Redis Administration

This page contains topics related to the administration of Redis instances. Every topic is self contained in form of a FAQ. New topics will be created in the future.

Redis setup hints

- We suggest deploying Redis using the **Linux operating system**. Redis is also tested heavily on osx, and tested from time to time on FreeBSD and OpenBSD systems. However Linux is where we do all the major stress testing, and where most production deployments are working.
- Make sure to set the Linux kernel **overcommit memory setting to 1**. Add `vm.overcommit_memory = 1` to `/etc/sysctl.conf` and then reboot or run the command `sysctl vm.overcommit_memory=1` for this to take effect immediately.
- Make sure to **setup some swap** in your system (we suggest as much as swap as memory). If Linux does not have swap and your Redis instance accidentally consumes too much memory, either Redis will crash for out of memory or the Linux kernel OOM killer will kill the Redis process.
- If you are using Redis in a very write-heavy application, while saving an RDB file on disk or rewriting the AOF log **Redis may use up to 2 times the memory normally used**. The additional memory used is proportional to the number of memory pages modified by writes during the saving process, so it is often proportional to the number of keys (or aggregate types items) touched during this time. Make sure to size your memory accordingly.
- Even if you have persistence disabled, Redis will need to perform RDB saves if you use replication.
- The use of Redis persistence with **EC2 EBS volumes is discouraged** since EBS performance is usually poor. Use ephemeral storage to persist and then move your persistence files to EBS when possible.
- If you are deploying using a virtual machine that uses the **Xen hypervisor you may experience slow fork() times**. This may block Redis from a few milliseconds up to a few seconds depending on the dataset size. Check the latency page for more information. This problem is not common to other hypervisors.
- Use `daemonize no` when run under daemontools.

Upgrading or restarting a Redis instance without downtime

Redis is designed to be a very long running process in your server. For instance many configuration options can be modified without any kind of restart using the CONFIG SET command.

Starting from Redis 2.2 it is even possible to switch from AOF to RDB snapshots persistence or the other way around without restarting Redis. Check the output of the 'CONFIG GET *' command for more information.

However from time to time a restart is mandatory, for instance in order to upgrade the Redis process to a newer version, or when you need to modify some configuration parameter that is currently not supported by the CONFIG command.

The following steps provide a very commonly used way in order to avoid any downtime.

- Setup your new Redis instance as a slave for your current Redis instance. In order to do so you need a different server, or a server that has enough RAM to keep two instances of Redis running at the same

time.

- If you use a single server, make sure that the slave is started in a different port than the master instance, otherwise the slave will not be able to start at all.
- Wait for the replication initial synchronization to complete (check the slave log file).
- Make sure using INFO that there are the same number of keys in the master and in the slave. Check with redis-cli that the slave is working as you wish and is replying to your commands.
- Configure all your clients in order to use the new instance (that is, the slave).
- Once you are sure that the master is no longer receiving any query (you can check this with the MONITOR command), elect the slave to master using the **SLAVEOF NO ONE** command, and shut down your master.

How fast is Redis?

Redis includes the `redis-benchmark` utility that simulates SETs/GETs done by N clients at the same time sending M total queries (it is similar to the Apache's `ab` utility). Below you'll find the full output of a benchmark executed against a Linux box.

The following options are supported:

```
Usage: redis-benchmark [-h <host>] [-p <port>] [-c <clients>] [-n <requests]> [-k <boolean>]
```

You need to have a running Redis instance before launching the benchmark. A typical example would be:

```
redis-benchmark -q -n 100000
```

Using this tool is quite easy, and you can also write your own benchmark, but as with any benchmarking activity, there are some pitfalls to avoid.

Pitfalls and misconceptions

The first point is obvious: the golden rule of a useful benchmark is to only compare apples and apples. Different versions of Redis can be compared on the same workload for instance. Or the same version of Redis, but with different options. If you plan to compare Redis to something else, then it is important to evaluate the functional and technical differences, and take them in account.

- Redis is a server: all commands involve network or IPC roundtrips. It is meaningless to compare it to embedded data stores such as SQLite, Berkeley DB, Tokyo/Kyoto Cabinet, etc ... because the cost of most operations is precisely dominated by network/protocol management.
- Redis commands return an acknowledgment for all usual commands. Some other data stores do not (for instance MongoDB does not implicitly acknowledge write operations). Comparing Redis to stores involving one-way queries is only mildly useful.
- Naively iterating on synchronous Redis commands does not benchmark Redis itself, but rather measure your network (or IPC) latency. To really test Redis, you need multiple connections (like redis-benchmark) and/or to use pipelining to aggregate several commands and/or multiple threads or processes.
- Redis is an in-memory data store with some optional persistency options. If you plan to compare it to transactional servers (MySQL, PostgreSQL, etc ...), then you should consider activating AOF and decide of a suitable fsync policy.
- Redis is a single-threaded server. It is not designed to benefit from multiple CPU cores. People are supposed to launch several Redis instances to scale out on several cores if needed. It is not really fair to compare one single Redis instance to a multi-threaded data store.

A common misconception is that redis-benchmark is designed to make Redis performances look stellar, the throughput achieved by redis-benchmark being somewhat artificial, and not achievable by a real application. This is actually plain wrong.

The redis-benchmark program is a quick and useful way to get some figures and evaluate the performance of a Redis instance on a given hardware. However, by default, it does not represent the maximum throughput a Redis instance can sustain. Actually, by using pipelining and a fast client (hiredis), it is fairly easy to write a

program generating more throughput than redis-benchmark. The default behavior of redis-benchmark is to achieve throughput by exploiting concurrency only (i.e. it creates several connections to the server). It does not use pipelining or any parallelism at all (one pending query per connection at most, and no multi-threading).

To run a benchmark using pipelining mode (and achieve higher throughputs), you need to explicitly use the -P option. Please note that it is still a realistic behavior since a lot of Redis based applications actively use pipelining to improve performance.

Finally, the benchmark should apply the same operations, and work in the same way with the multiple data stores you want to compare. It is absolutely pointless to compare the result of redis-benchmark to the result of another benchmark program and extrapolate.

For instance, Redis and memcached in single-threaded mode can be compared on GET/SET operations. Both are in-memory data stores, working mostly in the same way at the protocol level. Provided their respective benchmark application is aggregating queries in the same way (pipelining) and use a similar number of connections, the comparison is actually meaningful.

This perfect example is illustrated by the dialog between Redis (antirez) and memcached (dormando) developers.

antirez 1 - On Redis, Memcached, Speed, Benchmarks and The Toilet

dormando - Redis VS Memcached (slightly better bench)

antirez 2 - An update on the Memcached/Redis benchmark

You can see that in the end, the difference between the two solutions is not so staggering, once all technical aspects are considered. Please note both Redis and memcached have been optimized further after these benchmarks ...

Finally, when very efficient servers are benchmarked (and stores like Redis or memcached definitely fall in this category), it may be difficult to saturate the server. Sometimes, the performance bottleneck is on client side, and not server-side. In that case, the client (i.e. the benchmark program itself) must be fixed, or perhaps scaled out, in order to reach the maximum throughput.

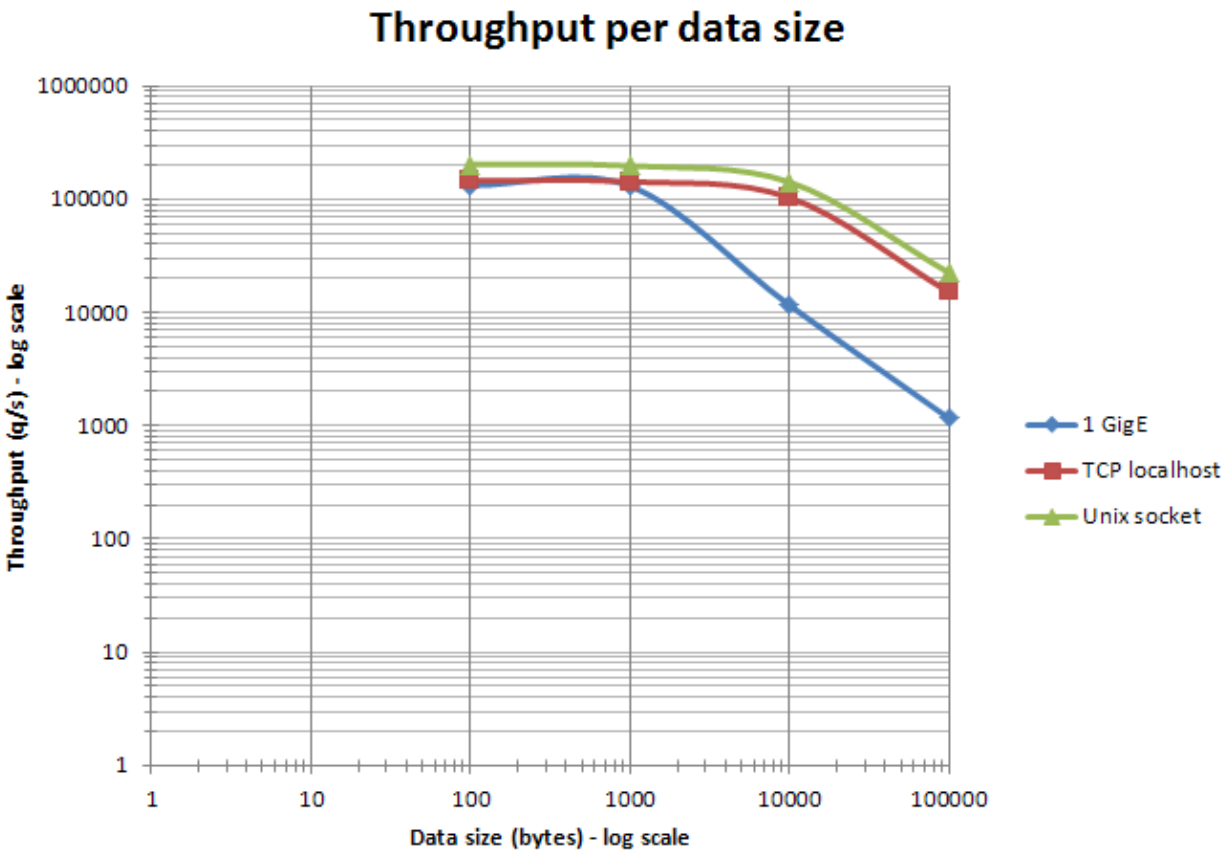
Factors impacting Redis performance

There are multiple factors having direct consequences on Redis performance. We mention them here, since they can alter the result of any benchmarks. Please note however, that a typical Redis instance running on a low end, non tuned, box usually provides good enough performance for most applications.

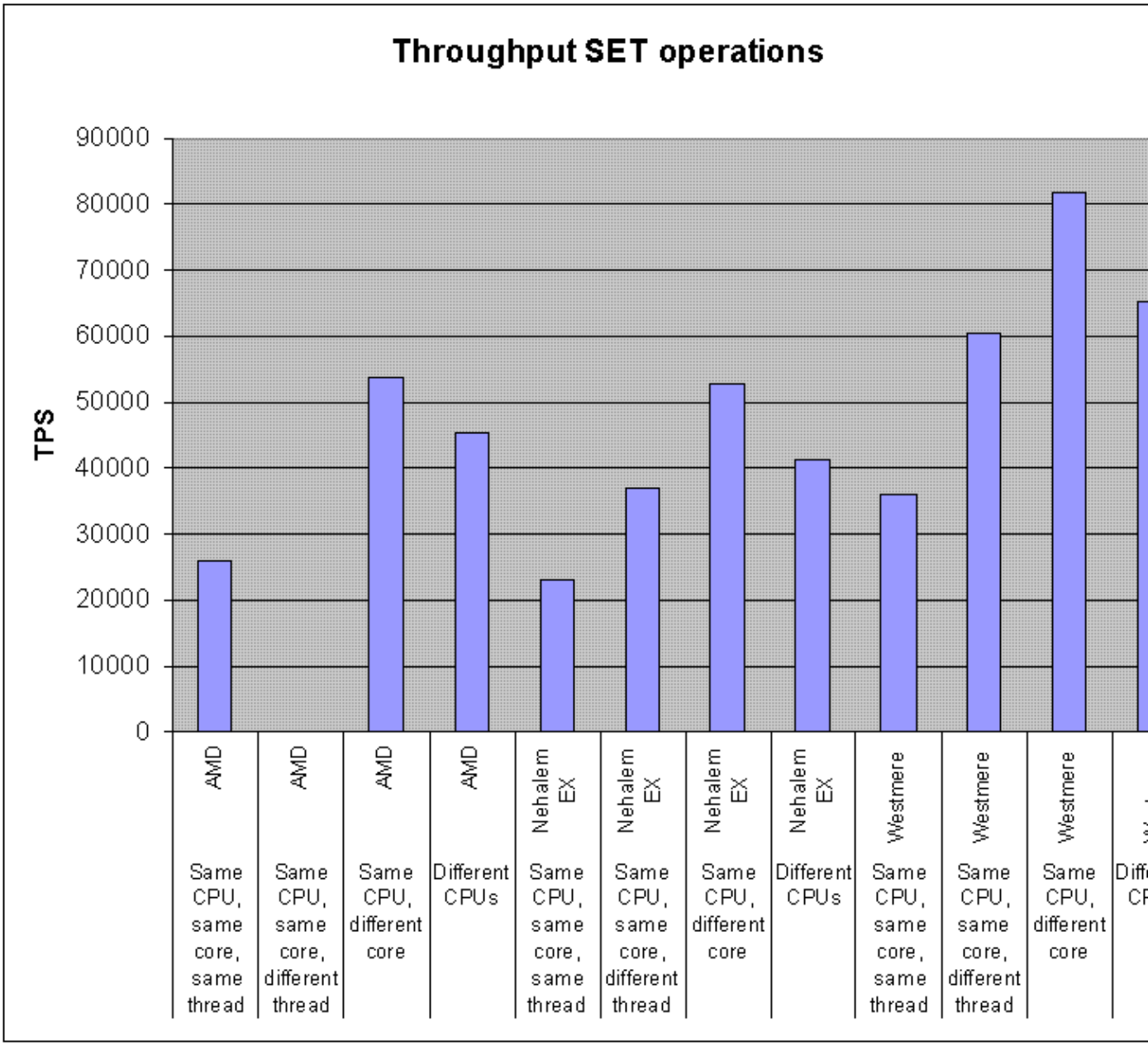
- Network bandwidth and latency usually have a direct impact on the performance. It is a good practice to use the ping program to quickly check the latency between the client and server hosts is normal before launching the benchmark. Regarding the bandwidth, it is generally useful to estimate the throughput in Gbits/s and compare it to the theoretical bandwidth of the network. For instance a benchmark setting 4 KB strings in Redis at 100000 q/s, would actually consume 3.2 Gbits/s of bandwidth and probably fit with a 10 GBits/s link, but not a 1 Gbits/s one. In many real world scenarios, Redis throughput is limited by the network well before being limited by the CPU. To consolidate several high-throughput Redis instances on a single server, it worth considering putting a

10 Gbits/s NIC or multiple 1 Gbits/s NICs with TCP/IP bonding.

- CPU is another very important factor. Being single-threaded, Redis favors fast CPUs with large caches and not many cores. At this game, Intel CPUs are currently the winners. It is not uncommon to get only half the performance on an AMD Opteron CPU compared to similar Nehalem EP/Westmere EP/Sandy bridge Intel CPUs with Redis. When client and server run on the same box, the CPU is the limiting factor with redis-benchmark.
- Speed of RAM and memory bandwidth seem less critical for global performance especially for small objects. For large objects (>10 KB), it may become noticeable though. Usually, it is not really cost effective to buy expensive fast memory modules to optimize Redis.
- Redis runs slower on a VM. Virtualization toll is quite high because for many common operations, Redis does not add much overhead on top of the required system calls and network interruptions. Prefer to run Redis on a physical box, especially if you favor deterministic latencies. On a state-of-the-art hypervisor (VMWare), result of redis-benchmark on a VM through the physical network is almost divided by 2 compared to the physical machine, with some significant CPU time spent in system and interruptions.
- When the server and client benchmark programs run on the same box, both the TCP/IP loopback and unix domain sockets can be used. It depends on the platform, but unix domain sockets can achieve around 50% more throughput than the TCP/IP loopback (on Linux for instance). The default behavior of redis-benchmark is to use the TCP/IP loopback.
- The performance benefit of unix domain sockets compared to TCP/IP loopback tends to decrease when pipelining is heavily used (i.e. long pipelines).
- When an ethernet network is used to access Redis, aggregating commands using pipelining is especially efficient when the size of the data is kept under the ethernet packet size (about 1500 bytes). Actually, processing 10 bytes, 100 bytes, or 1000 bytes queries almost result in the same throughput. See the graph below.

