Redis (Remote Dictionary Server) is an open source (BSD licensed), in-memory data structure store used as a database, cache, message broker, and streaming engine.

Redis supports asynchronous replication with fast non-blocking synchronization and auto-reconnection with partial resynchronization on net split.

Redis is a data structure server.

Extremely Fast: Can perform 110000 SET operations, 81000 GET operations per second.

Persistence: While all the data lives in memory, changes are asynchronously saved on disks.

**Command** **to run redis server**: redis-server --port 6380 --slaveof 127.0.0.1 6379

**DATA TYPES**

1. **Keys­­**

Redis keys are binary safe i.e. content of image can also be key. Empty string is also a valid key.

Very long/short keys are not a good idea. Try to stick with a schema.

Maximum allowed key size is 512 MB.

1. **Strings**

String data type is useful in caching HTML fragments or pages.

SET and the GET commands are the way we set and retrieve a string value.

**Commands**

SET mykey myvalue (Replace if any existing key is present) > OK

SET mykey newval nx (Fails if the key already exists) > (nil)

SET mykey newval2 xx (Only succeed if the key already exists) > OK

SET counter 100 > OK

INCR,INCRBY,DECR, DECRBY are atomic commands. Multiple clients issuing INCR against the same key will never enter into a race condition.

INCR counter > (integer) 101 //DECR is also there

INCRBY counter 50 >(integer) 151 //DECRBY is also there

The ability to set or retrieve the value of multiple keys in a single command is also useful for reduced latency. For this reason there are the MSET & MGET commands:

MSET a 10 b 20 c 30 > OK

MGET a b c > 1) "10" 2) "20" 3) "30"

**Altering and querying the key space**

 EXISTS command returns 1 or 0 & check if given key exists or not in the database

SET mykey hello > OK

EXISTS mykey > (integer) 1

TYPE mykey > (string)

DEL mykey > (integer) 1

EXISTS mykey > (integer) 0

TYPE mykey > none

**Key Expiration**

EXPIRE mykey 5 > deletes key after 5 seconds

TTL mykey > Time to live (returns time left in deletion)

SET mykey myvalue ex 20 > Sets a key and delete it after 20 seconds

1. **Lists**

Redis lists are linked lists of string values. Redis lists are frequently used to:

I. Implement stacks and queues.

II. Build queue management for background worker systems.

**Basic commands**

LPUSH/ RPUSH adds new element to the head/tail of a list.

LPOP/RPOP removes and returns an element from the head/tails of a list.

LLEN returns the length of a list.

LMOVE atomically moves elements from one list to another.

LTRIM reduces a list to the specified range of elements.

BLPOP/BRPOP removes and returns an element from the head/tail of a list. If the list is empty, the command blocks until an element becomes available or until the specified timeout is reached.

BLMOVE atomically moves elements from a source list to a target list. If the source list is empty, the command will block until a new element becomes available.

> rpush mylist A O/P (integer) 1

> rpush mylist B O/P (integer) 2

> lpush mylist first O/P (integer) 3

> lrange mylist 0 -1 // LRANGE KEY START END

1) "first"

2) "A"

3) "B"

> lpop mylist O/P “first”

> rpop mylist O/P “B”

>del mylist O/p (integer) 1

**Capped List**

It sets the range as the new list value instead of just displaying the new range.

> rpush mylist 1 2 3 4 5 O/P (integer) 5

> ltrim mylist 0 2 O/P OK

> lrange mylist 0 -1 O/P

1) "1"

2) "2"

3) "3"

1. **Hashes**

Redis hashes are record types structured as collections of field-value pairs. You can use hashes to represent basic objects and to store groupings of counters, among other things.

**Basic commands**

HSET sets the value of one or more fields on a hash.

HGET returns the value at a given field.

HMGET returns the values at one or more given fields.

HINCRBY increments the value at a given field by the integer provided.

Most Redis hash commands are O(1).

A few commands - such as HKEYS, HVALS, and HGETALL - are O(n),

Every hash can store up to 4,294,967,295 (2^32 - 1) field-value pairs.

**Commands**

> hset user:1000 username antirez birthyear 1977 verified 1

(integer) 3

> 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"

> hmget user:1000 username birthyear no-such-field

1) "antirez"

2) "1977"

3) (nil)

1. **Sets**

A Redis set is an unordered collection of unique strings (members). You can use Redis sets to efficiently:

I. Track unique items (e.g., track all unique IP addresses accessing a given blog post).

II. Represent relations (e.g., the set of all users with a given role).

III. Perform common set operations such as intersection, unions, and differences.

The max size of a Redis set is 2^32 - 1 (4,294,967,295) members.

**Basic commands**

SADD adds a new member to a set.

SREM removes the specified member from the set.

SISMEMBER tests a string for set membership.

SINTER returns the set of members that two or more sets have in common (i.e., the intersection).

SCARD returns the size (a.k.a. cardinality) of a set.

