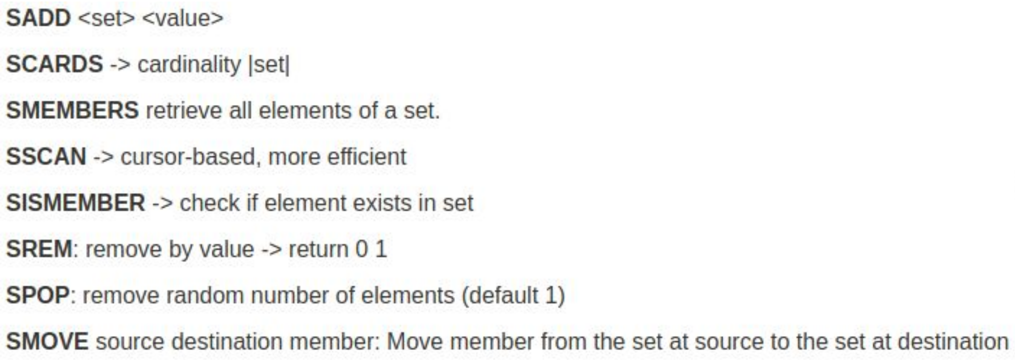
Lists

- Lists of strings, sorted by insertion order, duplicates are allowed
- Implemented as Linked List.
- Max length of a list is 2^32 - 1 elements
- Use cases
  - Store data sequentially in a Redis server where the write speeds are more desirable than read performance. E.g: log messages
  - Pub/Sub scenario: inter process communication
- **Commands**
  - **LPUSH, RPUSH, LPOP, RPOP, LLEN, LRANGE, LINDEX**



Sets

- Unordered collection of Strings contains no duplicates
- Implement hash table to keep items unique
- Max length of a list is 2^32 - 1 elements
- **Commands**



SADD <set> <value>

SCARDS -> cardinality |set|

SMEMBERS retrieve all elements of a set.

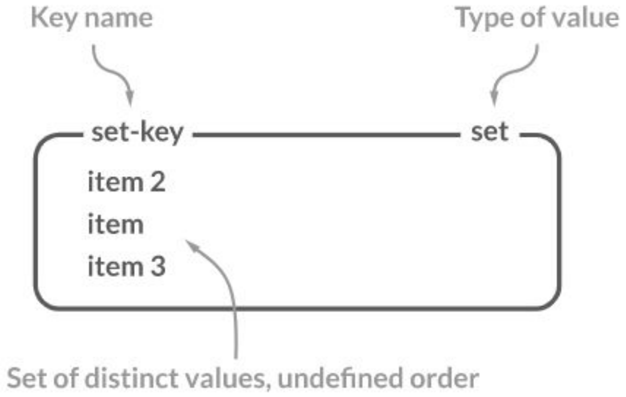
SSCAN -> cursor-based, more efficient

SISMEMBER -> check if element exists in set

SREM: remove by value -> return 0 1

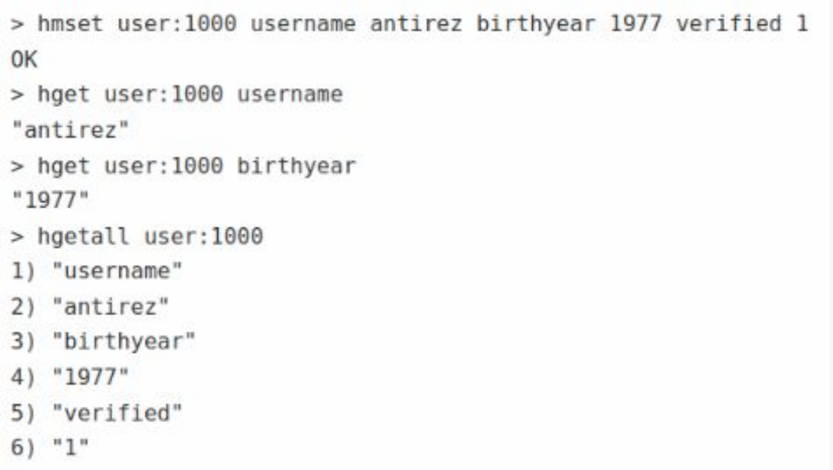
SPOP: remove random number of elements (default 1)

SMOVE source destination member: Move member from the set at source to the set at destination

