*Redis data types:*

1. *Redis Strings*: Redis strings store sequences of bytes, including text, serialized objects, and binary arrays. As such, strings are the most basic Redis data type. They're often used for caching, but they support additional functionality that lets you implement counters and perform bitwise operations, too.

***GET*** - O(1) - Get the value of key. If the key does not exist the special value nil is returned. An error is returned if the value stored at key is not a string, because GET only handles string values. GET mykey

***GETDEL* -** O(1) - Get the value of key and delete the key. This command is similar to GET, except for the fact that it also deletes the key on success (if and only if the key's value type is a string). *GETDEL mykey*

***GETEX*** - O(1) - Get the value of key and optionally set its expiration. GETEX is similar to GET, but is a write command with additional options. *GETEX mykey EX 60*

* *EX* *seconds* -- Set the specified expire time, in seconds.
* *PX* *milliseconds* -- Set the specified expire time, in milliseconds.
* *EXAT* *timestamp-seconds* -- Set the specified Unix time at which the key will expire, in seconds.
* *PXAT* *timestamp-milliseconds* -- Set the specified Unix time at which the key will expire, in milliseconds.
* *PERSIST* -- Remove the time to live associated with the key.

***GETRANGE* -** O(N) - Returns the substring of the string value stored at key, determined by the offsets start and end (both are inclusive). Negative offsets can be used in order to provide an offset starting from the end of the string. So -1 means the last character, -2 the penultimate and so forth.

The function handles out of range requests by limiting the resulting range to the actual length of the string.

SET mykey "This is a string" //"OK"

GETRANGE mykey 0 3 // "This"

GETRANGE mykey -3 -1 //"ing"

GETRANGE mykey 0 -1 //"This is a string"

GETRANGE mykey 10 100 //"string"

***INCR* -** O(1) - Increments the number stored at key by one. If the key does not exist, it is set to 0 before performing the operation. An error is returned if the key contains a value of the wrong type or contains a string that can not be represented as integer.

SET mykey "10" //"OK"

INCR mykey // 11

GET mykey //"11"

***INCRBY*** - O(1) - INCRBY mykey 5 // “16”

***INCRBYFLOAT*** - O(1) –

SET mykey 10.50 //"OK"

INCRBYFLOAT mykey 0.1 //"10.6"

INCRBYFLOAT mykey -5 //"5.6"

SET mykey 5.0e3 //"OK"

INCRBYFLOAT mykey 2.0e2 //"5200"

***DECR*** - O(1) - DECR mykey

***DECRBY*** - O(1) - DECRBY mykey 3

***LCS* -** O(N\*M) - The LCS command implements the longest common subsequence algorithm. Note that this is different than the longest common string algorithm, since matching characters in the string does not need to be contiguous.

MSET key1 ohmytext key2 mynewtext //OK

LCS key1 key2 //"mytext"

LCS key1 key2 LEN // 6

***MGET* -** O(N) – MGET key1 key2 nonexisting // ["Hello" , "World" , nill]

***MSET*** - O(N) - MSET key1 "Hello" key2 "World" // OK

***MSETNX*** - O(N) - Sets the given keys to their respective values. MSETNX will not perform any operation at all even if just a single key already exists.

Returns, 1 if the all the keys were set, 0 if no key was set (at least one key already existed).

MSETNX key1 "Hello" key2 "there" // 1

MSETNX key2 "new" key3 "world" //0

MGET key1 key2 key3 // ["Hello" , "there" , nill]

***SET*** – O(1) –

SET mykey "Hello" //"OK"

SET anotherkey "will expire in a minute" EX 60 //"OK"

***SETRANGE*** – O(1) – Overwrites part of the string stored at *key*, starting at the specified offset, for the entire length of *value*. If the offset is larger than the current length of the string at *key*, the string is padded with zero-bytes(spaces) to make *offset* fit. Non-existing keys are considered as empty strings, so this command will make sure it holds a string large enough to be able to set *value* at *offset*.

