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.

# Sets