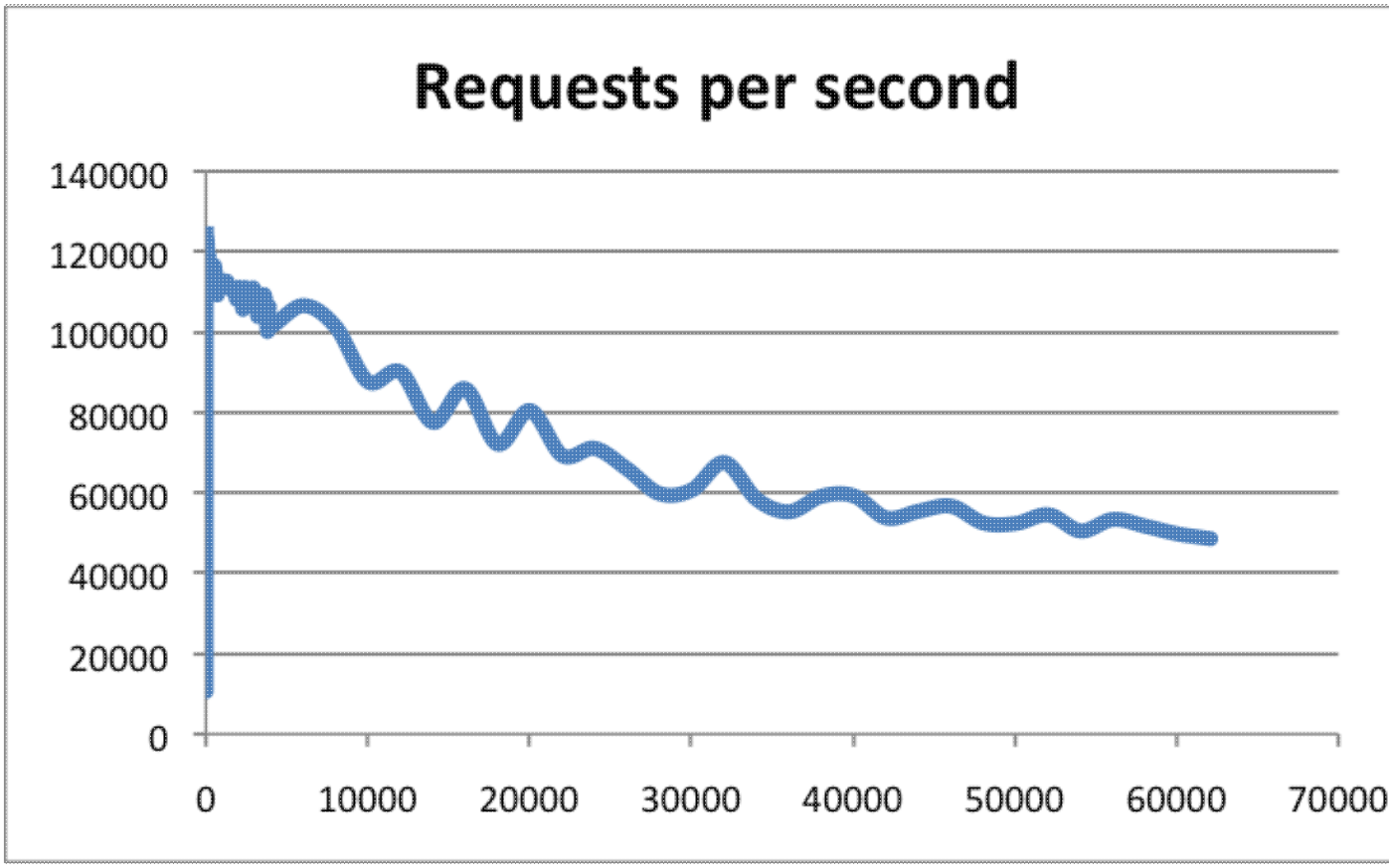


- On multi CPU sockets servers, Redis performance becomes dependant on the NUMA configuration and process location. The most visible effect is that redis-benchmark results seem non deterministic because client and server processes are distributed randomly on the cores. To get deterministic results, it is required to use process placement tools (on Linux: taskset or numactl). The most efficient combination is always to put the client and server on two different cores of the same CPU to benefit from the L3 cache. Here are some results of 4 KB SET benchmark for 3 server CPUs (AMD Istanbul, Intel Nehalem EX, and Intel Westmere) with different relative placements. Please note this benchmark is not meant to compare CPU models between themselves (CPUs exact model and frequency are therefore not disclosed).



- With high-end configurations, the number of client connections is also an important factor. Being based on epoll/kqueue, Redis event loop is quite scalable. Redis has already been benchmarked at more than 60000 connections, and was still able to sustain 50000 q/s in these conditions. As a rule of thumb, an instance with 30000 connections can only process half the throughput achievable with 100

connections. Here is an example showing the throughput of a Redis instance per number of connections:



- With high-end configurations, it is possible to achieve higher throughput by tuning the NIC(s) configuration and associated interruptions. Best throughput is achieved by setting an affinity between Rx/Tx NIC queues and CPU cores, and activating RPS (Receive Packet Steering) support. More information in this thread. Jumbo frames may also provide a performance boost when large objects are used.
- Depending on the platform, Redis can be compiled against different memory allocators (libc malloc, jemalloc, tcmalloc), which may have different behaviors in term of raw speed, internal and external fragmentation. If you did not compile Redis by yourself, you can use the INFO command to check the mem_allocator field. Please note most benchmarks do not run long enough to generate significant external fragmentation (contrary to production Redis instances).

Other things to consider

One important goal of any benchmark is to get reproducible results, so they can be compared to the results of other tests.

- A good practice is to try to run tests on isolated hardware as far as possible. If it is not possible, then the system must be monitored to check the benchmark is not impacted by some external activity.
- Some configurations (desktops and laptops for sure, some servers as well) have a variable CPU core frequency mechanism. The policy controlling this mechanism can be set at the OS level. Some CPU models are more aggressive than others at adapting the frequency of the CPU cores to the workload. To get reproducible results, it is better to set the highest possible fixed frequency for all the CPU

cores involved in the benchmark.
- An important point is to size the system accordingly to the benchmark. The system must have enough RAM and must not swap. On Linux, do not forget to set the overcommit_memory parameter correctly. Please note 32 and 64 bits Redis instances have not the same memory footprint.
- If you plan to use RDB or AOF for your benchmark, please check there is no other I/O activity in the system. Avoid putting RDB or AOF files on NAS or NFS shares, or on any other devices impacting your network bandwidth and/or latency (for instance, EBS on Amazon EC2).
- Set Redis logging level (loglevel parameter) to warning or notice. Avoid putting the generated log file on a remote filesystem.
- Avoid using monitoring tools which can alter the result of the benchmark. For instance using INFO at regular interval to gather statistics is probably fine, but MONITOR will impact the measured performance significantly.

Example of benchmark result

- The test was done with 50 simultaneous clients performing 100000 requests.
- The value SET and GET is a 256 bytes string.
- The Linux box is running *Linux 2.6*, it's *Xeon X3320 2.5 GHz*.
- Text executed using the loopback interface (127.0.0.1).

Results: *about 110000 SETs per second, about 81000 GETs per second.*

Latency percentiles

```
$ redis-benchmark -n 100000
```

Notes: changing the payload from 256 to 1024 or 4096 bytes does not change the numbers significantly (but reply packets are glued together up to 1024 bytes so GETs may be slower with big payloads). The same for the number of clients, from 50 to 256 clients I got the same numbers. With only 10 clients it starts to get a bit slower.

You can expect different results from different boxes. For example a low profile box like *Intel core duo T5500 clocked at 1.66 GHz running Linux 2.6* will output the following:

```
$ ./redis-benchmark -q -n 100000
```

Another one using a 64 bit box, a Xeon L5420 clocked at 2.5 GHz:

```
$ ./redis-benchmark -q -n 100000
```

Example of benchmark results with optimized high-end server hardware

- Redis version **2.4.2**
- Default number of connections, payload size = 256
- The Linux box is running *SLES10 SP3 2.6.16.60-0.54.5-smp*, CPU is 2 x *Intel X5670 @ 2.93 GHz*.
- Text executed while running redis server and benchmark client on the same CPU, but different cores.

Using a unix domain socket:

```
$ numactl -C 6 ./redis-benchmark -q -n 100000 -s /tmp/redis.sock -d 256
```

Using the TCP loopback:

```
$ numactl -C 6 ./redis-benchmark -q -n 100000 -d 256
```

This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

Redis configuration

Redis is able to start without a configuration file using a built-in default configuration, however this setup is only recommanded for testing and development purposes.

The proper way to configure Redis is by providing a Redis configuration file, usually called `redis.conf`.

The `redis.conf` file contains a number of directives that have a very simple format:

```
keyword argument1 argument2 ... argumentN
```

This is an example of configuration directive:

```
slaveof 127.0.0.1 6380
```

It is possible to provide strings containing spaces as arguments using quotes, as in the following example:

```
requirepass "hello world"
```

The list of configuration directives, and their meaning and intended usage is available in the self documented example redis.conf shipped into the Redis distribution.

- The self documented redis.conf for Redis 2.6.
- The self documented redis.conf for Redis 2.4.

Passing arguments via command line

Since Redis 2.6 it is possible to also pass Redis configuration parameters using the command line directly. This is very useful for testing purposes. The following is an example that stats a new Redis instance using port 6380 as a slave of the instance running at 127.0.0.1 port 6379.

```
./redis-server --port 6380 --slaveof 127.0.0.1 6379
```

The format of the arguments passed via the command line is exactly the same as the one used in the redis.conf file, with the exception that the keyword is prefixed with `--`.

Note that internally this generates an in-memory temporary config file (possibly concatenating the config file passed by the user if any) where arguments are translated into the format of redis.conf.

Changing Redis configuration while the server is running

It is possible to reconfigure Redis on the fly without stopping and restarting the service, or querying the current configuration programmatically using the special commands CONFIG SET and CONFIG GET

Not all the configuration directives are supported in this way, but most are supported as expected. Please refer to the CONFIG SET and CONFIG GET pages for more information.

Note that modifying the configuration on the fly **has no effects on the redis.conf file** so at the next restart of Redis the old configuration will be used instead.

Make sure to also modify the `redis.conf` file accordingly to the configuration you set using CONFIG SET. There are plans to provide a `CONFIG REWRITE` command that will be able to run the `redis.conf` file rewriting the configuration accordingly to the current server configuration, without modifying the comments and the structure of the current file.

Configuring Redis as a cache

If you plan to use Redis just as a cache where every key will have an expire set, you may consider using the following configuration instead (assuming a max memory limit of 2 megabytes as an example):

```
maxmemory 2mb
```

In this configuration there is no need for the application to set a time to live for keys using the EXPIRE command (or equivalent) since all the keys will be evicted using an approximated LRU algorithm as long as we hit the 2 megabyte memory limit.

This is more memory effective as setting expires on keys uses additional memory. Also an LRU behavior is usually to prefer compared to a fixed expire for every key, so that the *working set* of your data (the keys that are used more frequently) will likely last more.

Basically in this configuration Redis acts in a similar way to memcached.

When Redis is used as a cache in this way, if the application also requires the use Redis as a store, it is strongly suggested to create two Redis instances, one as a cache, configured in this way, and one as a store, configured accordingly to your persistence needs and only holding keys that are not about cached data.

Note: The user is adviced to read the example redis.conf to check how the other maxmemory policies available work.

A fifteen minute introduction to Redis data types

As you already probably know Redis is not a plain key-value store, actually it is a *data structures server*, supporting different kind of values. That is, you can set more than just strings as values of keys. All the following data types are supported as values:

- Binary-safe strings.
- Lists of binary-safe strings.
- Sets of binary-safe strings, that are collection of unique unsorted elements. You can think at this as a Ruby hash where all the keys are set to the 'true' value.
- Sorted sets, similar to Sets but where every element is associated to a floating number score. The elements are taken sorted by score. You can think of this as Ruby hashes where the key is the element and the value is the score, but where elements are always taken in order without requiring a sorting operation.

It's not always trivial to grasp how these data types work and what to use in order to solve a given problem from the command reference, so this document is a crash course to Redis data types and their most used patterns.

For all the examples we'll use the `redis-cli` utility, that's a simple but handy command line utility to issue commands against the Redis server.

Redis keys

Redis keys are binary safe, this means that you can use any binary sequence as a key, from a string like "foo" to the content of a JPEG file. The empty string is also a valid key.

A few other rules about keys:

- Too long keys are not a good idea, for instance a key of 1024 bytes is not a good idea not only memory-wise, but also because the lookup of the key in the dataset may require several costly key-comparisons.
- Too short keys are often not a good idea. There is little point in writing "u:1000:pwd" as key if you can write instead "user:1000:password", the latter is more readable and the added space is little compared to the space used by the key object itself and the value object. However it is not possible to deny that short keys will consume a bit less memory.
- Try to stick with a schema. For instance "object-type:id:field" can be a nice idea, like in "user:1000:password". I like to use dots for multi-words fields, like in "comment:1234:reply.to".

The string type

This is the simplest Redis type. If you use only this type, Redis will be something like a memcached server with persistence.

Let's play a bit with the string type:

```
$ redis-cli set mykey "my binary safe value"
```

As you can see using the SET command and the GET command is trivial to set values to strings and have the strings returned back.

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

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

```
$ redis-cli set counter 100
```

The INCR command parses the string value as an integer, increments it by one, and finally sets the obtained value as the new string value. There are other similar commands like INCRBY, DECR and DECRBY. Internally it's always the same command, acting in a slightly different way.

What does it mean that INCR is atomic? That even multiple clients issuing INCR against the same key will never incur into a race condition. For instance it can never happen that client 1 read "10", client 2 read "10" at the same time, both increment to 11, and set the new value of 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.