SET key1 "Hello World" //"OK"

SETRANGE key1 6 "Redis" //11

GET key1 //"Hello Redis"

***STRLEN*** – O(1) – Returns the length of the string value stored at key. An error is returned when key holds a non-string value. Returns 0 when key does not exist. STRLEN key1

***APPEND*** – O(1) – If key already exists and is a string, this command appends the value at the end of the string. If key does not exist it is created and set as an empty string and then the value is appended. APPEND mykey " World"

1. *Redis Lists*: Redis lists are linked lists of string values. Redis lists are frequently used to, Implement stacks and queues, Build queue management for background worker systems.

***LPUSH*** – O(1) - Insert all the specified values at the head of the list stored at key. If key does not exist, it is created as empty list before performing the push operations. When key holds a value that is not a list, an error is returned.

LPUSH mylist "world" "hello" //2

LRANGE mylist 0 -1 ["hello", "world"]

***LPUSHX* -** O(1) - Inserts specified values at the head of the list stored at key, only if key already exists and holds a list. In contrary to LPUSH, no operation will be performed when key does not yet exist. LPUSHX mylist "Hello"

***LRANGE*** - O(N) - Returns the specified elements of the list stored at key. The offsets start and stop are zero-based indexes, with 0 being the first element of the list (the head of the list), 1 being the next element and so on. LRANGE mylist 0 -1

***LREM*** - O(N) - Removes the first count occurrences of elements equal to element from the list stored at key. The count argument influences the operation in the following ways:

* count > 0: Remove elements equal to element moving from head to tail.
* count < 0: Remove elements equal to element moving from tail to head.
* count = 0: Remove all elements equal to element.

For example, LREM list -2 "hello" will remove the last two occurrences of "hello" in the list stored at list.

***LSET*** - O(N) - Sets the list element at index to element. An error is returned for out of range indexes.

***LTRIM*** - O(N) - Trim an existing list so that it will contain only the specified range of elements specified. Both start and stop are zero-based indexes, where 0 is the first element of the list (the head), 1 the next element and so on. LTRIM mylist 0 99

***LPOS*** - O(N) - The command returns the index of matching elements inside a Redis list. By default, when no options are given, it will scan the list from head to tail, looking for the first match of "element". If the element is found, its index (the zero-based position in the list) is returned. Otherwise, if no match is found, nil is returned. LPOS mylist c RANK 2

That is, the second occurrence of "c" is at position 6. A negative "rank" as the RANK argument tells LPOS to invert the search direction, starting from the tail to the head. So, we want to say, give me the first element starting from the tail of the list,

LPOS mylist c RANK -1

Sometimes we want to return not just the Nth matching element, but the position of all the first N matching elements. This can be achieved using the COUNT option.

LPOS mylist c COUNT 2

We can combine COUNT and RANK, so that COUNT will try to return up to the specified number of matches, but starting from the Nth match, as specified by the RANK option.

LPOS mylist c RANK -1 COUNT 2

LPOS mylist c COUNT 0 // give me all matches

Finally, the MAXLEN option tells the command to compare the provided element only with a given maximum number of list items. So for instance specifying MAXLEN 1000 will make sure that the command performs only 1000 comparisons, effectively running the algorithm on a subset of the list.

***LPOP*** – O(1) - Removes and returns the first elements of the list stored at key.

By default, the command pops a single element from the beginning of the list. When provided with the optional count argument, the reply will consist of up to count elements, depending on the list's length. LPOP mylist 2

***LMOVE*** – O(1) - Atomically returns and removes the first/last element (head/tail depending on the wherefrom argument) of the list stored at source, and pushes the element at the first/last element (head/tail depending on the whereto argument) of the list stored at destination.

For example: consider source holding the list a,b,c, and destination holding the list x,y,z. Executing LMOVE source destination RIGHT LEFT results in source holding a,b and destination holding c,x,y,z.

If source does not exist, the value nil is returned and no operation is performed. If source and destination are the same, the operation is equivalent to removing the first/last element from the list and pushing it as first/last element of the list, so it can be considered as a list rotation command (or a no-op if wherefrom is the same as whereto).

LMOVE mylist myotherlist RIGHT LEFT

***LLEN*** – O(1) - Returns the length of the list stored at key. If key does not exist, it is interpreted as an empty list and 0 is returned. An error is returned when the value stored at key is not a list. LLEN mylist

***LINSERT*** - O(N) - Inserts element in the list stored at key either before or after the reference value pivot.

When key does not exist, it is considered an empty list and no operation is performed.

