Redis Data Types

# Strings

Redis strings store **sequences of bytes**, including**:**

* Text
* serialized objects
* and binary arrays

As such, strings are the simplest type of value you can associate with a Redis key.

They're:

* **often used for caching** (like caching HTML fragments or pages)
* **but they support additional functionality that lets you implement counters**
* **and perform bitwise operations, too.**

Since Redis keys are strings, when we use the string type as a value too, we are mapping a string to another string.

 using the [SET](https://redis.io/docs/latest/commands/set/) and the [GET](https://redis.io/docs/latest/commands/get/) commands are the way we set and retrieve a string value. Note that [SET](https://redis.io/docs/latest/commands/set/) will replace any existing value already stored into the key, in the case that the key already exists, **even if the key is associated with a non-string value**.

> set bike:1 Deimos

OK

> get bike:1

"Deimos"

Values can be strings (**including binary data**) **of every kind**, for instance **you can store a jpeg image inside a value. A value can't be bigger than 512 MB.**

## set or retrieve the value of multiple keys in a single command for reduced latency

> mset bike:1 "Deimos" bike:2 "Ares" bike:3 "Vanth"

OK

> mget bike:1 bike:2 bike:3

1) "Deimos"

2) "Ares"

3) "Vanth"

## Strings as Counters

Even if strings are the basic values of Redis, there are interesting operations you can perform with them. For instance, one is **atomic increment:**

> set total\_crashes 0

OK

> incr total\_crashes

(integer) 1

> incrby total\_crashes 10

(integer) 11

The [INCR](https://redis.io/docs/latest/commands/incr/) command parses the string value as an integer, increments it by one, and finally sets the obtained value as the new value.

What does it mean that INCR is atomic?

**That even multiple clients issuing INCR against the same key will never enter into a race condition.** For instance, it will never happen that client 1 reads "10", client 2 reads "10" at the same time, both increment to 11, and set the new value to 11. The final value will always be 12 and **the read-increment-set operation is performed while all the other clients are not executing a command at the same time.**

**Practical point:**

if you have a system that increments a Redis key using [INCR](https://redis.io/docs/latest/commands/incr/) every time your web site receives a new visitor. You may want to collect this information once every hour, without losing a single increment. You can [GETSET](https://redis.io/docs/latest/commands/getset/) the key, assigning it the new value of "0" and reading the old value back.(This GETSET command has apparently been deprecated, use:

set key value get )

## Implementing Simple Locks with Redis Strings

SETNX sets the value of a key if and only if the key does not already exist. This property makes it ideal for implementing locks, as a successful SETNX operation indicates that the lock has been acquired.

More details:

<https://redis.io/glossary/redis-lock/>

<https://redis.io/docs/latest/develop/clients/patterns/distributed-locks/>

## Bitwise Operations

To perform bitwise operations on a string, see the [bitmaps data type](https://redis.io/docs/latest/develop/data-types/bitmaps/) section.

## Alternatives

If you're storing structured data as a serialized string, you may also want to consider Redis [hashes](https://redis.io/docs/latest/develop/data-types/hashes/) or [JSON](https://redis.io/docs/latest/develop/data-types/json/).

## Performance

Most string operations are O(1). However, be careful with the [SUBSTR](https://redis.io/docs/latest/commands/substr/), [GETRANGE](https://redis.io/docs/latest/commands/getrange/), and [SETRANGE](https://redis.io/docs/latest/commands/setrange/) commands, which can be O(n). These random-access string commands may cause performance issues when dealing with large strings.

## Limits

By default, a single Redis string can be a maximum of 512 MB.

# Hashes

Redis hashes are record types structured as **collections of field-value pairs**.

You can use hashes to:

* represent basic objects*(what is a basic object??)*
* and to store groupings of counters, among other things.

> HSET bike:1 model Deimos brand Ergonom type 'Enduro bikes' price 4972

(integer) 4

> HGET bike:1 model

"Deimos"

> HGET bike:1 price

"4972"

> HGETALL bike:1

1) "model"

2) "Deimos"

3) "brand"

4) "Ergonom"

5) "type"

6) "Enduro bikes"

7) "price"

8) "4972"

*Use HMGET to retrieve multiple fields.*

There are commands that are able to perform operations on individual fields as well, like [HINCRBY](https://redis.io/docs/latest/commands/hincrby/):

> HINCRBY bike:1 price 100

(integer) 5072

> HINCRBY bike:1 price -100

(integer) 4972

**It is worth noting that small hashes (i.e., a few elements with small values) are encoded in special way in memory that make them very memory efficient.**

## Field Expiration

New in Redis Open Source 7.4 is the ability to specify an **expiration time or a time-to-live** **(TTL) value for individual hash fields.** This capability is comparable to [key expiration](https://redis.io/docs/latest/develop/using-commands/keyspace/#key-expiration) and includes a number of similar commands.