Another interesting operation on string is the GETSET command, that does just what its name suggests: Set a key to a new value, returning the old value as result. Why this is useful? Example: you have a system that increments a Redis key using the INCR command every time your web site receives a new visit. You want to collect this information one time every hour, without losing a single key. You can GETSET the key, assigning it the new value of "0" and reading the old value back.

The List type

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), they are actually 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*.

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*. Adding a new element with the LPUSH command to the head of a ten elements list is the same speed as adding an element to the head of a 10 million elements list.

What's the downside? Accessing an element *by index* is very fast in lists implemented with an Array and not so fast in lists implemented by linked lists.

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, as you'll see in a moment, that Redis Lists can be taken at constant length in constant time.

First steps with Redis lists

The LPUSH command adds a new element into a list, on the left (at the head), while the RPUSH command adds a new element into a list, on the right (at the tail). Finally the LRANGE command extracts ranges of elements from lists:

```
$ redis-cli rpush messages "Hello how are you?"
```

Note that LRANGE takes two indexes, the first and the last element of the range to return. Both the indexes can be negative to tell Redis to start to count from the end, so -1 is the last element, -2 is the penultimate element of the list, and so forth.

As you can guess from the example above, lists could be used in order to implement a chat system. Another use is as queues in order to route messages between different processes. But the key point is that *you can use Redis lists every time you require to access data in the same order they are added*. This will not require any SQL ORDER BY operation, will be very fast, and will scale to millions of elements even with a toy Linux box.

For instance in ranking systems like that used by social news site reddit.com you can add every new submitted link into a List, and with LRANGE it's possible to paginate results in a trivial way.

In a blog engine implementation you can have a list for every post, where to push blog comments, and so forth.

Pushing IDs instead of the actual data in Redis lists

In the above example we pushed our "objects" (simply messages in the example) directly inside the Redis list, but this is often not the way to go, as objects can be referenced in multiple times: in a list to preserve their chronological order, in a Set to remember they are about a specific category, in another list but only if this object matches some kind of requisite, and so forth.

Let's return back to the reddit.com example. A better pattern for adding submitted links (news) to the list is the following:

```
$ redis-cli incr next.news.id
```

We obtained a unique incremental ID for our news object just incrementing a key, then used this ID to create the object setting a key for every field in the object. Finally the ID of the new object was pushed on the *submitted.news* list.

This is just the start. Check the command reference and read about all the other list related commands. You can remove elements, rotate lists, get and set elements by index, and of course retrieve the length of the list with LLEN.

Redis Sets

Redis Sets are unordered collection of binary-safe strings. The SADD command adds a new element to a set. It's also possible to do a number of other operations against sets like testing if a given element already exists, performing the intersection, union or difference between multiple sets and so forth. An example is worth 1000 words:

```
$ redis-cli sadd myset 1
```

I added three elements to my set and told Redis to return back all the elements. As you can see they are not sorted.

Now let's check if a given element exists:

```
$ redis-cli sismember myset 3
```

"3" is a member of the set, while "30" is not. Sets are very good for expressing relations between objects. For instance we can easily use Redis Sets in order to implement tags.

A simple way to model this is to have a Set for every object containing its associated tag IDs, and a Set for every tag containing the object IDs that have that tag.

For instance if our news ID 1000 is tagged with tag 1,2,5 and 77, we can specify the following five Sets - one Set for the object's tags, and four Sets for the four tags:

```
$ redis-cli sadd news:1000:tags 1
```

To get all the tags for a given object is trivial:

```
$ redis-cli smembers news:1000:tags
```

But there are other non trivial operations that are still easy to implement using the right Redis commands. For instance we may want a list of all the objects with the tags 1, 2, 10, and 27 together. We can do this using the SINTER that performs the intersection between different sets. So in order to reach our goal we can just use:

```
$ redis-cli sinter tag:1:objects tag:2:objects tag:10:objects tag:27:objects
```

Look at the command reference to discover other Set related commands, there are a bunch of interesting ones. Also make sure to check the SORT command as both Redis Sets and Lists are sortable.

A digression: How to get unique identifiers for strings

In our tags example we showed tag IDs without mention of how the IDs can be obtained. Basically for every tag added to the system, you need a unique identifier. You also want to be sure that there are no race conditions if multiple clients are trying to add the same tag at the same time. Also, if a tag already exists, you want its ID returned, otherwise a new unique ID should be created and associated to the tag.

Redis 1.4 will add the Hash type. With it it will be trivial to associate strings with unique IDs, but how to do this today with the current commands exported by Redis in a reliable way?

Our first attempt (that is broken) can be the following. Let's suppose we want to get a unique ID for the tag "redis":

- In order to make this algorithm binary safe (they are just tags but think to utf8, spaces and so forth) we start performing the SHA1 digest of the tag. SHA1(redis) = b840fc02d524045429941cc15f59e41cb7be6c52.
- Let's check if this tag is already associated with a unique ID with the command *GET tag:b840fc02d524045429941cc15f59e41cb7be6c52:id*.

- If the above GET returns an ID, return it back to the user. We already have the unique ID.
- Otherwise... create a new unique ID with *INCR next.tag.id* (assume it returned 123456).
- Finally associate this new ID to our tag with *SET tag:b840fc02d524045429941cc15f59e41cb7be6c52:id 123456* and return the new ID to the caller.

Nice. Or rather.. broken! What about if two clients perform these commands at the same time trying to get the unique ID for the tag "redis"? If the timing is right they'll both get *nil* from the GET operation, will both increment the *next.tag.id* key and will set two times the key. One of the two clients will return the wrong ID to the caller. To fix the algorithm is not hard fortunately, and this is the sane version:

- In order to make this algorithm binary safe (they are just tags but think to utf8, spaces and so forth) we start performing the SHA1 digest of the tag. SHA1(redis) = b840fc02d524045429941cc15f59e41cb7be6c52.
- Let's check if this tag is already associated with a unique ID with the command *GET tag:b840fc02d524045429941cc15f59e41cb7be6c52:id*.
- If the above GET returns an ID, return it back to the user. We already have the unique ID.
- Otherwise... create a new unique ID with *INCR next.tag.id* (assume it returned 123456).
- Finally associate this new ID to our tag with *SETNX tag:b840fc02d524045429941cc15f59e41cb7be6c52:id 123456*. By using SETNX if a different client was faster than this one the key will not be setted. Not only, SETNX returns 1 if the key is set, 0 otherwise. So... let's add a final step to our computation.
- If SETNX returned 1 (We set the key) return 123456 to the caller, it's our tag ID, otherwise perform *GET tag:b840fc02d524045429941cc15f59e41cb7be6c52:id* and return the value to the caller.

Sorted sets

Sets are a very handy data type, but... they are a bit too unsorted in order to fit well for a number of problems ;) This is why Redis 1.2 introduced Sorted Sets. They are very similar to Sets, collections of binary-safe strings, but this time with an associated score, and an operation similar to the List LRANGE operation to return items in order, but working against Sorted Sets, that is, the ZRANGE command.

Basically Sorted Sets are in some way the Redis equivalent of Indexes in the SQL world. For instance in our reddit.com example above there was no mention about how to generate the actual home page with news raked by user votes and time. We'll see how sorted sets can fix this problem, but it's better to start with something simpler, illustrating the basic working of this advanced data type. Let's add a few selected hackers with their year of birth as "score".

```
$ redis-cli zadd hackers 1940 "Alan Kay"
```

For sorted sets it's a joke to return these hackers sorted by their birth year because actually *they are already sorted*. Sorted sets are implemented via a dual-ported data structure containing both a skip list and a hash table, so every time we add an element Redis performs an O(log(N)) operation. That's good, but when we ask for sorted elements Redis does not have to do any work at all, it's already all sorted:

```
$ redis-cli zrange hackers 0 -1
```

Didn't know that Linus was younger than Yukihiro btw ;)

What if I want to order them the opposite way, youngest to oldest? Use ZREVRANGE instead of ZRANGE:

```
$ redis-cli zrevrange hackers 0 -1
```

A very important note, ZSets have just a "default" ordering but you are still free to call the SORT command against sorted sets to get a different ordering (but this time the server will waste CPU). An alternative for having multiple orders is to add every element in multiple sorted sets at the same time.

Operating on ranges

Sorted sets are more powerful than this. They can operate on ranges. Let's get all the individuals that were born up to the 1950 inclusive. We use the ZRANGEBYSCORE command to do it:

```
$ redis-cli zrangebyscore hackers -inf 1950
```

We asked Redis to return all the elements with a score between negative infinity and 1950 (both extremes are included).

It's also possible to remove ranges of elements. Let's remove all the hackers born between 1940 and 1960 from the sorted set:

```
$ redis-cli zremrangebyscore hackers 1940 1960
```

ZREMRANGEBYSCORE is not the best command name, but it can be very useful, and returns the number of removed elements.

Back to the Reddit example

For the last time, back to the Reddit example. Now we have a decent plan to populate a sorted set in order to generate the home page. A sorted set can contain all the news that are not older than a few days (we remove old entries from time to time using ZREMRANGEBYSCORE). A background job gets all the elements from this sorted set, get the user votes and the time of the news, and computes the score to populate the *reddit.home.page* sorted set with the news IDs and associated scores. To show the home page we just have to perform a blazingly fast call to ZRANGE.

From time to time we'll remove very old news from the *reddit.home.page* sorted set to keep our system working with fresh news only.

Updating the scores of a sorted set

Just a final note before wrapping up this tutorial. Sorted sets scores can be updated at any time. Just calling again ZADD against an element already included in the sorted set will update its score (and position) in O(log(N)), so sorted sets are suitable even when there are tons of updates.

This tutorial is in no way complete and has covered just the basics. Read the command reference to discover a lot more.

Thanks for reading, Salvatore.

Data types

Strings

Strings are the most basic kind of Redis value. Redis Strings are binary safe, this means that a Redis string can contain any kind of data, for instance a JPEG image or a serialized Ruby object.

A String value can be at max 512 Megabytes in length.

You can do a number of interesting things using strings in Redis, for instance you can:

- Use Strings as atomic counters using commands in the INCR family: INCR, DECR, INCRBY.
- Append to strings with the APPEND command.
- Use Strings as a random access vectors with GETRANGE and SETRANGE.
- Encode a lot of data in little space, or create a Redis backed Bloom Filter using GETBIT and SETBIT.

Check all the available string commands for more information.

Lists

Redis Lists are simply lists of strings, sorted by insertion order. It is possible to add elements to a Redis List pushing new elements on the head (on the left) or on the tail (on the right) of the list.

The LPUSH command inserts a new element on the head, while RPUSH inserts a new element on the tail. A new list is created when one of this operations is performed against an empty key. Similarly the key is removed from the key space if a list operation will empty the list. These are very handy semantics since all the list commands will behave exactly like they were called with an empty list if called with a non-existing key as argument.

Some example of list operations and resulting lists:

```
LPUSH mylist a   # now the list is "a"
```

The max length of a list is 2^{32} - 1 elements (4294967295, more than 4 billion of elements per list).

The main features of Redis Lists from the point of view of time complexity are the support for constant time insertion and deletion of elements near the head and tail, even with many millions of inserted items. Accessing elements is very fast near the extremes of the list but is slow if you try accessing the middle of a very big list, as it is an O(N) operation.

You can do many interesting things with Redis Lists, for instance you can:

- Model a timeline in a social network, using LPUSH in order to add new elements in the user time line, and using LRANGE in order to retrieve a few of recently inserted items.

- You can use LPUSH together with LTRIM to create a list that never exceeds a given number of elements, but just remembers the latest N elements.
- Lists can be used as a message passing primitive, See for instance the well known Resque Ruby library for creating background jobs.
- You can do a lot more with lists, this data type supports a number of commands, including blocking commands like BLPOP. Please check all the available commands operating on lists for more information.

Sets

Redis Sets are an unordered collection of Strings. It is possible to add, remove, and test for existence of members in O(1) (constant time regardless of the number of elements contained inside the Set).

Redis Sets have the desirable property of not allowing repeated members. Adding the same element multiple times will result in a set having a single copy of this element. Practically speaking this means that adding a member does not require a *check if exists then add* operation.

A very interesting thing about Redis Sets is that they support a number of server side commands to compute sets starting from existing sets, so you can do unions, intersections, differences of sets in very short time.

The max number of members in a set is 2^{32} - 1 (4294967295, more than 4 billion of members per set).

You can do many interesting things using Redis Sets, for instance you can:

- You can track unique things using Redis Sets. Want to know all the unique IP addresses visiting a given blog post? Simply use SADD every time you process a page view. You are sure repeated IPs will not be inserted.
- Redis Sets are good to represent relations. You can create a tagging system with Redis using a Set to represent every tag. Then you can add all the IDs of all the objects having a given tag into a Set representing this particular tag, using the SADD command. Do you want all the IDs of all the Objects having a three different tags at the same time? Just use SINTER.
- You can use Sets to extract elements at random using the SPOP or SRANDMEMBER commands.
- As usually check the full list of Set commands for more information.

Hashes

Redis Hashes are maps between string fields and string values, so they are the perfect data type to represent objects (eg: A User with a number of fields like name, surname, age, and so forth):

```
@cli
```

A hash with a few fields (where few means up to one hundred or so) is stored in a way that takes very little space, so you can store millions of objects in a small Redis instance.

While Hashes are used mainly to represent objects, they are capable of storing many elements, so you can use Hashes for many other tasks as well.

Every hash can store up to 2^{32} - 1 field-value pairs (more than 4 billion).

Check the full list of Hash commands for more information.

Sorted sets

Redis Sorted Sets are, similarly to Redis Sets, non repeating collections of Strings. The difference is that every member of a Sorted Set is associated with score, that is used in order to take the sorted set ordered, from the smallest to the greatest score. While members are unique, scores may be repeated.

With sorted sets you can add, remove, or update elements in a very fast way (in a time proportional to the logarithm of the number of elements). Since elements are *taken in order* and not ordered afterwards, you can also get ranges by score or by rank (position) in a very fast way. Accessing the middle of a sorted set is also very fast, so you can use Sorted Sets as a smart list of non repeating elements where you can quickly access everything you need: elements in order, fast existence test, fast access to elements in the middle!

In short with sorted sets you can do a lot of tasks with great performance that are really hard to model in other kind of databases.

With Sorted Sets you can:

- Take a leader board in a massive online game, where every time a new score is submitted you update it using ZADD. You can easily take the top users using ZRANGE, you can also, given an user name, return its rank in the listing using ZRANK. Using ZRANK and ZRANGE together you can show users with a score similar to a given user. All very *quickly*.
- Sorted Sets are often used in order to index data that is stored inside Redis. For instance if you have many hashes representing users, you can use a sorted set with elements having the age of the user as the score and the ID of the user as the value. So using ZRANGEBYSCORE it will be trivial and fast to retrieve all the users with a given interval of ages.
- Sorted Sets are probably the most advanced Redis data types, so take some time to check the full list of Sorted Set commands to discover what you can do with Redis!

This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Redis debugging guide

Redis is developed with a great stress on stability: we do our best with every release to make sure you'll experience a very stable product and no crashes. However even with our best efforts it is impossible to avoid all the critical bugs with 100% of success.

When Redis crashes it produces a detailed report of what happened, however sometimes looking at the crash report is not enough, nor it is possible for the Redis core team to reproduce the issue independently: in this scenario we need help from the user that is able to reproduce the issue.