An error is returned when key exists but does not hold a list value.

LINSERT mylist BEFORE "World" "There"

LRANGE mylist 0 -1 // ["Hello", "There", "World"]

***LINDEX*** - O(N) - Returns the element at index index in the list stored at key. The index is zero-based, so 0 means the first element, 1 the second element and so on. Negative indices can be used to designate elements starting at the tail of the list. Here, -1 means the last element, -2 means the penultimate and so forth.

LINDEX mylist 0 // "Hello"

LINDEX mylist -1 //"World"

***RPOP*** – O(1) - Removes and returns the last elements of the list stored at key.

RPOP mylist

RPOP mylist 2

***RPUSH*** – O(1) - Insert all the specified values at the tail of the list stored at key. If key does not exist, it is created as empty list before performing the push operation. When key holds a value that is not a list, an error is returned.

RPUSH mylist "hello" RPUSH mylist "there" "world"

**RPUSHX** – O(1) - Inserts specified values at the tail of the list stored at key, only if key already exists and holds a list. In contrary to RPUSH, no operation will be performed when key does not yet exist. RPUSHX mylist "World"

1. *Redis Sets*: A Redis set is an unordered collection of unique strings (members). You can use Redis sets to efficiently:Track unique items (e.g., track all unique IP addresses accessing a given blog post). Represent relations (e.g., the set of all users with a given role). Perform common set operations such as intersection, unions, and differences.

***SADD*** – O(1) - Add the specified members to the set stored at key. Specified members that are already a member of this set are ignored. If key does not exist, a new set is created before adding the specified members. SADD myset "1" "2" "3"

***SCARD*** – O(1) - Returns the set cardinality (number of elements) of the set stored at key. SCARD myset

***SISMEMBER*** – O(1) - Returns if member is a member of the set stored at key. SISMEMBER myset "one"

***SMEMBERS*** – O(N) - Returns all the members of the set value stored at key. SMEMBERS myset

***SMISMEMBER*** – O(N) - Returns whether each member is a member of the set stored at key. SMISMEMBER myset "one" "notamember" // [ 1 , 0 ]

***SMOVE*** – O(1) - Move member from the set at source to the set at destination. SMOVE myset myotherset "two"

***SPOP*** – O(1) - Removes and returns one or more random members from the set value store at key. SPOP myset 3

***SRANDMEMBER*** – O(1) - When called with just the key argument, return a random element from the set value stored at key.

If the provided count argument is positive, return an array of **distinct elements**. The array's length is either count or the set's size, whichever is lower.

If called with a negative count, the behavior changes and the command is allowed to return the **same element multiple times**. In this case, the number of returned elements is the absolute value of the specified count. SRANDMEMBER myset 2

***SREM*** – O(1) - Remove the specified members from the set stored at key. Specified members that are not a member of this set are ignored. If key does not exist, it is treated as an empty set and this command returns 0. SREM myset "four"

***SUNION*** – O(N) - Returns the members of the set resulting from the union of all the given sets.

key1 = {a,b,c,d}

key2 = {c}

key3 = {a,c,e}

SUNION key1 key2 key3 = {a,b,c,d,e}

***SUNIONSTORE*** – O(N) - This command is equal to SUNION, but instead of returning the resulting set, it is stored in destination.

If destination already exists, it is overwritten.

SUNIONSTORE key key1 key2

***SDIFF*** – O(N) - Returns the members of the set resulting from the difference between the first set and all the successive sets.

key1 = {a,b,c,d}

key2 = {c}

key3 = {a,c,e}

SDIFF key1 key2 key3 = {b,d}

***SDIFFSTORE*** – O(N) - This command is equal to SDIFF, but instead of returning the resulting set, it is stored in destination.

If destination already exists, it is overwritten.

SDIFFSTORE key key1 key2

***SINTER*** – O(N) - Returns the members of the set resulting from the intersection of all the given sets.

key1 = {a,b,c,d}

key2 = {c}

key3 = {a,c,e}

SINTER key1 key2 key3 = {c}

***SINTERCARD*** – O(N) -This command is similar to SINTER, but instead of returning the result set, it returns just the cardinality of the result. Returns the cardinality of the set which would result from the intersection of all the given sets.