*Support for hash field expiration in the official client libraries is not yet available, but you can test hash field expiration now with beta versions of the*[*Python (redis-py)*](https://github.com/redis/redis-py)*and*[*Java (Jedis)*](https://github.com/redis/jedis)*client libraries.*

* Use the following commands to set either an exact expiration time or a TTL value for specific fields:

[HEXPIRE](https://redis.io/docs/latest/commands/hexpire/): set the **remaining** TTL in seconds.

[HPEXPIRE](https://redis.io/docs/latest/commands/hpexpire/): set the **remaining** TTL in milliseconds.

[HEXPIREAT](https://redis.io/docs/latest/commands/hexpireat/): set the expiration time to a timestamp specified in seconds.

[HPEXPIREAT](https://redis.io/docs/latest/commands/hpexpireat/): set the expiration time to a timestamp specified in milliseconds.

* Use the following commands to retrieve either the exact time when or the remaining TTL until specific fields will expire:

[HEXPIRETIME](https://redis.io/docs/latest/commands/hexpiretime/): get the expiration time as a timestamp in seconds.

[HPEXPIRETIME](https://redis.io/docs/latest/commands/hpexpiretime/): get the expiration time as a timestamp in milliseconds.

[HTTL](https://redis.io/docs/latest/commands/httl/): get the remaining TTL in seconds.

[HPTTL](https://redis.io/docs/latest/commands/hpttl/): get the remaining TTL in milliseconds.

* Use the following command to remove the expiration of specific fields:

[HPERSIST](https://redis.io/docs/latest/commands/hpersist/): remove the expiration.

### Common Field Expiration Use Cases

Get Back to this use cases when you have more experience and this feature is more mature.

* **Event Tracking**: Use a hash key to store events from the last hour. Set each event's TTL to one hour. Use HLEN to count events from the past hour.
* **Fraud Detection**: Create a hash with hourly counters for events. Set each field's TTL to 48 hours. Query the hash to get the number of events per hour for the last 48 hours.
* **Customer Session Management**: Store customer data in hash keys. Create a new hash key for each session and add a session field to the customer’s hash key. Expire both the session key and the session field in the customer’s hash key automatically when the session expires.

I add: *what this means is probably that you create a hash key for each customer, whenever a session is created for that customer, you create a hash kay for that session and store the session data in this key, then you keep this hash key as a field inside the customers hash, then you have all the active sessions for the customer in its hash.*

*When the session keys expire, you then expire their corresponding fields in the customer’s hash using field expiration.*

* **Active Session Tracking**: Store all active sessions in a hash key. Set each session's TTL to expire automatically after inactivity. Use HLEN to count active sessions.

## Performance

Most Redis hash commands are O(1).

A few commands, such as [HKEYS](https://redis.io/docs/latest/commands/hkeys/), [HVALS](https://redis.io/docs/latest/commands/hvals/), [HGETALL](https://redis.io/docs/latest/commands/hgetall/), and most of the expiration-related commands, are O(n), where *n* is the number of field-value pairs.

*I add: the O(n) refers to for example the number of fields you want to set TTLs on.*

## Limits

Every hash can store up to 4,294,967,295 (2^32 - 1) field-value pairs. In practice, your hashes are limited only by the overall memory on the VMs hosting your Redis deployment.

# 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 then RPOP -> queue

LPUSH then LPOP -> STACK

## List Commands

* [LPUSH](https://redis.io/docs/latest/commands/lpush/) adds a new element to the head of a list;
* [RPUSH](https://redis.io/docs/latest/commands/rpush/) adds to the tail.
* [LPOP](https://redis.io/docs/latest/commands/lpop/) removes and returns an element from the head of a list at the same time;
* [RPOP](https://redis.io/docs/latest/commands/rpop/) does the same but from the tails of a list.
* [LLEN](https://redis.io/docs/latest/commands/llen/) returns the length of a list.
* [LMOVE](https://redis.io/docs/latest/commands/lmove/) **atomically** moves elements from one list to another.(e.g., From the left of the source to the right of the destination)
* [LRANGE](https://redis.io/docs/latest/commands/lrange/) returns the value of a range of indexes and both the start and the stop indexes are included in the result. Note that -1 is the index to the last node and -2 is the index to the penultimate node and so on.
* [LTRIM](https://redis.io/docs/latest/commands/ltrim/) reduces a list to the specified range of elements(both the start and stop indexes will be included). Used for example to make sure that the list won’t grow larger than 90 elements.