This little guide shows how to use GDB to provide all the informations the Redis developers will need to track the bug more easily.

What is GDB?

GDB is the Gnu Debugger: a program that is able to inspect the internal state of another program. Usually tracking and fixing a bug is an exercise in gathering more informations about the state of the program at the moment the bug happens, so GDB is an extremely useful tool.

GDB can be used in two ways:

- It can attach to a running program and inspect the state of it at runtime.
- It can inspect the state of a program that already terminated using what is called a *core file*, that is, the image of the memory at the time the program was running.

From the point of view of investigating Redis bugs we need to use both this GDB modes: the user able to reproduce the bug attaches GDB to his running Redis instance, and when the crash happens, he creates the `core` file that the in turn the developer will use to inspect the Redis internals at the time of the crash.

This way the developer can perform all the inspections in his computer without the help of the user, and the user is free to restart Redis in the production environment.

Compiling Redis without optimizations

By default Redis is compiled with the `-O2` switch, this means that compiler optimizations are enabled. This makes the Redis executable faster, but at the same time it makes Redis (like any other program) harder to inspect using GDB.

It is better to attach GDB to Redis compiled without optimizations using the `make noopt` command to compile it (instead of just using the plain `make` command). However if you have an already running Redis in production there is no need to recompile and restart it if this is going to create problems on your side. Even if by a lesser extent GDB still works against executables compiled with optimizations.

It is great if you make sure to recompile Redis with `make noopt` after the first crash, so that the next time it will be simpler to track the issue.

You should not be concerned with the loss of performances compiling Redis without optimizations, it is very

unlikely that this will cause problems in your environment since it is usually just a matter of a small percentage because Redis is not very CPU-bound (it does a lot of I/O to serve queries).

Attaching GDB to a running process

If you have an already running Redis server, you can attach GDB to it, so that if Redis will crash it will be possible to both inspect the internals and generate a `core dump` file.

After you attach GDB to the Redis process it will continue running as usually without any loss of performance, so this is not a dangerous procedure.

In order to attach GDB the first thing you need is the *process ID* of the running Redis instance (the *pid* of the process). You can easily obtain it using `redis-cli`:

```
$ redis-cli info | grep process_id
```

In the above example the process ID is **58414**.

- Login into your Redis server.
- (Optional but recommended) Start **screen** or **tmux** or any other program that will make sure that your GDB session will not be closed if your ssh connection will timeout. If you don't know what screen is do yourself a favour and Read this article
- Attach GDB to the running Redis server typing:

  gdb `<path-to-redis-executable>` `<pid>`

  For example: gdb /usr/local/bin/redis-server 58414

GDB will start and will attach to the running server printing something like the following:

```
Reading symbols for shared libraries + done
```

- At this point GDB is attached but **your Redis instance is blocked by GDB**. In order to let the Redis instance continue the execution just type **continue** at the GDB prompt, and press enter.

  ```
  (gdb) continue
  ```
- Done! Now your Redis instance has GDB attached. You can wait for... the next crash :)
- Now it's time to detach your screen / tmux session, if you are running GDB using it, pressing the usual **Ctrl-a a** key combination.

After the crash

Redis has a command to simulate a segmentation fault (in other words a bad crash) using the `DEBUG SEGFAULT` command (don't use it against a real production instance of course ;). So I'll use this command to crash my instance to show what happens in the GDB side:

```
(gdb) continue
```

As you can see GDB detected that Redis crashed, and was able to show me even the file name and line number causing the crash. This is already much better than the Redis crash report back trace (containing just

function names and binary offsets).

Obtaining the stack trace

The first thing to do is to obtain a full stack trace with GDB. This is as simple as using the **bt** command: (that is a short for backtrace):

```
(gdb) bt
```

This shows the backtrace, but we also want to dump the processor registers using the **info registers** command:

```
(gdb) info registers
```

Please **make sure to include** both this outputs in your bug report.

Obtaining the core file

The next step is to generate the core dump, that is the image of the memory of the running Redis process. This is performed using the `gcore` command:

```
(gdb) gcore
```

Now you have the core dump to send to the Redis developer, but **it is important to understand** that this happens to contain all the data that was inside the Redis instance at the time of the crash: Redis developers will make sure to don't share the content with any other, and will delete the file as soon as it is no longer used for debugging purposes, but you are warned that sending the core file you are sending your data.

If there are sensible stuff in the data set we suggest sending the dump directly to Salvatore Sanfilippo (that is the guy writing this doc) at the email address **antirez at gmail dot com**.

What to send to developers

Finally you can send everything to the Redis core team:

- The Redis executable you are using.
- The stack trace produced by the **bt** command, and the registers dump.
- The core file you generated with gdb.
- Informations about the operating system and GCC version, and Redis version you are using.

Thank you

Your help is extremely important! Many issues can only be tracked this way, thanks! It is also possible that helping Redis debugging you'll be among the winners of the next Redis Moka Award.

This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

FAQ

Why Redis is different compared to other key-value stores?

There are two main reasons.

- Redis is a different evolution path in the key-value DBs where values can contain more complex data types, with atomic operations defined against those data types. Redis data types are closely related to fundamental data structures and are exposed to the programmer as such, without additional abstraction layers.
- Redis is an in-memory but persistent on disk database, so it represents a different trade off where very high write and read speed is achieved with the limitation of data sets that can't be larger than memory. Another advantage of in memory databases is that the memory representation of complex data structure is much simpler to manipulate compared to the same data structure on disk, so Redis can do a lot with little internal complexity. At the same time an on-disk format that does not need to be suitable for random access is compact and always generated in an append-only fashion.

What's the Redis memory footprint?

To give you an example: 1 Million keys with the key being the natural numbers from 0 to 999999 and the string "Hello World" as value use 100MB on my Intel MacBook (32bit). Note that the same data stored linearly in an unique string takes something like 16MB, this is expected because with small keys and values there is a lot of overhead. Memcached will perform similarly, but a bit better as Redis has more overhead (type information, refcount and so forth) to represent different kinds of objects.

With large keys/values the ratio is much better of course.

64 bit systems will use considerably more memory than 32 bit systems to store the same keys, especially if the keys and values are small, this is because pointers takes 8 bytes in 64 bit systems. But of course the advantage is that you can have a lot of memory in 64 bit systems, so in order to run large Redis servers a 64 bit system is more or less required.

I like Redis high level operations and features, but I don't like that it takes everything in memory and I can't have a dataset larger the memory. Plans to change this?

In the past the Redis developers experimented with Virtual Memory and other systems in order to allow larger than RAM datasets, but after all we are very happy if we can do one thing well: data served from memory, disk used for storage. So for now there are no plans to create an on disk backend for Redis. Most of what Redis is, after all, is a direct result of its current design.

However many large users solved the issue of large datasets distributing among multiple Redis nodes, using client-side hashing. **Craigslist** and **Groupon** are two examples.

At the same time Redis Cluster, an automatically distributed and fault tolerant implementation of a Redis subset, is a work in progress, and may be a good solution for many use cases.

If my dataset is too big for RAM and I don't want to use consistent hashing or other ways to distribute the dataset across different nodes, what I can do to use Redis anyway?

A possible solution is to use both an on disk DB (MySQL or others) and Redis at the same time, basically take the state on Redis (metadata, small but often written info), and all the other things that get accessed very frequently: user auth tokens, Redis Lists with chronologically ordered IDs of the last N-comments, N-posts, and so on. Then use MySQL (or any other) as a simple storage engine for larger data, that is just create a table with an auto-incrementing ID as primary key and a large BLOB field as data field. Access MySQL data only by primary key (the ID). The application will run the high traffic queries against Redis but when there is to take the big data will ask MySQL for specific resources IDs.

Is there something I can do to lower the Redis memory usage?

If you can use Redis 32 bit instances, and make good use of small hashes, lists, sorted sets, and sets of integers, since Redis is able to represent those data types in the special case of a few elements in a much more compact way.

What happens if Redis runs out of memory?

With modern operating systems malloc() returning NULL is not common, usually the server will start swapping and Redis performances will degrade so you'll probably notice there is something wrong.

The INFO command will report the amount of memory Redis is using so you can write scripts that monitor your Redis servers checking for critical conditions.

Alternatively can use the "maxmemory" option in the config file to put a limit to the memory Redis can use. If this limit is reached Redis will start to reply with an error to write commands (but will continue to accept read-only commands), or you can configure it to evict keys when the max memory limit is reached in the case you are using Redis for caching.

Background saving is failing with a fork() error under Linux even if I've a lot of free RAM!

Short answer: `echo 1 > /proc/sys/vm/overcommit_memory` :)

And now the long one:

Redis background saving schema relies on the copy-on-write semantic of fork in modern operating systems: Redis forks (creates a child process) that is an exact copy of the parent. The child process dumps the DB on disk and finally exits. In theory the child should use as much memory as the parent being a copy, but actually thanks to the copy-on-write semantic implemented by most modern operating systems the parent and child process will *share* the common memory pages. A page will be duplicated only when it changes in the child or in the parent. Since in theory all the pages may change while the child process is saving, Linux can't tell in advance how much memory the child will take, so if the `overcommit_memory` setting is set to zero fork will fail unless there is as much free RAM as required to really duplicate all the parent memory pages, with the result that if you have a Redis dataset of 3 GB and just 2 GB of free memory it will fail.

Setting `overcommit_memory` to 1 says Linux to relax and perform the fork in a more optimistic allocation fashion, and this is indeed what you want for Redis.

A good source to understand how Linux Virtual Memory work and other alternatives for `overcommit_memory` and `overcommit_ratio` is this classic from Red Hat Magazine, "Understanding Virtual Memory".

Are Redis on-disk-snapshots atomic?

Yes, redis background saving process is always fork(2)ed when the server is outside of the execution of a command, so every command reported to be atomic in RAM is also atomic from the point of view of the disk snapshot.

Redis is single threaded, how can I exploit multiple CPU / cores?

It's very unlikely that CPU becomes your bottleneck with Redis, as usually Redis is either memory or network bound. For instance using pipelining Redis running on an average Linux system can deliver even 500k requests per second, so if your application mainly uses O(N) or O(log(N)) commands it is hardly going to use too much CPU.

However to maximize CPU usage you can start multiple instances of Redis in the same box and treat them as different servers. At some point a single box may not be enough anyway, so if you want to use multiple CPUs you can start thinking at some way to shard earlier.

In Redis there are client libraries such Redis-rb (the Ruby client) and Predis (one of the most used PHP clients) that are able to handle multiple servers automatically using *consistent hashing*.

What is the maximum number of keys a single Redis instance can hold? and what the max number of elements in a List, Set, Sorted Set?

In theory Redis can handle up to 2^{32} keys, and was tested in practice to handle at least 250 million of keys per instance. We are working in order to experiment with larger values.

Every list, set, and sorted set, can hold 2^{32} elements.

In other words your limit is likely the available memory in your system.

What Redis means actually?

It means REmote DIctionary Server.

Why did you started the Redis project?

Originally Redis was started in order to scale LLOOGG. But after I got the basic server working I liked the idea to share the work with other guys, and Redis was turned into an open source project.

This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

Event Library

Why is an Event Library needed at all?

Let us figure it out through a series of Q&As.

Q: What do you expect a network server to be doing all the time?
A: Watch for inbound connections on the port its listening and accept them.

Q: Calling accept yields a descriptor. What do I do with it?
A: Save the descriptor and do a non-blocking read/write operation on it.

Q: Why does the read/write have to be non-blocking?
A: If the file operation (even a socket in Unix is a file) is blocking how could the server for example accept other connection requests when its blocked in a file I/O operation.

Q: I guess I have to do many such non-blocking operations on the socket to see when it's ready. Am I right?
A: Yes. That is what an event library does for you. Now you get it.

Q: How do Event Libraries do what they do?
A: They use the operating system's polling facility along with timers.

Q: So are there any open source event libraries that do what you just described?
A: Yes. `libevent` and `libev` are two such event libraries that I can recall off the top of my head.

Q: Does Redis use such open source event libraries for handling socket I/O?
A: No. For various reasons Redis uses its own event library.

Redis Event Library

Redis implements its own event library. The event library is implemented in `ae.c`.

The best way to understand how the Redis event library works is to understand how Redis uses it.

Event Loop Initialization

`initServer` function defined in `redis.c` initializes the numerous fields of the `redisServer` structure variable. One such field is the Redis event loop `el`:

```
aeEventLoop *el
```

`initServer` initializes `server.el` field by calling `aeCreateEventLoop` defined in `ae.c`. The definition of `aeEventLoop` is below:

```
typedef struct aeEventLoop
```

aeCreateEventLoop

`aeCreateEventLoop` first `mallocs` `aeEventLoop` structure then calls `ae_epoll.c:aeApiCreate`.

`aeApiCreate` `mallocs` `aeApiState` that has two fields - `epfd` that holds the `epoll` file descriptor returned by a call from `epoll_create` and `events` that is of type `struct epoll_event` define by the Linux `epoll` library. The use of the `events` field will be described later.

Next is `ae.c:aeCreateTimeEvent`. But before that `initServer` call `anet.c:anetTcpServer` that creates and returns a *listening descriptor*. The descriptor listens on *port 6379* by default. The returned *listening descriptor* is stored in `server.fd` field.

aeCreateTimeEvent

`aeCreateTimeEvent` accepts the following as parameters:

- `eventLoop`: This is `server.el` in `redis.c`
- milliseconds: The number of milliseconds from the current time after which the timer expires.
- `proc`: Function pointer. Stores the address of the function that has to be called after the timer expires.
- `clientData`: Mostly `NULL`.
- `finalizerProc`: Pointer to the function that has to be called before the timed event is removed from the list of timed events.

`initServer` calls `aeCreateTimeEvent` to add a timed event to `timeEventHead` field of `server.el`. `timeEventHead` is a pointer to a list of such timed events. The call to `aeCreateTimeEvent` from `redis.c:initServer` function is given below:

```
aeCreateTimeEvent(server.el /*eventLoop*/, 1 /*milliseconds*/, serverCron /*proc*/, NULL /*clie
```

redis.c:serverCron performs many operations that helps keep Redis running properly.

aeCreateFileEvent

The essence of aeCreateFileEvent function is to execute epoll_ctl system call which adds a watch for EPOLLIN event on the *listening descriptor* create by anetTcpServer and associate it with the epoll descriptor created by a call to aeCreateEventLoop.

Following is an explanation of what precisely aeCreateFileEvent does when called from redis.c:initServer.

initServer passes the following arguments to aeCreateFileEvent:

- server.el: The event loop created by aeCreateEventLoop. The epoll descriptor is got from server.el.
- server.fd: The *listening descriptor* that also serves as an index to access the relevant file event structure from the eventLoop->events table and store extra information like the callback function.
- AE_READABLE: Signifies that server.fd has to be watched for EPOLLIN event.
- acceptHandler: The function that has to be executed when the event being watched for is ready. This function pointer is stored in eventLoop->events[server.fd]->rfileProc.

This completes the initialization of Redis event loop.

Event Loop Processing

ae.c:aeMain called from redis.c:main does the job of processing the event loop that is initialized in the previous phase.

ae.c:aeMain calls ae.c:aeProcessEvents in a while loop that processes pending time and file events.

aeProcessEvents

ae.c:aeProcessEvents looks for the time event that will be pending in the smallest amount of time by calling ae.c:aeSearchNearestTimer on the event loop. In our case there is only one timer event in the event loop that was created by ae.c:aeCreateTimeEvent.