SINTERCARD numkeys key [key ...] [LIMIT limit]

SINTERCARD 2 key1 key2 LIMIT 1

***SINTERSTORE*** – O(N) - This command is equal to SINTER, but instead of returning the resulting set, it is stored in destination.

If destination already exists, it is overwritten.

1. *Redis 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.

***HGET*** – O(1) - Returns the value associated with field in the hash stored at key. HGET myhash field1

***HEXISTS*** – O(1) - Returns if field is an existing field in the hash stored at key. HEXISTS myhash field1

***HGETALL*** – O(N) - Returns all fields and values of the hash stored at key. In the returned value, every field name is followed by its value, so the length of the reply is twice the size of the hash. HGETALL myhash

***HINCRBY*** – O(1) - Increments the number stored at field in the hash stored at key by increment. If key does not exist, a new key holding a hash is created. If field does not exist the value is set to 0 before the operation is performed. HINCRBY myhash field -1

***HINCRBYFLOAT*** – O(1) - HINCRBYFLOAT mykey field 0.1

***HKEYS*** – O(N) - Returns all field names in the hash stored at key. HKEYS myhash

***HLEN*** – O(1) - Returns the number of fields contained in the hash stored at key. HLEN myhash

***HMGET*** – O(N) - Returns the values associated with the specified fields in the hash stored at key.

HMGET myhash field1 field2 nofield // ["Hello", "World", nil]

***HRANDFIELD*** – O(N) - When called with just the key argument, return a random field from the hash value stored at key.

If the provided count argument is positive, return an array of **distinct fields**. The array's length is either count or the hash's number of fields HLEN, whichever is lower.

If called with a negative count, the behavior changes and the command is allowed to return the **same field multiple times**. In this case, the number of returned fields is the absolute value of the specified count.

The optional WITHVALUES modifier changes the reply so it includes the respective values of the randomly selected hash fields.

HRANDFIELD coin -5 WITHVALUES

***HSET*** – O(1) - Sets the specified fields to their respective values in the hash stored at key.

This command overwrites the values of specified fields that exist in the hash. If key doesn't exist, a new key holding a hash is created.

HSET myhash field2 "Hi" field3 "World"

***HSETNX*** – O(1) - Sets field in the hash stored at key to value, only if field does not yet exist. If key does not exist, a new key holding a hash is created. If field already exists, this operation has no effect. HSETNX myhash field "World"

***HSTRLEN*** – O(1) - Returns the string length of the value associated with field in the hash stored at key. If the key or the field do not exist, 0 is returned. HSTRLEN myhash f2

***HVALS*** – O(N) - Returns all values in the hash stored at key.

HVALS myhash

***HDEL*** – O(1) - Removes the specified fields from the hash stored at key. Specified fields that do not exist within this hash are ignored. If key does not exist, it is treated as an empty hash and this command returns 0. HDEL myhash field1

1. *Redis 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.

***ZADD*** - O(log(N)) - Adds all the specified members with the specified scores to the sorted set stored at key. It is possible to specify multiple score / member pairs. If a specified member is already a member of the sorted set, the score is updated and the element reinserted at the right position to ensure the correct ordering.

ZADD supports a list of options, specified after the name of the key and before the first score argument. Options are:

* **XX**: Only update elements that already exist. Don't add new elements.
* **NX**: Only add new elements. Don't update already existing elements.
* **LT**: Only update existing elements if the new score is **less than** the current score. This flag doesn't prevent adding new elements.
* **GT**: Only update existing elements if the new score is **greater than** the current score. This flag doesn't prevent adding new elements.

ZADD myzset 1 "one" //1

ZADD myzset 1 "uno" // 1

ZADD myzset 2 "two" 3 "three" // 2

ZRANGE myzset 0 -1 WITHSCORES

//[["one","1"],["uno","1"],["two","2"], ["three","3"]]

***ZCARD*** - O(1) - Returns the sorted set cardinality (number of elements) of the sorted set stored at key. ZCARD myzset

***ZCOUNT*** - O(log(N)) - Returns the number of elements in the sorted set at key with a score between min and max. ZCOUNT myzset 1 3

***ZDIFF*** - O(N) - Computes the difference between the first and all successive input sorted sets. ZDIFF numkeys key [key ...] [WITHSCORES]

ZDIFF 2 zset1 zset2 WITHSCORES

***ZDIFFSTORE*** - O(N) - Computes the difference between the first and all successive input sorted sets and stores the result in destination. The total number of input keys is specified by numkeys. If destination already exists, it is overwritten.