Lists also support several blocking commands. For example:

* [BLPOP](https://redis.io/docs/latest/commands/blpop/) removes and returns an element from the head of a list. **If the list is empty, the command blocks until an element becomes available or until the specified timeout is reached.**
* [BLMOVE](https://redis.io/docs/latest/commands/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 or until the specified timeout.

## What are Lists?

To explain the List data type, it's better to start with a little bit of theory, as the term List is often used in an improper way by information technology folks. For instance**, "Python Lists" are not what the name may suggest (Linked Lists), but rather Arrays (the same data type is called Array in Ruby actually).**

From a very general point of view a List is just a sequence of ordered elements: 10,20,1,2,3 is a list. But the properties of a List implemented using an Array are very different from the properties of a List implemented using a Linked List.

I add: *I assume by “ordered” they mean that you can access the elements via indices as opposed to a set where index of elements doesn’t make so much sense.*

**Redis lists are implemented via Linked Lists**. **This means that even if you have millions of elements inside a list, the operation of adding a new element in the head or in the tail of the list is performed in constant time.** The speed of adding a new element with the LPUSH command to the head of a list with ten elements is the same as adding an element to the head of list with 10 million elements.

Redis Lists are implemented with linked lists because for a database system it is crucial to be able to add elements to a very long list in a very fast way. Another strong advantage is that Redis Lists can be taken at constant length in constant time(I think they’re talking about capped lists that we’re going to cover in the upcoming sub-sections).

What's the downside?

**Accessing an element by index is very fast in lists implemented with an Array (constant time indexed access) and not so fast in lists implemented by linked lists** (where the operation requires an amount of work **proportional to the index of the accessed element**).

**When fast access to the middle of a large collection of elements is important, there is a different data structure that can be used, called sorted sets.**

## Common Use Cases for Lists

Lists are useful for a number of tasks. Two very representative use cases are the following:

* Remember the latest updates posted by users into a social network.
* Communication between processes, using a consumer-producer pattern where the producer pushes items into a list, and a consumer (usually a worker) consumes those items and executes actions. Redis has special list commands to make this use case both more reliable and efficient.

For example, both the popular Ruby libraries [resque](https://github.com/resque/resque) and [sidekiq](https://github.com/mperham/sidekiq) use Redis lists under the hood in order to implement background jobs.

The popular Twitter social network [takes the latest tweets](http://www.infoq.com/presentations/Real-Time-Delivery-Twitter) posted by users into Redis lists.

To describe a common use case step by step, imagine your home page shows the latest photos published in a photo sharing social network and you want to speedup access.

* Every time a user posts a new photo, we add its ID into a list with [LPUSH](https://redis.io/docs/latest/commands/lpush/).
* When users visit the home page, we use LRANGE 0 9 in order to get the latest 10 posted items.

## Capped Lists

In many use cases we just want to use lists to store the *latest items*, whatever they are: social network updates, logs, or anything else.

Redis allows us to use lists as a capped collection, only remembering the latest N items and discarding all the oldest items using the [LTRIM](https://redis.io/docs/latest/commands/ltrim/) command.

The [LTRIM](https://redis.io/docs/latest/commands/ltrim/) command is similar to [LRANGE](https://redis.io/docs/latest/commands/lrange/), but **instead of displaying the specified range of elements** **it sets this range as the new list value**. All the elements outside the given range are removed.