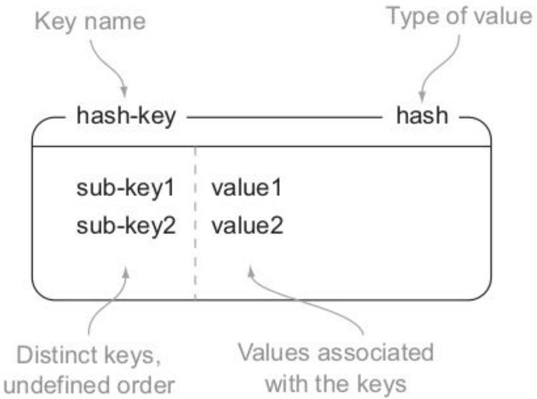


Hashes

- Mutable collections of field-value pairs
- Expiration time can only be defined on a Key, not a Field within a Key
- Use cases
  - store simple and complex data objects (ex: session cache)
- **Commands**

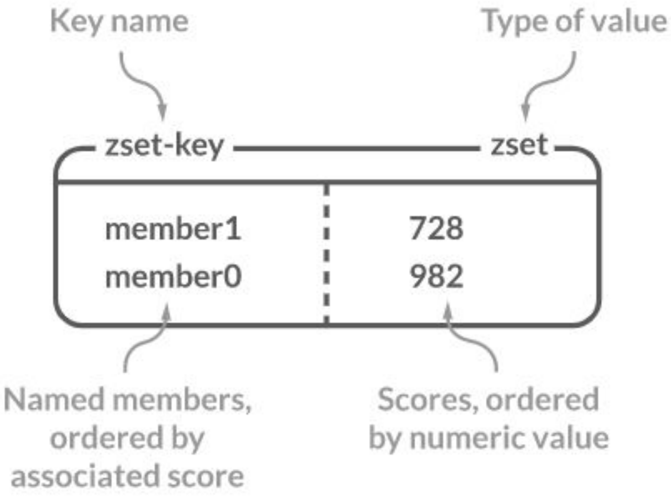


```
> hmset user:1000 username antirez birthyear 1977 verified 1
OK
> hget user:1000 username
"antirez"
> hget user:1000 birthyear
"1977"
> hgetall user:1000
1) "username"
2) "antirez"
3) "birthyear"
4) "1977"
5) "verified"
6) "1"
```



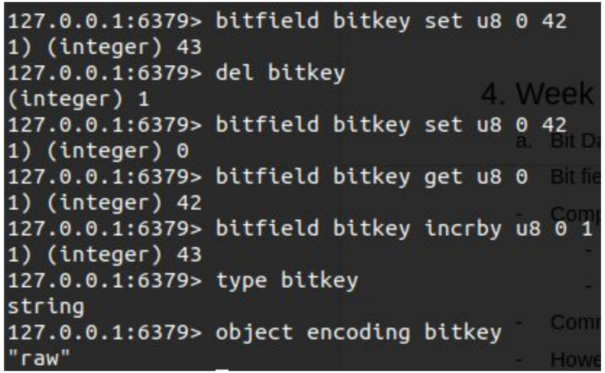
Sorted Set

- Mix between a Set and a Hash.
- Implemented as a skiplist → fast search linked list
- Keys (members) & Values (Scores: Float number)
- Ordering:
  - If A.score > B.score → A > B
  - A.score = B.score → lexicographic
- Commands
  - **ZADD**
  - **ZRANGE/ZRANK**
  - **ZREM**
  - ………


Key name
Type of value
zset-key
zset
member1
728
member0
982
Named members, ordered by associated score
Scores, ordered by numeric value

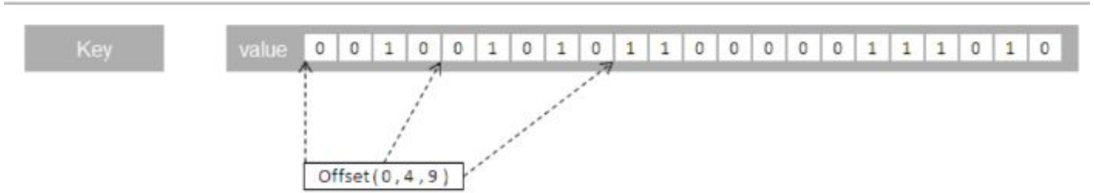
Bitmaps

- Sequence of bits where each bit can store 0 or 1
- memory efficient, support fast data lookups, and can store up to 2^32 bits
- under the hood Bitmap is a string
- Usecase
  - Store boolean information of a extremely large domain into (relatively) small space ~ decent performance.
- Commands