ZDIFFSTORE out 2 zset1 zset2

***ZINCRBY*** - O(log(N)) - Increments the score of member in the sorted set stored at key by increment. If member does not exist in the sorted set, it is added with increment as its score (as if its previous score was 0.0). If key does not exist, a new sorted set with the specified member as its sole member is created.

ZINCRBY myzset 2 "one"

**ZINTER** - O(N) - ZINTER 2 zset1 zset2 WITHSCORES

**ZINTERCARD** - O(N) - ZINTERCARD 2 zset1 zset2 LIMIT 1

**ZLEXCOUNT** - O(log(N)) - When all the elements in a sorted set are inserted with the same score, in order to force lexicographical ordering, this command returns the number of elements in the sorted set at key with a value between min and max.

Valid *start* and *stop* must start with ( or [, in order to specify if the range item is respectively exclusive or inclusive. The special values of + or - for *start* and *stop* have the special meaning or positively infinite and negatively infinite strings, so for instance the command **ZRANGEBYLEX myzset - +** is guaranteed to return all the elements in the sorted set, if all the elements have the same score.

ZADD myzset 0 a 0 b 0 c 0 d 0 e

ZADD myzset 0 f 0 g

ZLEXCOUNT myzset - + // 7

ZLEXCOUNT myzset [b [f // 5

**ZMPOP** - O(N) - Pops one or more elements, that are member-score pairs, from the first non-empty sorted set in the provided list of key names. When the MIN modifier is used, the elements popped are those with the lowest scores from the first non-empty sorted set. The MAX modifier causes elements with the highest scores to be popped. The optional COUNT can be used to specify the number of elements to pop, and is set to 1 by default.

The number of popped elements is the minimum from the sorted set's cardinality and COUNT's value.

ZADD myzset 1 "one" 2 "two" 3 "three"

ZMPOP 1 myzset MIN

1) "myzset"

2) 1) 1) "one"

2) "1"

ZRANGE myzset 0 -1 WITHSCORES

1) "two"

2) "2"

3) "three"

4) "3"

ZMPOP 1 myzset MAX COUNT 10

1) "myzset"

2) 1) 1) "three"

2) "3"

2) 1) "two"

2) "2"

ZADD myzset2 4 "four" 5 "five" 6 "six"

ZMPOP 2 myzset myzset2 MIN COUNT 10

1) "myzset2"

2) 1) 1) "four"

2) "4"

2) 1) "five"

2) "5"

3) 1) "six"

2) "6"

**ZMSCORE** - O(N) - Returns the scores associated with the specified members in the sorted set stored at key.

For every member that does not exist in the sorted set, a nil value is returned.

ZADD myzset 1 "one"

(integer) 1

ZADD myzset 2 "two"

(integer) 1

ZMSCORE myzset "one" "two" "nofield"

1) "1"

2) "2"

3) (nil)

**ZPOPMAX** - O(log(N)) - Removes and returns up to count members with the highest scores in the sorted set stored at key.

When left unspecified, the default value for count is 1. Specifying a count value that is higher than the sorted set's cardinality will not produce an error. When returning multiple elements, the one with the highest score will be the first, followed by the elements with lower scores.

ZPOPMAX myzset

ZPOPMIN- O(log(N)) - ZPOPMIN myzset

**ZRANDMEMBER** - O(N) - When called with just the key argument, return a random element from the sorted set value stored at key.

If the provided count argument is positive, return an array of **distinct elements**. The array's length is either count or the sorted set's cardinality ZCARD, whichever is lower.

If called with a negative count, the behavior changes and the command is allowed to return the **same element multiple times**. In this case, the number of returned elements is the absolute value of the specified count.

The optional WITHSCORES modifier changes the reply so it includes the respective scores of the randomly selected elements from the sorted set.