Remember, that timer event created by aeCreateTimeEvent has by now probably elapsed because it had a expiry time of one millisecond. Since, the timer has already expired the seconds and microseconds fields of the tvp timeval structure variable is initialized to zero.

The tvp structure variable along with the event loop variable is passed to ae_epoll.c:aeApiPoll.

aeApiPoll functions does a epoll_wait on the epoll descriptor and populates the eventLoop->fired table with the details:

- fd: The descriptor that is now ready to do a read/write operation depending on the mask value.
- mask: The read/write event that can now be performed on the corresponding descriptor.

`aeApiPoll` returns the number of such file events ready for operation. Now to put things in context, if any client has requested for a connection then `aeApiPoll` would have noticed it and populated the `eventLoop->fired` table with an entry of the descriptor being the *listening descriptor* and mask being `AE_READABLE`.

Now, `aeProcessEvents` calls the `redis.c:acceptHandler` registered as the callback. `acceptHandler` executes accept on the *listening descriptor* returning a *connected descriptor* with the client. `redis.c:createClient` adds a file event on the *connected descriptor* through a call to `ae.c:aeCreateFileEvent` like below:

```
if (aeCreateFileEvent(server.el, c->fd, AE_READABLE,
```

`c` is the `redisClient` structure variable and `c->fd` is the connected descriptor.

Next the `ae.c:aeProcessEvent` calls `ae.c:processTimeEvents`

processTimeEvents

`ae.processTimeEvents` iterates over list of time events starting at `eventLoop->timeEventHead`.

For every timed event that has elapsed `processTimeEvents` calls the registered callback. In this case it calls the only timed event callback registered, that is, `redis.c:serverCron`. The callback returns the time in milliseconds after which the callback must be called again. This change is recorded via a call to `ae.c:aeAddMilliSeconds` and will be handled on the next iteration of `ae.c:aeMain` while loop.

That's all.

Hacking Strings

The implementation of Redis strings is contained in `sds.c` (`sds` stands for Simple Dynamic Strings).

The C structure `sdshdr` declared in `sds.h` represents a Redis string:

```
struct sdshdr {
```

The `buf` character array stores the actual string.

The `len` field stores the length of `buf`. This makes obtaining the length of a Redis string an O(1) operation.

The `free` field stores the number of additional bytes available for use.

Together the `len` and `free` field can be thought of as holding the metadata of the `buf` character array.

Creating Redis Strings

A new data type named `sds` is defined in `sds.h` to be a synonym for a character pointer:

```
typedef char *sds;
```

`sdsnewlen` function defined in `sds.c` creates a new Redis String:

```
sds sdsnewlen(const void *init, size_t initlen) {
```

Remember a Redis string is a variable of type `struct sdshdr`. But `sdsnewlen` returns a character pointer!!

That's a trick and needs some explanation.

Suppose I create a Redis string using `sdsnewlen` like below:

```
sdsnewlen("redis", 5);
```

This creates a new variable of type `struct sdshdr` allocating memory for `len` and `free` fields as well as for the `buf` character array.

```
sh = zmalloc(sizeof(struct sdshdr)+initlen+1); // initlen is length of init argument.
```

After `sdsnewlen` successfully creates a Redis string the result is something like:

```
-----------
```

`sdsnewlen` returns `sh->buf` to the caller.

What do you do if you need to free the Redis string pointed by `sh`?

You want the pointer `sh` but you only have the pointer `sh->buf`.

Can you get the pointer `sh` from `sh->buf`?

Yes. Pointer arithmetic. Notice from the above ASCII art that if you subtract the size of two longs from `sh->buf` you get the pointer `sh`.

The `sizeof` two longs happens to be the size of `struct sdshdr`.

Look at `sdslen` function and see this trick at work:

```
size_t sdslen(const sds s) {
```

Knowing this trick you could easily go through the rest of the functions in `sds.c`.

The Redis string implementation is hidden behind an interface that accepts only character pointers. The users of Redis strings need not care about how its implemented and treat Redis strings as a character pointer.

Virtual Memory technical specification

This document details the internals of the Redis Virtual Memory subsystem. The intended audience is not the final user but programmers willing to understand or modify the Virtual Memory implementation.

Keys vs Values: what is swapped out?

The goal of the VM subsystem is to free memory transferring Redis Objects from memory to disk. This is a very generic command, but specifically, Redis transfers only objects associated with *values*. In order to understand better this concept we'll show, using the DEBUG command, how a key holding a value looks from the point of view of the Redis internals:

```
redis> set foo bar
```

As you can see from the above output, the Redis top level hash table maps Redis Objects (keys) to other Redis Objects (values). The Virtual Memory is only able to swap *values* on disk, the objects associated to *keys* are always taken in memory: this trade off guarantees very good lookup performances, as one of the main design goals of the Redis VM is to have performances similar to Redis with VM disabled when the part of the dataset frequently used fits in RAM.

How does a swapped value looks like internally

When an object is swapped out, this is what happens in the hash table entry:

- The key continues to hold a Redis Object representing the key.
- The value is set to NULL

So you may wonder where we store the information that a given value (associated to a given key) was swapped out. Just in the key object!

This is how the Redis Object structure *robj* looks like:

```
/* The actual Redis Object */
```

As you can see there are a few fields about VM. The most important one is *storage*, that can be one of this values:

- REDIS*VM*MEMORY: the associated value is in memory.
- REDIS*VM*SWAPPED: the associated values is swapped, and the value entry of the hash table is just set to NULL.
- REDIS*VM*LOADING: the value is swapped on disk, the entry is NULL, but there is a job to load the object from the swap to the memory (this field is only used when threaded VM is active).
- REDIS*VM*SWAPPING: the value is in memory, the entry is a pointer to the actual Redis Object, but there is an I/O job in order to transfer this value to the swap file.

If an object is swapped on disk (REDIS*VM*SWAPPED or REDIS*VM*LOADING), how do we know where it is stored, what type it is, and so forth? That's simple: the *vtype* field is set to the original type of the Redis object swapped, while the *vm* field (that is a *redisObjectVM* structure) holds information about the location of

the object. This is the definition of this additional structure:

```
/* The VM object structure */
```

As you can see the structure contains the page at which the object is located in the swap file, the number of pages used, and the last access time of the object (this is very useful for the algorithm that select what object is a good candidate for swapping, as we want to transfer on disk objects that are rarely accessed).

As you can see, while all the other fields are using unused bytes in the old Redis Object structure (we had some free bit due to natural memory alignment concerns), the *vm* field is new, and indeed uses additional memory. Should we pay such a memory cost even when VM is disabled? No! This is the code to create a new Redis Object:

```
... some code ...
```

As you can see if the VM system is not enabled we allocate just `sizeof(*o)-sizeof(struct redisObjectVM)` of memory. Given that the *vm* field is the last in the object structure, and that this fields are never accessed if VM is disabled, we are safe and Redis without VM does not pay the memory overhead.

The Swap File

The next step in order to understand how the VM subsystem works is understanding how objects are stored inside the swap file. The good news is that's not some kind of special format, we just use the same format used to store the objects in .rdb files, that are the usual dump files produced by Redis using the SAVE command.

The swap file is composed of a given number of pages, where every page size is a given number of bytes. This parameters can be changed in redis.conf, since different Redis instances may work better with different values: it depends on the actual data you store inside it. The following are the default values:

```
vm-page-size 32
```

Redis takes a "bitmap" (an contiguous array of bits set to zero or one) in memory, every bit represent a page of the swap file on disk: if a given bit is set to 1, it represents a page that is already used (there is some Redis Object stored there), while if the corresponding bit is zero, the page is free.

Taking this bitmap (that will call the page table) in memory is a huge win in terms of performances, and the memory used is small: we just need 1 bit for every page on disk. For instance in the example below 134217728 pages of 32 bytes each (4GB swap file) is using just 16 MB of RAM for the page table.

Transferring objects from memory to swap

In order to transfer an object from memory to disk we need to perform the following steps (assuming non threaded VM, just a simple blocking approach):

- Find how many pages are needed in order to store this object on the swap file. This is trivially accomplished just calling the function `rdbSavedObjectPages` that returns the number of pages used by an object on disk. Note that this function does not duplicate the .rdb saving code just to understand what will be the length *after* an object will be saved on disk, we use the trick of opening /dev/null and writing the object there, finally calling `ftello` in order check the amount of bytes required. What we do basically is to save the object on a virtual very fast file, that is, /dev/null.

- Now that we know how many pages are required in the swap file, we need to find this number of contiguous free pages inside the swap file. This task is accomplished by the `vmFindContiguousPages` function. As you can guess this function may fail if the swap is full, or so fragmented that we can't easily find the required number of contiguous free pages. When this happens we just abort the swapping of the object, that will continue to live in memory.
- Finally we can write the object on disk, at the specified position, just calling the function `vmWriteObjectOnSwap`.

As you can guess once the object was correctly written in the swap file, it is freed from memory, the storage field in the associated key is set to REDIS*VM*SWAPPED, and the used pages are marked as used in the page table.

Loading objects back in memory

Loading an object from swap to memory is simpler, as we already know where the object is located and how many pages it is using. We also know the type of the object (the loading functions are required to know this information, as there is no header or any other information about the object type on disk), but this is stored in the *vtype* field of the associated key as already seen above.

Calling the function `vmLoadObject` passing the key object associated to the value object we want to load back is enough. The function will also take care of fixing the storage type of the key (that will be REDIS*VM*MEMORY), marking the pages as freed in the page table, and so forth.

The return value of the function is the loaded Redis Object itself, that we'll have to set again as value in the main hash table (instead of the NULL value we put in place of the object pointer when the value was originally swapped out).

How blocking VM works

Now we have all the building blocks in order to describe how the blocking VM works. First of all, an important detail about configuration. In order to enable blocking VM in Redis `server.vm_max_threads` must be set to zero. We'll see later how this max number of threads info is used in the threaded VM, for now all it's needed to now is that Redis reverts to fully blocking VM when this is set to zero.

We also need to introduce another important VM parameter, that is, `server.vm_max_memory`. This parameter is very important as it is used in order to trigger swapping: Redis will try to swap objects only if it is using more memory than the max memory setting, otherwise there is no need to swap as we are matching the user requested memory usage.

Blocking VM swapping

Swapping of object from memory to disk happens in the cron function. This function used to be called every second, while in the recent Redis versions on git it is called every 100 milliseconds (that is, 10 times per second). If this function detects we are out of memory, that is, the memory used is greater than the vm-max-memory setting, it starts transferring objects from memory to disk in a loop calling the function `vmSwapOneObect`. This function takes just one argument, if 0 it will swap objects in a blocking way, otherwise if it is 1, I/O threads are used. In the blocking scenario we just call it with zero as argument.

vmSwapOneObject acts performing the following steps:

- The key space in inspected in order to find a good candidate for swapping (we'll see later what a good candidate for swapping is).
- The associated value is transferred to disk, in a blocking way.
- The key storage field is set to REDIS*VM*SWAPPED, while the *vm* fields of the object are set to the right values (the page index where the object was swapped, and the number of pages used to swap it).
- Finally the value object is freed and the value entry of the hash table is set to NULL.

The function is called again and again until one of the following happens: there is no way to swap more objects because either the swap file is full or nearly all the objects are already transferred on disk, or simply the memory usage is already under the vm-max-memory parameter.

What values to swap when we are out of memory?

Understanding what's a good candidate for swapping is not too hard. A few objects at random are sampled, and for each their *swappability* is commuted as:

```
swappability = age*log(size_in_memory)
```

The age is the number of seconds the key was not requested, while size*in*memory is a fast estimation of the amount of memory (in bytes) used by the object in memory. So we try to swap out objects that are rarely accessed, and we try to swap bigger objects over smaller one, but the latter is a less important factor (because of the logarithmic function used). This is because we don't want bigger objects to be swapped out and in too often as the bigger the object the more I/O and CPU is required in order to transfer it.

Blocking VM loading

What happens if an operation against a key associated with a swapped out object is requested? For instance Redis may just happen to process the following command:

```
GET foo
```

If the value object of the `foo` key is swapped we need to load it back in memory before processing the operation. In Redis the key lookup process is centralized in the `lookupKeyRead` and `lookupKeyWrite` functions, this two functions are used in the implementation of all the Redis commands accessing the keyspace, so we have a single point in the code where to handle the loading of the key from the swap file to memory.

So this is what happens:

- The user calls some command having as argument a swapped key
- The command implementation calls the lookup function
- The lookup function search for the key in the top level hash table. If the value associated with the requested key is swapped (we can see that checking the *storage* field of the key object), we load it back in memory in a blocking way before to return to the user.

This is pretty straightforward, but things will get more *interesting* with the threads. From the point of view of the blocking VM the only real problem is the saving of the dataset using another process, that is, handling BGSAVE and BGREWRITEAOF commands.

Background saving when VM is active

The default Redis way to persist on disk is to create .rdb files using a child process. Redis calls the fork() system call in order to create a child, that has the exact copy of the in memory dataset, since fork duplicates the whole program memory space (actually thanks to a technique called Copy on Write memory pages are shared between the parent and child process, so the fork() call will not require too much memory).

In the child process we have a copy of the dataset in a given point in the time. Other commands issued by clients will just be served by the parent process and will not modify the child data.

The child process will just store the whole dataset into the dump.rdb file and finally will exit. But what happens when the VM is active? Values can be swapped out so we don't have all the data in memory, and we need to access the swap file in order to retrieve the swapped values. While child process is saving the swap file is shared between the parent and child process, since:

- The parent process needs to access the swap file in order to load values back into memory if an operation against swapped out values are performed.
- The child process needs to access the swap file in order to retrieve the full dataset while saving the data set on disk.

In order to avoid problems while both the processes are accessing the same swap file we do a simple thing, that is, not allowing values to be swapped out in the parent process while a background saving is in progress. This way both the processes will access the swap file in read only. This approach has the problem that while the child process is saving no new values can be transferred on the swap file even if Redis is using more memory than the max memory parameters dictates. This is usually not a problem as the background saving will terminate in a short amount of time and if still needed a percentage of values will be swapped on disk ASAP.

An alternative to this scenario is to enable the Append Only File that will have this problem only when a log rewrite is performed using the BGREWRITEAOF command.

The problem with the blocking VM

The problem of blocking VM is that... it's blocking :) This is not a problem when Redis is used in batch processing activities, but for real-time usage one of the good points of Redis is the low latency. The blocking VM will have bad latency behaviors as when a client is accessing a swapped out value, or when Redis needs to swap out values, no other clients will be served in the meantime.

Swapping out keys should happen in background. Similarly when a client is accessing a swapped out value other clients accessing in memory values should be served mostly as fast as when VM is disabled. Only the clients dealing with swapped out keys should be delayed.

All this limitations called for a non-blocking VM implementation.