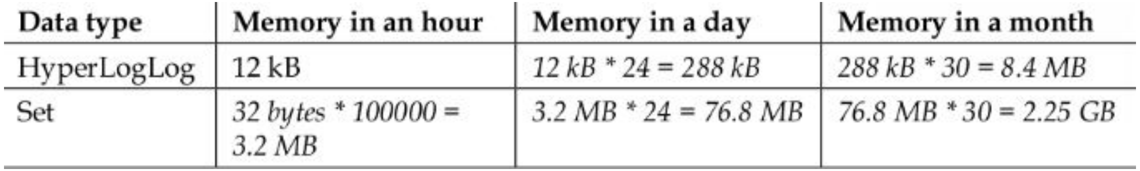
```
127.0.0.1:6379> bitfield bitkey set u8 0 42
1) (integer) 43
127.0.0.1:6379> del bitkey
(integer) 1
127.0.0.1:6379> bitfield bitkey set u8 0 42
1) (integer) 0
127.0.0.1:6379> bitfield bitkey get u8 0
1) (integer) 42
127.0.0.1:6379> bitfield bitkey incrby u8 0 1
1) (integer) 43
127.0.0.1:6379> type bitkey
string
127.0.0.1:6379> object encoding bitkey
"raw"
```

4. Week
a. Bit Da
Bit fie
- Comp
- Comm
- Howe



HyperLogLog

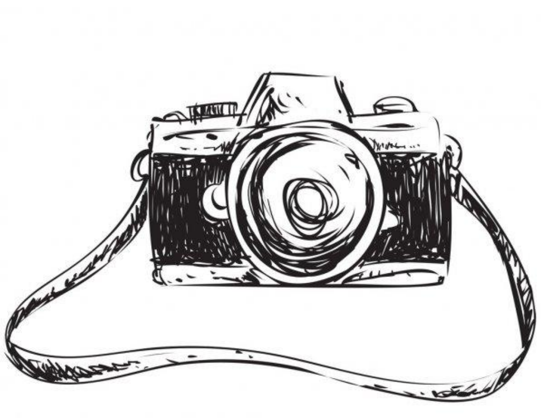
- Probabilistic data structure used to count unique things (set cardinality) → **efficient counting**
- Trade memory for precision: *standard error (< 1%)*
- Space efficient ㅡ the maximum size ~ 12kb per key
- Use case:
  - Count very large set that you don’t have a space for perfectly accurate counts.
  - E.g: counting # unique users who visited a website, unique words in a book
- HyperLogLog vs Set
  - Given 100,000 unique visitors (each has unique UUID 32 bytes string), Redis key is created per hour

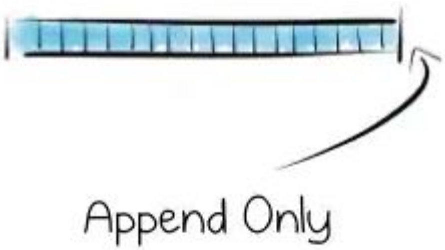


Data type	Memory in an hour	Memory in a day	Memory in a month
HyperLogLog	12 kB	*12 kB * 24 = 288 kB*	*288 kB * 30 = 8.4 MB*
Set	*32 bytes * 100000 = 3.2 MB*	*3.2 MB * 24 = 76.8 MB*	*76.8 MB * 30 = 2.25 GB*

- Commands: **PFADD, PFCOUNT, PFMERGE**

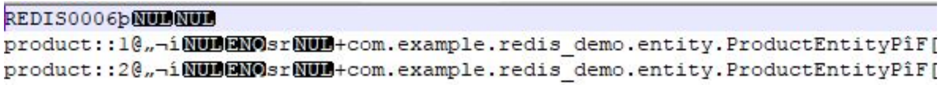
REDIS Persistence



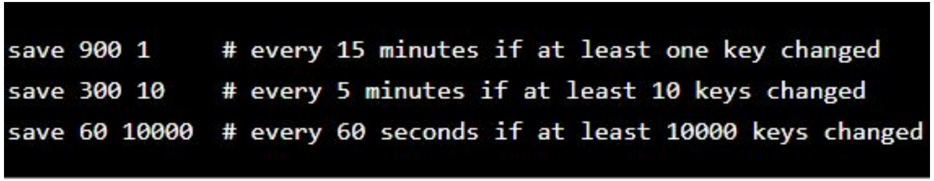


REDIS Database Backup (RDB)

- Point-in-time snapshots của toàn bộ db
- Pros:
  - Dữ liệu DB được lưu gọn trong một file
  - DB lên nhanh hơn so với dùng AOF
- Cons:
  - Do là point-in-time => có thể bị sót dữ liệu
  - Tốn tài nguyên do quá trình snap shot