ZRANDMEMBER dadi -5 WITHSCORES

**ZRANGE** - O(log(N)) -

ZADD myzset 1 "one" 2 "two" 3 "three"

(integer) 3

ZRANGE myzset 0 -1

1) "one"

2) "two"

3) "three"

ZRANGE myzset 0 1 WITHSCORES

1) "one"

2) "1"

3) "two"

4) "2"

**ZRANGESTORE** - O(log(N)) - This command is like ZRANGE, but stores the result in the <dst> destination key.

ZRANGESTORE dstzset srczset 2 -1

**ZRANK** - O(log(N)) - Returns the rank of member in the sorted set stored at key, with the scores ordered from low to high. The rank (or index) is 0-based, which means that the member with the lowest score has rank 0.

ZADD myzset 1 "one"

(integer) 1

ZADD myzset 2 "two"

(integer) 1

ZADD myzset 3 "three"

(integer) 1

ZRANK myzset "three"

(integer) 2

ZRANK myzset "four"

(nil)

**ZREM** - O(log(N)) - ZREM myzset "two"

**ZREMRANGEBYRANK** - O(log(N)) - Removes all elements in the sorted set stored at key with rank between start and stop. Both start and stop are 0 -based indexes with 0 being the element with the lowest score.

ZADD myzset 1 "one" 2 "two" 3 "three"

ZREMRANGEBYRANK myzset 0 1

ZRANGE myzset 0 -1 WITHSCORES

1) "three"

2) "3"

**ZREMRANGEBYSCORE** - O(log(N)) - Removes all elements in the sorted set stored at key with a score between min and max (inclusive).

ZREMRANGEBYSCORE myzset -inf 2

**ZREVRANK** - O(N) - Returns the rank of member in the sorted set stored at key, with the scores ordered from high to low. The rank (or index) is 0-based, which means that the member with the highest score has rank 0.

ZADD myzset 1 "one" 2 "two" 3 "three"

ZREVRANK myzset "one" // 2

ZREVRANK myzset "four" //nil

**ZSCORE** - O(1) - Returns the score of member in the sorted set at key. ZSCORE myzset "one"

**ZUNION** - O(N) -

ZADD myzset1 1 "one" 2 "two" 3 "three"

ZADD myzset2 1 "one" 2 "two"

ZUNION 2 myzset1 myzset2

1) "one"

2) "three"

3) "two"

ZUNION 2 zset1 zset2 WITHSCORES

***Redis Generic Commands:***

***COPY*:** O(N) for collections and O(1) for strings

SET dolly "sheep"

COPY dolly clone

***DEL*:** O(N) for collections and O(1) for strings

DEL key1 key2 key3

***EXISTS*:** O(N) where N is the number of keys to check

EXISTS key1 //1

***EXPIRE*:** O(1)

The EXPIRE command supports a set of options:

* NX -- Set expiry only when the key has no expiry
* XX -- Set expiry only when the key has an existing expiry
* GT -- Set expiry only when the new expiry is greater than current one
* LT -- Set expiry only when the new expiry is less than current one

EXPIRE mykey 10 NX

***EXPIREAT*:** O(1)

It takes an absolute Unix Timestamp (seconds since January 1, 1970). A timestamp in the past will delete the key immediately**.**

EXPIREAT mykey 1293840000

***EXPIRETIME*:** O(1)

Returns the absolute Unix timestamp (since January 1, 1970) in seconds at which the given key will expire.

EXPIRETIME mykey //1293840000

***KEYS:*** O(N)

Supported glob-style patterns:

* h?llo matches hello, hallo and hxllo
* h\*llo matches hllo and heeeello
* h[ae]llo matches hello and hallo, but not hillo
* h[^e]llo matches hallo, hbllo, ... but not hello
* h[a-b]llo matches hallo and hbllo

MSET firstname Jack lastname Stuntman age 35

KEYS \*name\*

1) "lastname" 2) "firstname"

KEYS a??

1. "age" redis> KEYS \* 1) "lastname" 2) "firstname" 3) "age"

***PERSIST***:O(1)

Remove the existing timeout on key, turning the key from *volatile* (a key with an expire set) to *persistent* (a key that will never expire as no timeout is associated).

PERSIST mykey

***RANDOMKEY*:** O(1)

Return a random key from the currently selected database.