> sadd myset 1 2 3

(integer) 3

> smembers myset

1. 3

2. 1

3. 2

> sismember myset 3

1. **Sorted Sets**

A Redis sorted set is a collection of unique strings (members) ordered by an associated score. When more than one string has the same score, the strings are ordered lexicographically. Some use cases for sorted sets include:

Leaderboards. For example, you can use sorted sets to easily maintain ordered lists of the highest scores in a massive online game.

Rate limiters. In particular, you can use a sorted set to build a sliding-window rate limiter to prevent excessive API requests.

Basic commands

ZADD adds a new member and associated score to a sorted set. If the member already exists, the score is updated.

ZRANGE returns members of a sorted set, sorted within a given range.

ZRANK returns the rank of the provided member, assuming the sorted is in ascending order.

ZREVRANK returns the rank of the provided member, assuming the sorted set is in descending order.

1. **Bitmaps**

Redis bitmaps are an extension of the string data type that lets you treat a string like a bit vector. Bitwise operations can be performed on strings. Since strings are binary safe blobs and their maximum length is 512 MB, they are suitable to set up to 2^32 different bits. Some use cases:

Efficient set representations for cases where the members of a set correspond to the integers 0-N.

Object permissions, where each bit represents a particular permission, similar to the way that file systems store permissions.

SETBIT/ GETBIT used to set or get the bit value (0/1).

BITOP performs bit-wise operations between different strings. The provided operations are AND, OR, XOR and NOT.

BITCOUNT performs population counting, reporting the number of bits set to 1.

BITPOS finds the first bit having the specified value of 0 or 1.

> setbit key 10 1

(integer) 1

> getbit key 10

(integer) 1

> getbit key 11

(integer) 0

1. **Hyperlog log**

HyperLogLog is a data structure that estimates the cardinality of a set. As a probabilistic data structure, HyperLogLog trades perfect accuracy for efficient space utilization.

The Redis HyperLogLog implementation uses up to 12 KB and provides a standard error of 0.81%.

**Basic commands**

PFADD adds an item to a HyperLogLog.

PFCOUNT returns an estimate of the number of items in the set.

PFMERGE combines two or more HyperLogLogs into one.

**Performance**

Writing (PFADD) to and reading from (PFCOUNT) the HyperLogLog is done in constant time and space. Merging HLLs is O(n), where n is the number of sketches.

**Limits**

The HyperLogLog can estimate the cardinality of sets with up to 18,446,744,073,709,551,616 (2^64) members.

1. **Streams**

A Redis stream is a data structure that acts like an append-only log. Can be record and simultaneously syndicate events in real time. Use cases:

Event sourcing (e.g., tracking user actions, clicks, etc.)

Sensor monitoring (e.g., readings from devices in the field)

Notifications (e.g., storing a record of each user's notifications in a separate stream)

Redis generates a unique ID for each stream entry. You can use these IDs to retrieve their associated entries later or to read and process all subsequent entries in the stream.

Redis streams support several trimming strategies (to prevent streams from growing unbounded) and more than one consumption strategy (see XREAD, XREADGROUP, and XRANGE).

**Basic commands**

XADD adds a new entry to a stream.

XREAD reads one or more entries, starting at a given position and moving forward in time.

XRANGE returns a range of entries between two supplied entry IDs.

XLEN returns the length of a stream.

Adding an entry to a stream is O(1). Accessing any single entry is O(n), where n is the length of the ID.

> XADD temperatures:us-ny:10007 \* temp\_f 87.2 pressure 29.69 humidity 46

"1658354918398-0"

> XADD temperatures:us-ny:10007 \* temp\_f 83.1 pressure 29.21 humidity 46.5

"1658354934941-0"

> XADD temperatures:us-ny:10007 \* temp\_f 81.9 pressure 28.37 humidity 43.7

"1658354957524-0"

XRANGE temperatures:us-ny:10007 1658354934941-0 + COUNT 2

1) 1) "1658354934941-0"

2) 1) "temp\_f"

2) "83.1"

3) "pressure"

4) "29.21"

5) "humidity"

6) "46.5"

2) 1) "1658354957524-0"

2) 1) "temp\_f"

2) "81.9"

3) "pressure"

4) "28.37"

5) "humidity"

6) "43.7"

1. **Transactions**

Redis Transactions allow the execution of a group of commands in a single step. All the commands in a transaction are serialized and executed sequentially. A request sent by another client will never be served in the middle of the execution of a Redis Transaction. This guarantees that the commands are executed as a single isolated operation.

**Commands:** MULTI, EXEC, DISCARD and WATCH.

> MULTI

OK

> INCR foo

QUEUED

> INCR bar

QUEUED

> EXEC

1) (integer) 1

2) (integer) 1

Redis does not support rollbacks of transactions since supporting rollbacks would have a significant impact on the simplicity and performance of Redis. DISCARD can be used in order to abort a transaction. In this case, no commands are executed and the state of the connection is restored to normal.

Optimistic locking using check-and-set

WATCH is used to provide a check-and-set (CAS) behavior to Redis transactions.