Threaded VM

There are basically three main ways to turn the blocking VM into a non blocking one. * 1: One way is obvious, and in my opionion, not a good idea at all, that is, turning Redis itself into a theaded server: if every request is served by a different thread automatically other clients don't need to wait for blocked ones. Redis is

fast, exports atomic operations, has no locks, and is just 10k lines of code, *because* it is single threaded, so this was not an option for me. * 2: Using non-blocking I/O against the swap file. After all you can think Redis already event-loop based, why don't just handle disk I/O in a non-blocking fashion? I also discarded this possiblity because of two main reasons. One is that non blocking file operations, unlike sockets, are an incompatibility nightmare. It's not just like calling select, you need to use OS-specific things. The other problem is that the I/O is just one part of the time consumed to handle VM, another big part is the CPU used in order to encode/decode data to/from the swap file. This is I picked option three, that is... * 3: Using I/O threads, that is, a pool of threads handling the swap I/O operations. This is what the Redis VM is using, so let's detail how this works.

I/O Threads

The threaded VM design goals where the following, in order of importance:

- Simple implementation, little room for race conditions, simple locking, VM system more or less completely decoupled from the rest of Redis code.
- Good performances, no locks for clients accessing values in memory.
- Ability to decode/encode objects in the I/O threads.

The above goals resulted in an implementation where the Redis main thread (the one serving actual clients) and the I/O threads communicate using a queue of jobs, with a single mutex. Basically when main thread requires some work done in the background by some I/O thread, it pushes an I/O job structure in the `server.io_newjobs` queue (that is, just a linked list). If there are no active I/O threads, one is started. At this point some I/O thread will process the I/O job, and the result of the processing is pushed in the `server.io_processed` queue. The I/O thread will send a byte using an UNIX pipe to the main thread in order to signal that a new job was processed and the result is ready to be processed.

This is how the `iojob` structure looks like:

```
typedef struct iojob {
```

There are just three type of jobs that an I/O thread can perform (the type is specified by the `type` field of the structure):

- REDIS*IOJOB*LOAD: load the value associated to a given key from swap to memory. The object offset inside the swap file is `page`, the object type is `key->vtype`. The result of this operation will populate the `val` field of the structure.
- REDIS*IOJOB*PREPARE_SWAP: compute the number of pages needed in order to save the object pointed by `val` into the swap. The result of this operation will populate the `pages` field.
- REDIS*IOJOB*DO_SWAP: Transfer the object pointed by `val` to the swap file, at page offset `page`.

The main thread delegates just the above three tasks. All the rest is handled by the main thread itself, for instance finding a suitable range of free pages in the swap file page table (that is a fast operation), deciding what object to swap, altering the storage field of a Redis object to reflect the current state of a value.

Non blocking VM as probabilistic enhancement of blocking VM

So now we have a way to request background jobs dealing with slow VM operations. How to add this to the mix of the rest of the work done by the main thread? While blocking VM was aware that an object was swapped out just when the object was looked up, this is too late for us: in C it is not trivial to start a

background job in the middle of the command, leave the function, and re-enter in the same point the computation when the I/O thread finished what we requested (that is, no co-routines or continuations or alike).

Fortunately there was a much, much simpler way to do this. And we love simple things: basically consider the VM implementation a blocking one, but add an optimization (using non the no blocking VM operations we are able to perform) to make the blocking *very* unlikely.

This is what we do:

- Every time a client sends us a command, *before* the command is executed, we examine the argument vector of the command in search for swapped keys. After all we know for every command what arguments are keys, as the Redis command format is pretty simple.
- If we detect that at least a key in the requested command is swapped on disk, we block the client instead of really issuing the command. For every swapped value associated to a requested key, an I/O job is created, in order to bring the values back in memory. The main thread continues the execution of the event loop, without caring about the blocked client.
- In the meanwhile, I/O threads are loading values in memory. Every time an I/O thread finished loading a value, it sends a byte to the main thread using an UNIX pipe. The pipe file descriptor has a readable event associated in the main thread event loop, that is the function `vmThreadedIOCompletedJob`. If this function detects that all the values needed for a blocked client were loaded, the client is restarted and the original command called.

So you can think at this as a blocked VM that almost always happen to have the right keys in memory, since we pause clients that are going to issue commands about swapped out values until this values are loaded.

If the function checking what argument is a key fails in some way, there is no problem: the lookup function will see that a given key is associated to a swapped out value and will block loading it. So our non blocking VM reverts to a blocking one when it is not possible to anticipate what keys are touched.

For instance in the case of the SORT command used together with the GET or BY options, it is not trivial to know beforehand what keys will be requested, so at least in the first implementation, SORT BY/GET resorts to the blocking VM implementation.

Blocking clients on swapped keys

How to block clients? To suspend a client in an event-loop based server is pretty trivial. All we do is canceling its read handler. Sometimes we do something different (for instance for BLPOP) that is just marking the client as blocked, but not processing new data (just accumulating the new data into input buffers).

Aborting I/O jobs

There is something hard to solve about the interactions between our blocking and non blocking VM, that is, what happens if a blocking operation starts about a key that is also "interested" by a non blocking operation at the same time?

For instance while SORT BY is executed, a few keys are being loaded in a blocking manner by the sort command. At the same time, another client may request the same keys with a simple *GET key* command, that will trigger the creation of an I/O job to load the key in background.

The only simple way to deal with this problem is to be able to kill I/O jobs in the main thread, so that if a key that we want to load or swap in a blocking way is in the REDIS*VM*LOADING or REDIS*VM*SWAPPING state (that is, there is an I/O job about this key), we can just kill the I/O job about this key, and go ahead with the blocking operation we want to perform.

This is not as trivial as it is. In a given moment an I/O job can be in one of the following three queues:

- server.io_newjobs: the job was already queued but no thread is handling it.
- server.io_processing: the job is being processed by an I/O thread.
- server.io_processed: the job was already processed. The function able to kill an I/O job is `vmCancelThreadedIOJob`, and this is what it does:
- If the job is in the newjobs queue, that's simple, removing the iojob structure from the queue is enough as no thread is still executing any operation.
- If the job is in the processing queue, a thread is messing with our job (and possibly with the associated object!). The only thing we can do is waiting for the item to move to the next queue in a *blocking way*. Fortunately this condition happens very rarely so it's not a performance problem.
- If the job is in the processed queue, we just mark it as *canceled* marking setting the `canceled` field to 1 in the iojob structure. The function processing completed jobs will just ignored and free the job instead of really processing it.

Questions?

This document is in no way complete, the only way to get the whole picture is reading the source code, but it should be a good introduction in order to make the code review / understanding a lot simpler.

Something is not clear about this page? Please leave a comment and I'll try to address the issue possibly integrating the answer in this document.

Redis Internals documentation

Redis source code is not very big (just 20k lines of code for the 2.2 release) and we try hard to make it simple and easy to understand. However we have some documentation explaining selected parts of the Redis internals.

Redis dynamic strings

String is the basic building block of Redis types.

Redis is a key-value store. All Redis keys are strings and its also the simplest value type.

Lists, sets, sorted sets and hashes are other more complex value types and even these are composed of strings.

Hacking Strings documents the Redis String implementation details.

Redis Virtual Memory

We have a document explaining virtual memory implementation details, but warning: this document refers to the 2.0 VM implementation. 2.2 is different... and better.

Redis Event Library

Read event library to understand what an event library does and why its needed.

Redis event library documents the implementation details of the event library used by Redis.

This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by 

Introduction to Redis

Redis is an open source, advanced **key-value store**. It is often referred to as a **data structure server** since keys can contain strings, hashes, lists, sets and sorted sets.

You can run **atomic operations** on these types, like appending to a string; incrementing the value in a hash; pushing to a list; computing set intersection, union and difference; or getting the member with highest ranking in a sorted set.

In order to achieve its outstanding performance, Redis works with an **in-memory dataset**. Depending on your use case, you can persist it either by dumping the dataset to disk every once in a while, or by appending each command to a log.

Redis also supports trivial-to-setup master-slave replication, with very fast non-blocking first synchronization, auto-reconnection on net split and so forth.

Other features include a simple check-and-set mechanism, pub/sub and configuration settings to make Redis behave like a cache.

You can use Redis from most programming languages out there.

Redis is written in **ANSI C** and works in most POSIX systems like Linux, *BSD, OS X without external dependencies. Linux and OSX are the two operating systems where Redis is developed and more tested, and we **recommend using Linux for deploying**. Redis may work in Solaris-derived systems like SmartOS, but the support is *best effort*. There is no official support for Windows builds, although you may have some options.

This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Redis latency problems troubleshooting

This document will help you understand what the problem could be if you are experiencing latency problems with Redis.

In this context *latency* is the maximum delay between the time a client issues a command and the time the reply to the command is received by the client. Usually Redis processing time is extremely low, in the sub microsecond range, but there are certain conditions leading to higher latency figures.

Measuring latency

If you are experiencing latency problems, probably you know how to measure it in the context of your application, or maybe your latency problem is very evident even macroscopically. However redis-cli can be used to measure the latency of a Redis server in milliseconds, just try:

```
redis-cli --latency -h `host` -p `port`
```

Latency induced by network and communication

Clients connect to Redis using a TCP/IP connection or a Unix domain connection. The typical latency of a 1 GBits/s network is about 200 us, while the latency with a Unix domain socket can be as low as 30 us. It actually depends on your network and system hardware. On top of the communication itself, the system adds some more latency (due to thread scheduling, CPU caches, NUMA placement, etc ...). System induced latencies are significantly higher on a virtualized environment than on a physical machine.

The consequence is even if Redis processes most commands in sub microsecond range, a client performing many roundtrips to the server will have to pay for these network and system related latencies.

An efficient client will therefore try to limit the number of roundtrips by pipelining several commands together. This is fully supported by the servers and most clients. Aggregated commands like MSET/MGET can be also used for that purpose. Starting with Redis 2.4, a number of commands also support variadic parameters for all data types.

Here are some guidelines:

- If you can afford it, prefer a physical machine over a VM to host the server.
- Do not systematically connect/disconnect to the server (especially true for web based applications). Keep your connections as long lived as possible.
- If your client is on the same host than the server, use Unix domain sockets.
- Prefer to use aggregated commands (MSET/MGET), or commands with variadic parameters (if possible) over pipelining.
- Prefer to use pipelining (if possible) over sequence of roundtrips.
- Future version of Redis will support Lua server-side scripting (experimental branches are already available) to cover cases that are not suitable for raw pipelining (for instance when the result of a command is an input for the following commands).

On Linux, some people can achieve better latencies by playing with process placement (taskset), cgroups, real-time priorities (chrt), NUMA configuration (numactl), or by using a low-latency kernel. Please note

vanilla Redis is not really suitable to be bound on a **single** CPU core. Redis can fork background tasks that can be extremely CPU consuming like bgsave or AOF rewrite. These tasks must **never** run on the same core as the main event loop.

In most situations, these kind of system level optimizations are not needed. Only do them if you require them, and if you are familiar with them.

Single threaded nature of Redis

Redis uses a *mostly* single threaded design. This means that a single process serves all the client requests, using a technique called **multiplexing**. This means that Redis can serve a single request in every given moment, so all the requests are served sequentially. This is very similar to how Node.js works as well. However, both products are often not perceived as being slow. This is caused in part by the small about of time to complete a single request, but primarily because these products are designed to not block on system calls, such as reading data from or writing data to a socket.

I said that Redis is *mostly* single threaded since actually from Redis 2.4 we use threads in Redis in order to perform some slow I/O operations in the background, mainly related to disk I/O, but this does not change the fact that Redis serves all the requests using a single thread.

Latency generated by slow commands

A consequence of being single thread is that when a request is slow to serve all the other clients will wait for this request to be served. When executing normal commands, like **GET** or **SET** or **LPUSH** this is not a problem at all since this commands are executed in constant (and very small) time. However there are commands operating on many elements, like **SORT**, **LREM**, **SUNION** and others. For instance taking the intersection of two big sets can take a considerable amount of time.

The algorithmic complexity of all commands is documented. A good practice is to systematically check it when using commands you are not familiar with.

If you have latency concerns you should either not use slow commands against values composed of many elements, or you should run a replica using Redis replication where to run all your slow queries.

It is possible to monitor slow commands using the Redis Slow Log feature.

Additionally, you can use your favorite per-process monitoring program (top, htop, prstat, etc ...) to quickly check the CPU consumption of the main Redis process. If it is high while the traffic is not, it is usually a sign that slow commands are used.

Latency generated by fork

In order to generate the RDB file in background, or to rewrite the Append Only File if AOF persistence is enabled, Redis has to fork background processes. The fork operation (running in the main thread) can induce latency by itself.

Forking is an expensive operation on most Unix-like systems, since it involves copying a good number of objects linked to the process. This is especially true for the page table associated to the virtual memory mechanism.

For instance on a Linux/AMD64 system, the memory is divided in 4 KB pages. To convert virtual addresses to physical addresses, each process stores a page table (actually represented as a tree) containing at least a pointer per page of the address space of the process. So a large 24 GB Redis instance requires a page table of 24 GB / 4 KB * 8 = 48 MB.

When a background save is performed, this instance will have to be forked, which will involve allocating and copying 48 MB of memory. It takes time and CPU, especially on virtual machines where allocation and initialization of a large memory chunk can be expensive.

Fork time in different systems

Modern hardware is pretty fast to copy the page table, but Xen is not. The problem with Xen is not virtualization-specific, but Xen-specific. For instance using VMware or Virutal Box does not result into slow fork time. The following is a table that compares fork time for different Redis instance size. Data is obtained performing a BGSAVE and looking at the `latest_fork_usec` filed in the INFO command output.

- **Linux beefy VM on VMware** 6.0GB RSS forked in 77 milliseconds (12.8 milliseconds per GB).
- **Linux running on physical machine (Unknown HW)** 6.1GB RSS forked in 80 milliseconds (13.1 milliseconds per GB)
- **Linux running on physical machine (Xeon @ 2.27Ghz)** 6.9GB RSS forked into 62 millisecodns (9 milliseconds per GB).
- **Linux VM on 6sync (KVM)** 360 MB RSS forked in 8.2 milliseconds (23.3 millisecond per GB).
- **Linux VM on EC2 (Xen)** 6.1GB RSS forked in 1460 milliseconds (239.3 milliseconds per GB).
- **Linux VM on Linode (Xen)** 0.9GBRSS forked into 382 millisecodns (424 milliseconds per GB).

As you can see a VM running on Xen has a performance hit that is between one order to two orders of magnitude. We believe this is a severe problem with Xen and we hope it will be addressed ASAP.

Latency induced by swapping (operating system paging)

Linux (and many other modern operating systems) is able to relocate memory pages from the memory to the disk, and vice versa, in order to use the system memory efficiently.

If a Redis page is moved by the kernel from the memory to the swap file, when the data stored in this memory page is used by Redis (for example accessing a key stored into this memory page) the kernel will stop the Redis process in order to move the page back into the main memory. This is a slow operation involving random I/Os (compared to accessing a page that is already in memory) and will result into anomalous latency experienced by Redis clients.

The kernel relocates Redis memory pages on disk mainly because of three reasons:

- The system is under memory pressure since the running processes are demanding more physical memory than the amount that is available. The simplest instance of this problem is simply Redis using more memory than the one available.
- The Redis instance data set, or part of the data set, is mostly completely idle (never accessed by clients), so the kernel could swap idle memory pages on disk. This problem is very rare since even a moderately slow instance will touch all the memory pages often, forcing the kernel to retain all the pages in memory.
- Some processes are generating massive read or write I/Os on the system. Because files are generally cached, it tends to put pressure on the kernel to increase the filesystem cache, and therefore generate

swapping activity. Please note it includes Redis RDB and/or AOF background threads which can produce large files.