***RENAME*:** O(1)

RENAME mykey myotherkey

***RENAMENX*:** O(1)

Renames key to newkey if newkey does not yet exist. It returns an error when key does not exist.

***SORT*:**O(N)

Returns or stores the elements contained in the list, set or sorted set at key.

By default, sorting is numeric and elements are compared by their value interpreted as double precision floating point number. This is SORT in its simplest form: SORT mylist

In order to sort the numbers from large to small, use the DESC modifier: SORT mylist DESC

When mylist contains string values and you want to sort them lexicographically, use the ALPHA modifier: SORT mylist ALPHA

The number of returned elements can be limited using the LIMIT modifier. This modifier takes the offset argument, specifying the number of elements to skip and the count argument, specifying the number of elements to return from starting at offset. The following example will return 10 elements of the sorted version of mylist, starting at element 0 (offset is zero-based): SORT mylist LIMIT 0 10

Almost all modifiers can be used together. The following example will return the first 5 elements, lexicographically sorted in descending order: SORT mylist LIMIT 0 5 ALPHA DESC

By default, SORT returns the sorted elements to the client. With the STORE option, the result will be stored as a list at the specified key instead of being returned to the client.

SORT mylist STORE resultkey

***SORT\_RO*:** O(N)

Read-only variant of the SORT command. It is exactly like the original SORT but refuses the STORE option and can safely be used in read-only replicas.

***TTL*:** O(1)

Returns the remaining time to live of a key that has a timeout.

* The command returns -2 if the key does not exist.
* The command returns -1 if the key exists but has no associated expire.

TTL mykey

***TYPE*:** O(1)

Returns the string representation of the type of the value stored at key. The different types that can be returned are: string, list, set, zset, hash and stream. TYPE key1

***UNLINK*:** O(1)

This command is very similar to DEL: it removes the specified keys. Just like DEL a key is ignored if it does not exist. However the command performs the actual memory reclaiming in a different thread, so it is not blocking, while DEL is. This is where the command name comes from: the command just unlinks the keys from the keyspace. The actual removal will happen later asynchronously. UNLINK key1 key2 key3

***SCAN*:** O(1)

The SCAN command and the closely related commands SSCAN, HSCAN and ZSCAN are used in order to incrementally iterate over a collection of elements.

* SCAN iterates the set of keys in the currently selected Redis database.
* SSCAN iterates elements of Sets types.
* HSCAN iterates fields of Hash types and their associated values.
* ZSCAN iterates elements of Sorted Set types and their associated scores.

Since these commands allow for incremental iteration, returning only a small number of elements per call, they can be used in production without the downside of commands like KEYS or SMEMBERS that may block the server for a long time (even several seconds) when called against big collections of keys or elements.

SCAN is a cursor based iterator. This means that at every call of the command, the server returns an updated cursor that the user needs to use as the cursor argument in the next call.

An iteration starts when the cursor is set to 0, and terminates when the cursor returned by the server is 0. The following is an example of SCAN iteration:

scan 0

1) "17"

2) 1) "key:12"

2) "key:8"

3) "key:4"

4) "key:14"

5) "key:16"

6) "key:17"

7) "key:15"

8) "key:10"

9) "key:3"

10) "key:7"

11) "key:1"

scan 17

1) "0"

2) 1) "key:5"

2) "key:18"

3) "key:0"

4) "key:2"

5) "key:19"

6) "key:13"

7) "key:6"

8) "key:9"

9) "key:11"

In the example above, the first call uses zero as a cursor, to start the iteration. The second call uses the cursor returned by the previous call as the first element of the reply, that is, 17.

As you can see the **SCAN return value** is an array of two values: the first value is the new cursor to use in the next call, the second value is an array of elements.

Since in the second call the returned cursor is 0, the server signaled to the caller that the iteration finished, and the collection was completely explored. Starting an iteration with a cursor value of 0, and calling SCAN until the returned cursor is 0 again is called a **full iteration**.

**Scan guarantees**

The SCAN command, and the other commands in the SCAN family, are able to provide to the user a set of guarantees associated to full iterations.