For example, if you're adding bikes on the end of a list of repairs, but only want to worry about the 3 that have been on the list **the longest**:

> RPUSH bikes:repairs bike:1 bike:2 bike:3 bike:4 bike:5

(integer) 5

> LTRIM bikes:repairs 0 2

OK

> LRANGE bikes:repairs 0 -1

1) "bike:1"

2) "bike:2"

3) "bike:3"

The above [LTRIM](https://redis.io/docs/latest/commands/ltrim/) command tells Redis to keep just list elements from index 0 to 2, everything else will be discarded. This allows for a very simple but useful pattern: doing a List push operation + a List trim operation together to add a new element and discard elements exceeding a limit. Using [LTRIM](https://redis.io/docs/latest/commands/ltrim/) with negative indexes can then be used to keep only the 3 most recently added:

> RPUSH bikes:repairs bike:1 bike:2 bike:3 bike:4 bike:5

(integer) 5

> LTRIM bikes:repairs **-3 -1**

OK

> LRANGE bikes:repairs 0 -1

1) "bike:3"

2) "bike:4"

3) "bike:5"

The above combination adds new elements and keeps only the 3 newest elements into the list. With [LRANGE](https://redis.io/docs/latest/commands/lrange/) you can access the top items without any need to remember very old data.

Note: while [LRANGE](https://redis.io/docs/latest/commands/lrange/) is technically an O(N) command, accessing small ranges towards the head or the tail of the list is a constant time operation.

## Blocking Operations on Lists

Lists have a special feature that make them suitable **to implement queues, and in general as a building block for inter process communication systems**: blocking operations.

Imagine you want to push items into a list with one process, and use a different process in order to actually do some kind of work with those items. This is the usual producer/consumer setup, and can be implemented in the following simple way:

* To push items into the list, producers call [LPUSH](https://redis.io/docs/latest/commands/lpush/).
* To extract / process items from the list, consumers call [RPOP](https://redis.io/docs/latest/commands/rpop/).

However, it is possible that sometimes the list is empty and there is nothing to process, so [RPOP](https://redis.io/docs/latest/commands/rpop/) just returns NULL. In this case a consumer is forced to wait some time and retry again with [RPOP](https://redis.io/docs/latest/commands/rpop/). This is called *polling*, and is not a good idea in this context because it has several drawbacks:

1. Forces Redis and clients to process useless commands (all the requests when the list is empty will get no actual work done, they'll just return NULL).
2. Adds a delay to the processing of items, since after a worker receives a NULL, it waits some time. To make the delay smaller, we could wait less between calls to [RPOP](https://redis.io/docs/latest/commands/rpop/), with the effect of amplifying problem number 1, i.e. more useless calls to Redis.

So Redis implements commands called [BRPOP](https://redis.io/docs/latest/commands/brpop/) and [BLPOP](https://redis.io/docs/latest/commands/blpop/) which are versions of [RPOP](https://redis.io/docs/latest/commands/rpop/) and [LPOP](https://redis.io/docs/latest/commands/lpop/) able to block if the list is empty: **they'll return to the caller only when a new element is added to the list, or when a user-specified timeout is reached.**

This is an example of a [BRPOP](https://redis.io/docs/latest/commands/brpop/) call we could use in the worker:

> RPUSH bikes:repairs bike:1 bike:2

(integer) 2

> BRPOP bikes:repairs 1

1) "bikes:repairs"

2) "bike:2"

> BRPOP bikes:repairs 1

1) "bikes:repairs"

2) "bike:1"

> BRPOP bikes:repairs 1

(nil)

(2.01s)

It means: "wait for elements in the list bikes:repairs, but return if after 1 second no element is available".

**Note that you can use 0 as timeout to wait for elements forever, and you can also specify multiple lists and not just one**, **in order to wait on multiple lists at the same time, and get notified when the first list receives an element.**

*if you wait on multiple lists, the very first one that gets populated will result in BLPOP to return and the waiting ends. If one of them already has some values, the command will return immediately. I checked for what happens If more than one has values when the command is called and apparently the pop happens for the first list.*

A few things to note about [BRPOP](https://redis.io/docs/latest/commands/brpop/):

1. **Clients are served in an ordered way**: the first client that blocked waiting for a list, is served first when an element is pushed by some other client, and so forth.
2. The return value is different compared to [RPOP](https://redis.io/docs/latest/commands/rpop/): it is a two-element array since it **also includes the name of the key**, because [**BRPOP**](https://redis.io/docs/latest/commands/brpop/)**and**[**BLPOP**](https://redis.io/docs/latest/commands/blpop/)**are able to block waiting for elements from multiple lists.**
3. **If the timeout is reached, NULL is returned.**

There are more things you should know about lists and blocking ops. You can read more on the following:

* It is possible to build safer queues or rotating queues using [LMOVE](https://redis.io/docs/latest/commands/lmove/).(HOW and WHY?)
* There is also a blocking variant of the command, called [BLMOVE](https://redis.io/docs/latest/commands/blmove/).