Fortunately Linux offers good tools to investigate the problem, so the simplest thing to do is when latency due to swapping is suspected is just to check if this is the case.

The first thing to do is to checking the amount of Redis memory that is swapped on disk. In order to do so you need to obtain the Redis instance pid:

```
$ redis-cli info | grep process_id
```

Now enter the /proc file system directory for this process:

```
$ cd /proc/5454
```

Here you'll find a file called **smaps** that describes the memory layout of the Redis process (assuming you are using Linux 2.6.16 or newer). This file contains very detailed information about our process memory maps, and one field called **Swap** is exactly what we are looking for. However there is not just a single swap field since the smaps file contains the different memory maps of our Redis process (The memory layout of a process is more complex than a simple linear array of pages).

Since we are interested in all the memory swapped by our process the first thing to do is to grep for the Swap field across all the file:

```
$ cat smaps | grep 'Swap:'
```

If everything is 0 kb, or if there are sporadic 4k entries, everything is perfectly normal. Actually in our example instance (the one of a real web site running Redis and serving hundreds of users every second) there are a few entries that show more swapped pages. To investigate if this is a serious problem or not we change our command in order to also print the size of the memory map:

```
$ cat smaps | egrep '^(Swap|Size)'
```

As you can see from the output, there is a map of 720896 kB (with just 12 kB swapped) and 156 kb more swapped in another map: basically a very small amount of our memory is swapped so this is not going to create any problem at all.

If instead a non trivial amount of the process memory is swapped on disk your latency problems are likely related to swapping. If this is the case with your Redis instance you can further verify it using the **vmstat** command:

```
$ vmstat 1
```

c

The interesting part of the output for our needs are the two columns **si** and **so**, that counts the amount of memory swapped from/to the swap file. If you see non zero counts in those two columns then there is swapping activity in your system.

Finally, the **iostat** command can be used to check the global I/O activity of the system.

```
$ iostat -xk 1
```

If your latency problem is due to Redis memory being swapped on disk you need to lower the memory pressure in your system, either adding more RAM if Redis is using more memory than the available, or avoiding running other memory hungry processes in the same system.

Latency due to AOF and disk I/O

Another source of latency is due to the Append Only File support on Redis. The AOF basically uses two system calls to accomplish its work. One is write(2) that is used in order to write data to the append only file, and the other one is fdatasync(2) that is used in order to flush the kernel file buffer on disk in order to ensure the durability level specified by the user.

Both the write(2) and fdatasync(2) calls can be source of latency. For instance write(2) can block both when there is a system wide sync in progress, or when the output buffers are full and the kernel requires to flush on disk in order to accept new writes.

The fdatasync(2) call is a worse source of latency as with many combinations of kernels and file systems used it can take from a few milliseconds to a few seconds to complete, especially in the case of some other process doing I/O. For this reason when possible Redis does the fdatasync(2) call in a different thread since Redis 2.4.

We'll see how configuration can affect the amount and source of latency when using the AOF file.

The AOF can be configured to perform an fsync on disk in three different ways using the **appendfsync** configuration option (this setting can be modified at runtime using the **CONFIG SET** command).

- When appendfsync is set to the value of **no** Redis performs no fsync. In this configuration the only source of latency can be write(2). When this happens usually there is no solution since simply the disk can't cope with the speed at which Redis is receiving data, however this is uncommon if the disk is not seriously slowed down by other processes doing I/O.
- When appendfsync is set to the value of **everysec** Redis performs an fsync every second. It uses a different thread, and if the fsync is still in progress Redis uses a buffer to delay the write(2) call up to two seconds (since write would block on Linux if an fsync is in progress against the same file). However if the fsync is taking too long Redis will eventually perform the write(2) call even if the fsync is still in progress, and this can be a source of latency.
- When appendfsync is set to the value of **always** an fsync is performed at every write operation before replying back to the client with an OK code (actually Redis will try to cluster many commands executed at the same time into a single fsync). In this mode performances are very low in general and it is strongly recommended to use a fast disk and a file system implementation that can perform the fsync in short time.

Most Redis users will use either the **no** or **everysec** setting for the appendfsync configuration directive. The suggestion for minimum latency is to avoid other processes doing I/O in the same system. Using an SSD disk can help as well, but usually even non SSD disks perform well with the append only file if the disk is spare as Redis writes to the append only file without performing any seek.

If you want to investigate your latency issues related to the append only file you can use the strace command under Linux:

```
sudo strace -p $(pidof redis-server) -T -e trace=fdatasync
```

The above command will show all the fdatasync(2) system calls performed by Redis in the main thread. With

the above command you'll not see the fdatasync system calls performed by the background thread when the appendfsync config option is set to **everysec**. In order to do so just add the -f switch to strace.

If you wish you can also see both fdatasync and write system calls with the following command:

```
sudo strace -p $(pidof redis-server) -T -e trace=fdatasync,write
```

However since write(2) is also used in order to write data to the client sockets this will likely show too many things unrelated to disk I/O. Apparently there is no way to tell strace to just show slow system calls so I use the following command:

```
sudo strace -f -p $(pidof redis-server) -T -e trace=fdatasync,write 2>&1 | grep -v '0.0' | grep
```

Latency generated by expires

Redis evict expired keys in two ways:

- One *lazy* way expires a key when it is requested by a command, but it is found to be already expired.
- One *active* way expires a few keys every 100 milliseconds.

The active expiring is designed to be adaptive. An expire cycle is started every 100 milliseconds (10 times per second), and will do the following:

- Sample `REDIS_EXPIRELOOKUPS_PER_CRON` keys, evicting all the keys already expired.
- If the more than 25% of the keys were found expired, repeat.

Given that `REDIS_EXPIRELOOKUPS_PER_CRON` is set to 10 by default, and the process is performed ten times per second, usually just 100 keys per second are actively expired. This is enough to clean the DB fast enough even when already expired keys are not accessed for a log time, so that the *lazy* algorithm does not help. At the same time expiring just 100 keys per second has no effects in the latency a Redis instance.

However the algorithm is adaptive and will loop if it founds more than 25% of keys already expired in the set of sampled keys. But given that we run the algorithm ten times per second, this means that the unlucky event of more than 25% of the keys in our random sample are expiring at least *in the same second*.

Basically this means that **if the database contains has many many keys expiring in the same second, and this keys are at least 25% of the current population of keys with an expire set**, Redis can block in order to reach back a percentage of keys already expired that is less than 25%.

This approach is needed in order to avoid using too much memory for keys that area already expired, and usually is absolutely harmless since it's strange that a big number of keys are going to expire in the same exact second, but it is not impossible that the user used EXPIREAT extensively with the same Unix time.

In short: be aware that many keys expiring at the same moment can be a source of latency.

Redis software watchdog

Redis 2.6 introduces the *Redis Software Watchdog* that is a debugging tool designed to track those latency problems that for one reason or the other esacped an analysis using normal tools.

The software watchdog is an experimental feature. While it is designed to be used in production enviroments care should be taken to backup the database before proceeding as it could possibly have unexpected interactions with the normal execution of the Redis server.

It is important to use it only as *last resort* when there is no way to track the issue by other means.

This is how this feature works:

- The user enables the softare watchdog using te `CONFIG SET` command.
- Redis starts monitoring itself constantly.
- If Redis detects that the server is blocked into some operation that is not returning fast enough, and that may be the source of the latency issue, a low level report about where the server is blocked is dumped on the log file.
- The user contacts the developers writing a message in the Redis Google Group, including the watchdog report in the message.

Note that this feature can not be enabled using the redis.conf file, because it is designed to be enabled only in already running instances and only for debugging purposes.

To enable the feature just use the following:

```
CONFIG SET watchdog-period 500
```

The period is specified in milliseconds. In the above example I specified to log latency issues only if the server detects a delay of 500 milliseconds or greater. The minimum configurable period is 200 milliseconds.

When you are done with the software watchdog you can turn it off setting the `watchdog-period` parameter to 0. **Important:** remember to do this because keeping the instance with the watchdog turned on for a longer time than needed is generally not a good idea.

The following is an example of what you'll see printed in the log file once the software watchdog detects a delay longer than the configured one:

```
[8547 | signal handler] (1333114359)
```

Note: in the example the **DEBUG SLEEP** command was used in order to block the server. The stack trace is different if the server blocks in a different context.

If you happen to collect multiple watchdog stack traces you are encouraged to send everything to the Redis Google Group: the more traces we obtain, the simpler it will be to understand what the problem with your instance is.

APPENDIX A: Experimenting with huge pages

Latency introduced by fork can be mitigated using huge pages at the cost of a bigger memory usage during persistence. The following appeindex describe in details this feature as implemented in the Linux kernel.

Some CPUs can use different page size though. AMD and Intel CPUs can support 2 MB page size if needed. These pages are nicknamed *huge pages*. Some operating systems can optimize page size in real time, transparently aggregating small pages into huge pages on the fly.

On Linux, explicit huge pages management has been introduced in 2.6.16, and implicit transparent huge pages are available starting in 2.6.38. If you run recent Linux distributions (for example RH 6 or derivatives), transparent huge pages can be activated, and you can use a vanilla Redis version with them.

This is the preferred way to experiment/use with huge pages on Linux.

Now, if you run older distributions (RH 5, SLES 10-11, or derivatives), and not afraid of a few hacks, Redis requires to be patched in order to support huge pages.

The first step would be to read Mel Gorman's primer on huge pages

There are currently two ways to patch Redis to support huge pages.

- For Redis 2.4, the embedded jemalloc allocator must be patched. patch by Pieter Noordhuis. Note this patch relies on the anonymous mmap huge page support, only available starting 2.6.32, so this method cannot be used for older distributions (RH 5, SLES 10, and derivatives).
- For Redis 2.2, or 2.4 with the libc allocator, Redis makefile must be altered to link Redis with the libhugetlbfs library. It is a straightforward change

Then, the system must be configured to support huge pages.

The following command allocates and makes N huge pages available:

```
$ sudo sysctl -w vm.nr_hugepages=<N>
```

The following command mounts the huge page filesystem:

```
$ sudo mount -t hugetlbfs none /mnt/hugetlbfs
```

In all cases, once Redis is running with huge pages (transparent or not), the following benefits are expected:

- The latency due to the fork operations is dramatically reduced. This is mostly useful for very large instances, and especially on a VM.
- Redis is faster due to the fact the translation look-aside buffer (TLB) of the CPU is more efficient to cache page table entries (i.e. the hit ratio is better). Do not expect miracle, it is only a few percent gain at most.
- Redis memory cannot be swapped out anymore, which is interesting to avoid outstanding latencies due to virtual memory.

Unfortunately, and on top of the extra operational complexity, there is also a significant drawback of running Redis with huge pages. The COW mechanism granularity is the page. With 2 MB pages, the probability a page is modified during a background save operation is 512 times higher than with 4 KB pages. The actual memory required for a background save therefore increases a lot, especially if the write traffic is truly random, with poor locality. With huge pages, using twice the memory while saving is not anymore a theoretical incident. It really happens.

The result of a complete benchmark can be found here.

Redis Mass Insertion

Sometimes Redis instances needs to be loaded with big amount of preexisting or user generated data in a short amount of time, so that million of keys will be created as fast as possible.

This is called a *mass insertion*, and the goal of this document is to provide information about how to feed Redis with data as fast as possible.

Use the protocol, Luke

Using a normal Redis client to perform mass insertion is not a good idea for a few reasons: the naive approach of sending one command after the other is slow because there is to pay the round trip time for every command. It is possible to use pipelining, but for mass insertion of many records you need to write new commands while you read replies at the same time to make sure you are inserting as fast as possible.

Only a small percentage of clients support non-blocking I/O, and not all the clients are able to parse the replies in an efficient way in order to maximize throughput. For all this reasons the preferred way to mass import data into Redis is to generate a text file containing the Redis protocol, in raw format, in order to call the commands needed to insert the required data.

For instance if I need to generate a large data set where there are billions of keys in the form: `keyN -> ValueN' I will create a file containing the following commands in the Redis protocol format:

```
SET Key0 Value0
```

Once this file is created, the remaining action is to feed it to Redis as fast as possible. In the past the way to do this was to use the `netcat` with the following command:

```
(cat data.txt; sleep 10) | nc localhost 6379 > /dev/null
```

However this is not a very reliable way to perform mass import because netcat does not really know when all the data was transferred and can't check for errors. In the unstable branch of Redis at github the `redis-cli` utility supports a new mode called **pipe mode** that was designed in order to perform mass insertion. (This feature will be available in a few days in Redis 2.6-RC4 and in Redis 2.4.14).

Using the pipe mode the command to run looks like the following:

```
cat data.txt | redis-cli --pipe
```

That will produce an output similar to this:

```
All data transferred. Waiting for the last reply...
```

The redis-cli utility will also make sure to only redirect errors received from the Redis instance to the standard output.

Generating Redis Protocol

The Redis protocol is extremely simple to generate and parse, and is Documented here. However in order to generate protocol for the goal of mass insertion you don't need to understand every detail of the protocol, but just that every command is represented in the following way:

```
*<args><cr><lf>
```

Where <cr> means "\r" (or ASCII character 13) and <lf> means "\n" (or ASCII character 10).

For instance the command **SET key value** is represented by the following protocol:

```
*3<cr><lf>
```

Or represented as a quoted string:

```
"*3\r\n$3\r\nSET\r\n$3\r\nkey\r\n$5\r\nvalue\r\n"
```

The file you need to generate for mass insertion is just composed of commands represented in the above way, one after the other.

The following Ruby function generates valid protocol:

```
def gen_redis_proto(*cmd)
```

Using the above function it is possible to easily generate the key value pairs in the above example, with this program:

```
(0...1000).each{|n|
```

We can run the program directly in pipe to redis-cli in order to perform our first mass import session.

```
$ ruby proto.rb | redis-cli --pipe
```

How the pipe mode works under the hoods

The magic needed inside the pipe mode of redis-cli is to be as fast as netcat and still be able to understand when the last reply was sent by the server at the same time.

This is obtained in the following way:

- redis-cli --pipe tries to send data as fast as possible to the server.
- At the same time it reads data when available, trying to parse it.
- Once there is no more data to read from stdin, it sends a special **ECHO** command with a random 20 bytes string: we are sure this is the latest command sent, and we are sure we can match the reply checking if we receive the same 20 bytes as a bulk reply.
- Once this special final command is sent, the code receiving replies starts to match replies with this 20 bytes. When the matching reply is reached it can exit with success.

Using this trick we don't need to parse the protocol we send to the server in order to understand how many commands we are sending, but just the replies.

However while parsing the replies we take a counter of all the replies parsed so that at the end we are able to tell the user the amount of commands transferred to the server by the mass insert session.

This page is a work in progress. Currently it is just a list of things you should check if you have problems with memory.

Special encoding of small aggregate data types

Since Redis 2.2 many data types are optimized to use less space up to a certain size. Hashes, Lists, Sets composed of just integers, and Sorted Sets, when smaller than a given number of elements, and up to a maximum element size, are encoded in a very memory efficient way that uses *up to 10 times less memory* (with 5 time less memory used being the average saving).