WATCHed keys are monitored in order to detect changes against them. If at least one watched key is modified before the EXEC command, the whole transaction aborts, and EXEC returns a Null reply to notify that the transaction failed.

WATCH mykey

val = GET mykey

val = val + 1

MULTI

SET mykey $val

EXEC

Using the above code, if there are race conditions and another client modifies the result of val in the time between our call to WATCH and our call to EXEC, the transaction will fail.

When EXEC is called, all keys are UNWATCHed, regardless of whether the transaction was aborted or not. Also when a client connection is closed, everything gets UNWATCHed.

**PUB/SUB**

SUBSCRIBE, UNSUBSCRIBE and PUBLISH implement the Publish/Subscribe messaging paradigm where senders (publishers) are not programmed to send their messages to specific receivers (subscribers). Rather, published messages are characterized into channels, without knowledge of what (if any) subscribers there may be. Subscribers express interest in one or more channels, and only receive messages that are of interest, without knowledge of what (if any) publishers there are.

To subscribe to channels foo and bar the client issues a SUBSCRIBE.

For instance in order to subscribe to channels foo and bar the client issues a SUBSCRIBE providing the names of the channels:

SUBSCRIBE foo bar

Messages sent by other clients to these channels will be pushed by Redis to all the subscribed clients.

A client subscribed to one or more channels should not issue commands, although it can subscribe and unsubscribe to and from other channels.

Pub/Sub has no relation to the key space. It was made to not interfere with it on any level, including database numbers.

A client may receive a single message multiple times if it's subscribed to multiple patterns matching a published message, or if it is subscribed to both patterns and channels matching the message. Like in the following example:

SUBSCRIBE foo

PSUBSCRIBE f\*

In the above example, if a message is sent to channel foo, the client will receive two messages: one of type message and one of type pmessage.

**COMMANDS**

SUBSCRIBE news

SUBSCRIBE news movies

PUBLISH news "News Message"

PUBLISH news "Latest News Message"

PUBLISH movie "New movie release"

**Pattern Based Subscription**

PSUBSCRIBE new\* (\* is wild card character matches for all words starting with new)

PSUBSCRIBE he?lo (? is for single character can be replaced with any character)

PSUBSCRIBE b[ia]ll ([...] any 1 character can take place bill/ball both will be subscribed)

PUBLISH newer "newer Message";

PUBLISH hewlo "hewlo Message";

**PUBSUB COMMANDS**

PUBSUB CHANNELS [pattern] -> returns list of all subscribed channels (Only Non-pattern-based subscriptions) If no pattern is specified otherwise if pattern is specified only channels matching the specified glob-style pattern are listed.

PUBSUB NUMSUB news -> return number of subscriptions for particular channel. (Only non-pattern-based subscriptions)

PUBSUB NUMPAT -> returns total number of unique patterns all the clients are subscribed to.

PUNSUBSCRIBE [pattern [pattern ...]] -> Unsubscribes the client from the given patterns, or from all of them if none is given.

UNSUBSCRIBE [channel [channel ...]] -> Unsubscribes the client from the given channels, or from all of them if none is given.

**Key Space Notifications**

Keyspace notifications allow clients to subscribe to Pub/Sub channels in order to receive events affecting the Redis data set in some way.

PUBLISH \_\_keyspace@0\_\_:mykey del

PUBLISH \_\_keyevent@0\_\_:del mykey

K Keyspace events, published with \_\_keyspace@<db>\_\_ prefix.

E Keyevent events, published with \_\_keyevent@<db>\_\_ prefix.

The first channel listens to all the events targeting the key mykey and the other channel listens only to del operation events on the key mykey.

**COMMANDS**

**CONFIG SET notify-keyspace-events KEA**

psubscribe \_\_key\*\_\_:\*

set mykey 20

expire mykey 10

**OUTPUT**

1) "psubscribe"

2) "\_\_key\*\_\_:\*"

3) (integer) 1

1) "pmessage"

2) "\_\_key\*\_\_:\*"

3) "\_\_keyspace@0\_\_:mykey"

4) "set"

1) "pmessage"

2) "\_\_key\*\_\_:\*"

3) "\_\_keyevent@0\_\_:set"

4) "mykey"

1) "pmessage"

2) "\_\_key\*\_\_:\*"

3) "\_\_keyspace@0\_\_:mykey"

4) "expire"

1) "pmessage"

2) "\_\_key\*\_\_:\*"

3) "\_\_keyevent@0\_\_:expire"

4) "mykey"

1) "pmessage"

2) "\_\_key\*\_\_:\*"

3) "\_\_keyspace@0\_\_:mykey"

4) "expired"

1) "pmessage"

2) "\_\_key\*\_\_:\*"

3) "\_\_keyevent@0\_\_:expired"

4) "mykey"

**Using Unix Time**

If you want a key to expire at a specific time, you can use the EXPIREAT command. This command takes a Unix timestamp as the duration.

The timeout can also be cleared, turning the key back into a persistent key, using the PERSIST command.

If a key is renamed with RENAME, the associated time to live is transferred to the new key name.