## Automatic Creation and Removal of Keys

So far in our examples we never had to create empty lists before pushing elements, or removing empty lists when they no longer have elements inside. It is Redis' responsibility to delete keys when lists are left empty, or to create an empty list if the key does not exist and we are trying to add elements to it, for example, with [LPUSH](https://redis.io/docs/latest/commands/lpush/).

This is not specific to lists, it applies to all the Redis data types composed of multiple elements -- **Streams, Sets, Sorted Sets and Hashes.**

Basically, we can summarize the behavior with three rules:

1. When we add an element to an aggregate data type, if the target key does not exist, an empty aggregate data type is created before adding the element.
2. When we remove elements from an aggregate data type, if the value remains empty, the key is automatically destroyed. **The Stream data type is the only exception to this rule.**
3. Calling a read-only command such as [LLEN](https://redis.io/docs/latest/commands/llen/) (which returns the length of the list), or a write command removing elements, with an empty key, always produces the same result **as if the key is holding an empty aggregate type of the type the command expects to find.**

## Performance

List operations that access its head or tail are O(1), which means they're highly efficient. However, commands that manipulate elements within a list are usually O(n). Examples of these include [LINDEX](https://redis.io/docs/latest/commands/lindex/), [LINSERT](https://redis.io/docs/latest/commands/linsert/), and [LSET](https://redis.io/docs/latest/commands/lset/). Exercise caution when running these commands, mainly when operating on large lists.

## Limits

The max length of a Redis list is 2^32 - 1 (4,294,967,295) elements.

## Alternatives

Consider [Redis streams](https://redis.io/docs/latest/develop/data-types/streams/) as an alternative to lists when you need to store and process an indeterminate series of events.

# 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**.

## Set Commands

* [SADD](https://redis.io/docs/latest/commands/sadd/) adds a new member to a set. if you add a member that already exists, it will be ignored.
* [SREM](https://redis.io/docs/latest/commands/srem/) removes the specified member from the set.
* [SISMEMBER](https://redis.io/docs/latest/commands/sismember/) tests a string for set membership. It can be used on single as well as multiple items:

> SISMEMBER bikes:racing:france bike:1

(integer) 1

> SMISMEMBER bikes:racing:france bike:2 bike:3 bike:4

1) (integer) 1

2) (integer) 1

3) (integer) 0

* SMEMBERS returns all the elements. There is no order guarantee with a set. Redis is free to return the elements in any order at every call.
* [SINTER](https://redis.io/docs/latest/commands/sinter/) returns the set of members that two or more sets have in common (i.e., the intersection).
* [SCARD](https://redis.io/docs/latest/commands/scard/) returns the size (a.k.a. cardinality) of a set.
* SDIFF Returns the members of the set resulting from the difference **between the first set and all the successive sets**.( Keys that do not exist are considered to be empty sets.) For example:

key1 = {a,b,c,d}

key2 = {c}

key3 = {a,c,e}

SDIFF key1 key2 key3 = {b,d}

You'll also note that the order of sets passed to [SDIFF](https://redis.io/docs/latest/commands/sdiff/) matters, since the difference is not commutative.

* you can use the [SPOP](https://redis.io/docs/latest/commands/spop/) command to remove a random item from a set. You can also return a random item from a set without removing it using the [SRANDMEMBER](https://redis.io/docs/latest/commands/srandmember/) command.

## Performance

Most set operations, including adding, removing, and checking whether an item is a set member, are O(1). This means that they're highly efficient.

However, for large sets with hundreds of thousands of members or more, you should exercise caution when running the [SMEMBERS](https://redis.io/docs/latest/commands/smembers/) command. This command is O(n) and returns the entire set in a single response. As an alternative, consider the [SSCAN](https://redis.io/docs/latest/commands/sscan/), which lets you retrieve all members of a set **iteratively**.

## Alternatives

Sets membership checks on large datasets (or on streaming data) **can use a lot of memory**. If you're concerned about memory usage **and don't need perfect precision**, consider a [Bloom filter or Cuckoo filter](https://redis.io/docs/latest/develop/data-types/probabilistic/bloom-filter/) as an alternative to a set. *These are Probabilistic data structures and give approximations of statistics rather than precise values.*

**Redis sets are frequently used as a kind of index**. If you need to index and query your data, **consider** the [JSON](https://redis.io/docs/latest/develop/data-types/json/) data type and the [Redis Query Engine](https://redis.io/docs/latest/develop/ai/search-and-query/) features.

## Limits

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

# Sorted Sets