* A full iteration always retrieves all the elements that were present in the collection from the start to the end of a full iteration. This means that if a given element is inside the collection when an iteration is started, and is still there when an iteration terminates, then at some point SCAN returned it to the user.
* A full iteration never returns any element that was NOT present in the collection from the start to the end of a full iteration. So if an element was removed before the start of an iteration, and is never added back to the collection for all the time an iteration lasts, SCAN ensures that this element will never be returned.

However because SCAN has very little state associated (just the cursor) it has the following drawbacks:

* A given element may be returned multiple times. It is up to the application to handle the case of duplicated elements, for example only using the returned elements in order to perform operations that are safe when re-applied multiple times.
* Elements that were not constantly present in the collection during a full iteration, may be returned or not: it is undefined.

***Memcached***

To connect to memcached server with telnet and start a session:

telnet localhost 11111

1. *SET*:

Memcached **set** command is used to set a new value to a new or existing key.

set KEY META\_DATA EXPIRY\_TIME LENGTH\_IN\_BYTES [noreply]

VALUE

The keywords in the syntax are as described below −

* ***KEY*** − It is the name of the key by which data is stored and retrieved from Memcached.
* ***META\_DATA*** − It is the 32-bit unsigned integer that the server stores with the data provided by the user, and returns along with the data when the item is retrieved.
* ***EXPIRY\_TIME*** − It is the expiration time in seconds. 0 means no delay. If exptime is more than 30 days, Memcached uses it as UNIX timestamp for expiration.
* ***LENGTH\_IN\_BYTES*** − It is the number of bytes in the data block that needs to be stored. This is the length of the data that needs to be stored in Memcached.
* ***noreply* (optional)** - It is a parameter that informs the server not to send any reply.
* ***value*** − It is the data that needs to be stored. The data needs to be passed on the new line after executing the command with the above options.

***Output:***

* **STORED** indicates success.
* **ERROR** indicates incorrect syntax or error while saving data.

1. ***ADD***: Memcached **add** command is used to set a value to a new key. If the key already exists, then it gives the output NOT\_STORED.

add KEY META\_DATA EXPIRY\_TIME LENGTH\_IN\_BYTES [noreply]

VALUE

OUTPUTS can be either STORED or NOT\_STORED

1. ***REPLACE***: Memcached **replace** command is used to replace the value of an existing key. If the key does not exist, then it gives the output NOT\_STORED.

replace KEY META\_DATA EXPIRY\_TIME LENGTH\_IN\_BYTES [noreply]

VALUE

1. ***APPEND*:** Memcached append command is used to add some data in an existing key. The data is stored after the existing data of the key.

append KEY META\_DATA EXPIRY\_TIME LENGTH\_IN\_BYTES [noreply]

VALUE

OUTPUTS can be either STORED or NOT\_STORED

1. ***PREPEND*:** Memcached prepend command is used to add some data in an existing key. The data is stored before the existing data of the key.

prepend KEY META\_DATA EXPIRY\_TIME LENGTH\_IN\_BYTES [noreply]

VALUE

OUTPUTS can be either STORED or NOT\_STORED

1. **GET:** Memcached get command is used to get the value stored at key. If the key does not exist in Memcached, then it returns nothing.

get KEY

1. ***DELETE***: delete key [noreply]

OUTPUTS can be DELETED, ERROR or NOT\_FOUND

1. ***INCR/DECR***: Memcached **incr** and **decr** commands are used to increment or decrement the numeric value of an existing key. If the key is not found, then it returns **NOT\_FOUND**. If the key is not numeric, then it returns **CLIENT\_ERROR cannot increment or decrement non-numeric value**. Otherwise, **ERROR** is returned.

incr KEY INC\_VALUE

decr KEY DEC\_VALUE

1. ***STATS***: Memcached **stats** command is used to return server statistics such as PID, version, connections, etc.

**stats**

ye 'ole basic stats command.

**stats items**

Returns some information, broken down by slab, about items stored in memcached.

**stats slabs**

Returns more information, broken down by slab, about items stored in memcached. More centered to performance of a slab rather than counts of particular items.

1. ***flush\_all*:** Memcached flush\_all command is used to delete all data (key-value pairs) from the Memcached server. It accepts an optional parameter called time that sets a time after which the Memcached data is to be cleared.

flush\_all [time] [noreply]