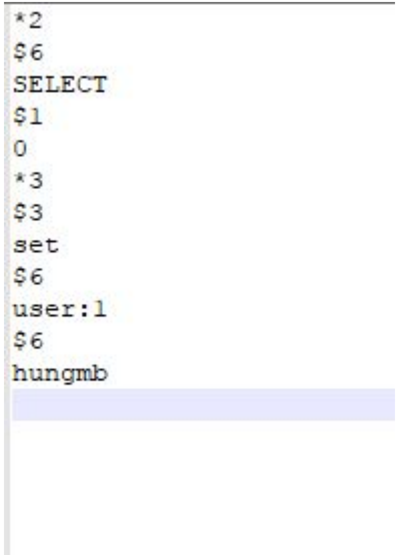
```
REDIS0006þNULNUL
product::1@„¬íNULENQsrNUL+com.example.redis_demo.entity.ProductEntityPîF[
product::2@„¬íNULENQsrNUL+com.example.redis_demo.entity.ProductEntityPîF[
```



```
save 900 1     # every 15 minutes if at least one key changed
save 300 10    # every 5 minutes if at least 10 keys changed
save 60 10000  # every 60 seconds if at least 10000 keys changed
```

Append Only File (AOF)

- Log của toàn bộ các câu lệnh write vào DB
- Pros:
    - Có khả năng ghi lại tất cả các câu lệnh real-time
    - Append only -> không hỏng cả file khi write bị lỗi
    - Dễ dàng quản lý các câu lệnh
- Cons:
    - Càng real-time thì càng tốn tài nguyên
    - AOF file thường lớn hơn RDB file
    - Tốc độ dựng lại DB chậm hơn RDB



```
*2
$6
SELECT
$1
0
*3
$3
set
$6
user:1
$6
hungmb
```

Use case của từng strat

- RDB

Khuyến nghị sử dụng nếu không yêu cầu 100% toàn vẹn dữ liệu (chấp nhận mất dữ liệu được lưu trước đó X(s) hoặc Y(câu lệnh)

- AOF

Không chịu mất Data thì phải chịu mất tài nguyên

Redis suggestion

- only cache, no need persistence -> không dùng cả 2
- cần persistence nhưng có thể chấp nhận data loss -> dùng RDB (faster reload/file size nhỏ hơn)
- cần đảm bảo persistence mức độ cao -> dùng AOF + RDB (AOF để đảm bảo lưu các command, RDB để back up cho AOF)

Caching bằng redis trong java

Redis Cache Configuration & Redis CacheManager

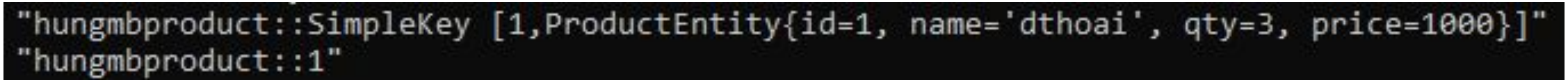
- Redis Cache Manager giúp
  - Cấu hình một số thông số của Cache (port, host, lib sử dụng…)
  - Định nghĩa trước một số Cache Name sẽ sử dụng (tên/Redis Config sử dụng cho cache name)
  - …
- Redis Cache Configuration giúp:
  - Thiết lập kiểu serialize cho key/value
  - Thiết lập giá trị mặc định của TTL
  - Đặt prefix cho key
  - …

Cache Annotations

- @Cachable: Đánh dấu method/class sẽ được cache
- @CachePut: Đánh dấu update lại giá trị của key cũ (nếu có) khi method được gọi
- @CacheEvict: Đánh dấu xóa giá trị key khi method được gọi

@Cacheable

- Các argument
  + key: chọn target để lấy làm key. Default = “ ” => lấy mọi params
  + value: alias cho cacheName
  + unless/constraint: đưa điều kiện để cache
- Giá trị được cache là giá trị trả về của method



```
"hungmbproduct::SimpleKey [1,ProductEntity{id=1, name='dthoai', qty=3, price=1000}]"
"hungmbproduct::1"
```

Redis Template

- Spring cung cấp Redis Template để thực hiện trực tiếp các operation trên Redis cache
- Ưu điểm của Redis Template:
  - Hỗ trợ sử dụng command của Redis
  - Hỗ trợ tương tác với Redis Data Structure
- Nhược điểm:
  - Phải config không dùng annotation
  - Phải học thêm hàm của RedisTemplate để tương tác với DB

Store object trong redis

- Store Json object trong một key, quản lý các object thông qua một list Id
- Store object trong một hash, quản lý các object thông qua list Id
- Store các object trong một hash với key là Id

Future work (if exists)

- Redis Pub/Sub pattern
- Redis Transaction
- Redis Stream
- Scaling with Redis: partitioning, Redis cluster, sentinel