This is completely transparent from the point of view of the user and API. Since this is a CPU / memory trade off it is possible to tune the maximum number of elements and maximum element size for special encoded types using the following redis.conf directives.

```
hash-max-zipmap-entries 64 (hahs-max-ziplist-entries for Redis >= 2.6)
```

If a specially encoded value will overflow the configured max size, Redis will automatically convert it into normal encoding. This operation is very fast for small values, but if you change the setting in order to use specially encoded values for much larger aggregate types the suggestion is to run some benchmark and test to check the conversion time.

Using 32 bit instances

Redis compiled with 32 bit target uses a lot less memory per key, since pointers are small, but such an instance will be limited to 4 GB of maximum memory usage. To compile Redis as 32 bit binary use *make 32bit*. RDB and AOF files are compatible between 32 bit and 64 bit instances (and between little and big endian of course) so you can switch from 32 to 64 bit, or the contrary, without problems.

Bit and byte level operations

Redis 2.2 introduced new bit and byte level operations: GETRANGE, SETRANGE, GETBIT and SETBIT. Using this commands you can treat the Redis string type as a random access array. For instance if you have an application where users are identified by an unique progressive integer number, you can use a bitmap in order to save information about sex of users, setting the bit for females and clearing it for males, or the other way around. With 100 millions of users this data will take just 12 megabyte of RAM in a Redis instance. You can do the same using GETRANGE and SETRANGE in order to store one byte of information for user. This is just an example but it is actually possible to model a number of problems in very little space with this new primitives.

Use hashes when possible

Small hashes are encoded in a very small space, so you should try representing your data using hashes every time it is possible. For instance if you have objects representing users in a web application, instead of using different keys for name, surname, email, password, use a single hash with all the required fields.

If you want to know more about this, read the next section.

Using hashes to abstract a very memory efficient plain key-value store on top of Redis

I understand the title of this section is a bit scaring, but I'm going to explain in details what this is about.

Basically it is possible to model a plain key-value store using Redis where values can just be just strings, that is not just more memory efficient than Redis plain keys but also much more memory efficient than memcached.

Let's start with some fact: a few keys use a lot more memory than a single key containing an hash with a few fields. How is this possible? We use a trick. In theory in order to guarantee that we perform lookups in constant time (also known as O(1) in big O notation) there is the need to use a data structure with a constant time complexity in the average case, like an hash table.

But many times hashes contain just a few fields. When hashes are small we can instead just encode them in an O(N) data structure, like a linear array with length-prefixed key value pairs. Since we do this only when N is small, the amortized time for HGET and HSET commands is still O(1): the hash will be converted into a real hash table as soon as the number of elements it contains will grow too much (you can configure the limit in redis.conf).

This does not work well just from the point of view of time complexity, but also from the point of view of constant times, since a linear array of key value pairs happens to play very well with the CPU cache (it has a better cache locality than an hash table).

However since hash fields and values are not (always) represented as full featured Redis objects, hash fields can't have an associated time to live (expire) like a real key, and can only contain a string. But we are okay with this, this was anyway the intention when the hash data type API was designed (we trust simplicity more than features, so nested data structures are not allowed, as expires of single fields are not allowed).

So hashes are memory efficient. This is very useful when using hashes to represent objects or to model other problems when there are group of related fields. But what about if we have a plain key value business?

Imagine we want to use Redis as a cache for many small objects, that can be JSON encoded objects, small HTML fragments, simple key -> boolean values and so forth. Basically anything is a string -> string map with small keys and values.

Now let's assume the objects we want to cache are numbered, like:

- object:102393
- object:1234
- object:5

This is what we can do. Every time there is to perform a SET operation to set a new value, we actually split the key into two parts, one used as a key, and used as field name for the hash. For instance the object named *object:1234" is actually split into:

- a Key named object:12
- a Field named 34

So we use all the characters but the latest two for the key, and the final two characters for the hash field name. To set our key we use the following command:

```
HSET object:12 34 somevalue
```

As you can see every hash will end containing 100 fields, that is an optimal compromise between CPU and memory saved.

There is another very important thing to note, with this schema every hash will have more or less 100 fields regardless of the number of objects we cached. This is since our objects will always end with a number, and not a random string. In some way the final number can be considered as a form of implicit pre-sharding.

What about small numbers? Like object:2? We handle this case using just "object:" as a key name, and the whole number as the hash field name. So object:2 and object:10 will both end inside the key "object:", but one as field name "2" and one as "10".

How much memory we save this way?

I used the following Ruby program to test how this works:

```
require 'rubygems'
```

This is the result against a 64 bit instance of Redis 2.2:

- UseOptimization set to true: 1.7 MB of used memory
- UseOptimization set to false; 11 MB of used memory

This is an order of magnitude, I think this makes Redis more or less the most memory efficient plain key value store out there.

WARNING: for this to work, make sure that in your redis.conf you have something like this:

```
hash-max-zipmap-entries 256
```

Also remember to set the following field accordingly to the maximum size of your keys and values:

```
hash-max-zipmap-value 1024
```

Every time an hash will exceed the number of elements or element size specified it will be converted into a real hash table, and the memory saving will be lost.

You may ask, why don't you do this implicitly in the normal key space so that I don't have to care? There are two reasons: one is that we tend to make trade offs explicit, and this is a clear tradeoff between many things: CPU, memory, max element size. The second is that the top level key space must support a lot of interesting things like expires, LRU data, and so forth so it is not practical to do this in a general way.

But the Redis Way is that the user must understand how things work so that he is able to pick the best compromise, and to understand how the system will behave exactly.

Work in progress

Work in progress... more tips will be added soon.

This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

This page provides a technical description of Redis persistence, it is a suggested read for all the Redis users. For a wider overview of Redis persistence and the durability guarantees it provides you may want to also read Redis persistence demystified.

Redis Persistence

Redis provides a different range of persistence options:

- The RDB persistence performs point-in-time snapshots of your dataset at specified intervals.
- the AOF persistence logs every write operation received by the server, that will be played again at server startup, reconstructing the original dataset. Commands are logged using the same format as the Redis protocol itself, in an append-only fashion. Redis is able to rewrite the log on background when it gets too big.
- If you wish, you can disable persistence at all, if you want your data to just exist as long as the server is running.
- It is possible to combine both AOF and RDB in the same instance. Notice that, in this case, when Redis restarts the AOF file will be used to reconstruct the original dataset since it is guaranteed to be the most complete.

The most important thing to understand is the different trade-offs between the RDB and AOF persistence. Let's start with RDB:

RDB advantages

- RDB is a very compact single-file point-in-time representation of your Redis data. RDB files are perfect for backups. For instance you may want to archive your RDB files every hour for the latest 24 hours, and to save an RDB snapshot every day for 30 days. This allows you to easily restore different versions of the data set in case of disasters.
- RDB is very good for disaster recovery, being a single compact file can be transfered to far data centers, or on Amazon S3 (possibly encrypted).
- RDB maximizes Redis performances since the only work the Redis parent process needs to do in order to persist is forking a child that will do all the rest. The parent instance will never perform disk I/O or alike.
- RDB allows faster restarts with big datasets compared to AOF.

RDB disadvantages

- RDB is NOT good if you need to minimize the chance of data loss in case Redis stops working (for example after a power outage). You can configure different *save points* where an RDB is produced (for instance after at least five minutes and 100 writes against the data set, but you can have multiple save points). However you'll usually create an RDB snapshot every five minutes or more, so in case of Redis stopping working without a correct shutdown for any reason you should be prepared to lose the latest minutes of data.
- RDB needs to fork() often in order to persist on disk using a child process. Fork() can be time consuming if the dataset is big, and may result in Redis to stop serving clients for some millisecond or even for one second if the dataset is very big and the CPU performance not great. AOF also needs to fork() but you can tune how often you want to rewrite your logs without any trade-off on durability.

AOF advantages

- Using AOF Redis is much more durable: you can have different fsync policies: no fsync at all, fsync every second, fsync at every query. With the default policy of fsync every second write performances are still great (fsync is performed using a background thread and the main thread will try hard to perform writes when no fsync is in progress.) but you can only lose one second worth of writes.
- The AOF log is an append only log, so there are no seeks, nor corruption problems if there is a power outage. Even if the log ends with an half-written command for some reason (disk full or other reasons) the redis-check-aof tool is able to fix it easily.
- Redis is able to automatically rewrite the AOF in background when it gets too big. The rewrite is completely safe as while Redis continues appending to the old file, a completely new one is produced with the minimal set of operations needed to create the current data set, and once this second file is ready Redis switches the two and starts appending to the new one.
- AOF contains a log of all the operations one after the other in an easy to understand and parse format. You can even easily export an AOF file. For instance even if you flushed everything for an error using a FLUSHALL command, if no rewrite of the log was performed in the meantime you can still save your data set just stopping the server, removing the latest command, and restarting Redis again.

AOF disadvantages

- AOF files are usually bigger than the equivalent RDB files for the same dataset.
- AOF can be slower then RDB depending on the exact fsync policy. In general with fsync set to *every second* performances are still very high, and with fsync disabled it should be exactly as fast as RDB even under high load. Still RDB is able to provide more guarantees about the maximum latency even in the case of an huge write load.
- In the past we experienced rare bugs in specific commands (for instance there was one involving blocking commands like BRPOPLPUSH) causing the AOF produced to don't reproduce exactly the same dataset on reloading. This bugs are rare and we have tests in the test suite creating random complex datasets automatically and reloading them to check everything is ok, but this kind of bugs are almost impossible with RDB persistence. To make this point more clear: the Redis AOF works incrementally updating an existing state, like MySQL or MongoDB does, while the RDB snapshotting creates everything from scratch again and again, that is conceptually more robust. However 1) It should be noted that every time the AOF is rewritten by Redis it is recreated from scratch starting from the actual data contained in the data set, making resistance to bugs stronger compared to an always appending AOF file (or one rewritten reading the old AOF instead of reading the data in memory). 2) We never had a single report from users about an AOF corruption that was detected in the real world.

Ok, so what should I use?

The general indication is that you should use both persistence methods if you want a degree of data safety comparable to what PostgreSQL can provide you.

If you care a lot about your data, but still can live with a few minutes of data lose in case of disasters, you can simply use RDB alone.

There are many users using AOF alone, but we discourage it since to have an RDB snapshot from time to time is a great idea for doing database backups, for faster restarts, and in the event of bugs in the AOF engine.

Note: for all these reasons we'll likely end up unifying AOF and RDB into a single persistence model in the future (long term plan).

The following sections will illustrate a few more details about the two persistence models.

Snapshotting

By default Redis saves snapshots of the dataset on disk, in a binary file called `dump.rdb`. You can configure Redis to have it save the dataset every N seconds if there are at least M changes in the dataset, or you can manually call the SAVE or BGSAVE commands.

For example, this configuration will make Redis automatically dump the dataset to disk every 60 seconds if at least 1000 keys changed:

```
save 60 1000
```

This strategy is known as *snapshotting*.

How it works

Whenever Redis needs to dump the dataset to disk, this is what happens:

- Redis forks. We now have a child and a parent process.
- The child starts to write the dataset to a temporary RDB file.
- When the child is done writing the new RDB file, it replaces the old one.

This method allows Redis to benefit from copy-on-write semantics.

Append-only file

Snapshotting is not very durable. If your computer running Redis stops, your power line fails, or you accidentally `kill -9` your instance, the latest data written on Redis will get lost. While this may not be a big deal for some applications, there are use cases for full durability, and in these cases Redis was not a viable option.

The *append-only file* is an alternative, fully-durable strategy for Redis. It became available in version 1.1.

You can turn on the AOF in your configuration file:

```
appendonly yes
```

From now on, every time Redis receives a command that changes the dataset (e.g. SET) it will append it to the AOF. When you restart Redis it will re-play the AOF to rebuild the state.

Log rewriting

As you can guess, the AOF gets bigger and bigger as write operations are performed. For example, if you are incrementing a counter 100 times, you'll end up with a single key in your dataset containing the final value,

but 100 entries in your AOF. 99 of those entries are not needed to rebuild the current state.

So Redis supports an interesting feature: it is able to rebuild the AOF in the background without interrupting service to clients. Whenever you issue a BGREWRITEAOF Redis will write the shortest sequence of commands needed to rebuild the current dataset in memory. If you're using the AOF with Redis 2.2 you'll need to run BGREWRITEAOF from time to time. Redis 2.4 is able to trigger log rewriting automatically (see the 2.4 example configuration file for more information).

How durable is the append only file?

You can configure how many times Redis will `fsync` data on disk. There are three options:

- `fsync` every time a new command is appended to the AOF. Very very slow, very safe.
- `fsync` every second. Fast enough (in 2.4 likely to be as fast as snapshotting), and you can lose 1 second of data if there is a disaster.
- Never `fsync`, just put your data in the hands of the Operating System. The faster and less safe method.

The suggested (and default) policy is to `fsync` every second. It is both very fast and pretty safe. The `always` policy is very slow in practice (although it was improved in Redis 2.0) â there is no way to make `fsync` faster than it is.

What should I do if my AOF gets corrupted?

It is possible that the server crashes while writing the AOF file (this still should never lead to inconsistencies), corrupting the file in a way that is no longer loadable by Redis. When this happens you can fix this problem using the following procedure:

- Make a backup copy of your AOF file.
- Fix the original file using the `redis-check-aof` tool that ships with Redis:

  ```
  $ redis-check-aof --fix
  ```
- Optionally use `diff -u` to check what is the difference between two files.
- Restart the server with the fixed file.

How it works

Log rewriting uses the same copy-on-write trick already in use for snapshotting. This is how it works:

- Redis forks, so now we have a child and a parent process.
- The child starts writing the new AOF in a temporary file.
- The parent accumulates all the new changes in an in-memory buffer (but at the same time it writes the new changes in the old append-only file, so if the rewriting fails, we are safe).
- When the child is done rewriting the file, the parent gets a signal, and appends the in-memory buffer at the end of the file generated by the child.
- Profit! Now Redis atomically renames the old file into the new one, and starts appending new data into the new file.

How I can switch to AOF, if I'm currently using dump.rdb snapshots?

There is a different procedure to do this in Redis 2.0 and Redis 2.2, as you can guess it's simpler in Redis 2.2 and does not require a restart at all.

Redis >= 2.2

- Make a backup of your latest dump.rdb file.
- Transfer this backup into a safe place.
- Issue the following two commands:
- redis-cli config set appendonly yes
- redis-cli config set save ""
- Make sure that your database contains the same number of keys it contained.
- Make sure that writes are appended to the append only file correctly.

The first CONFIG command enables the Append Only File. In order to do so **Redis will block** to generate the initial dump, then will open the file for writing, and will start appending all the next write queries.

The second CONFIG command is used to turn off snapshotting persistence. This is optional, if you wish you can take both the persistence methods enabled.

IMPORTANT: remember to edit your redis.conf to turn on the AOF, otherwise when you restart the server the configuration changes will be lost and the server will start again with the old configuration.

Redis 2.0

- Make a backup of your latest dump.rdb file.
- Transfer this backup into a safe place.
- Stop all the writes against the database!
- Issue a redis-cli bgrewriteaof. This will create the append only file.
- Stop the server when Redis finished generating the AOF dump.
- Edit redis.conf end enable append only file persistence.
- Restart the server.
- Make sure that your database contains the same number of keys it contained.
- Make sure that writes are appended to the append only file correctly.

Interactions between AOF and RDB persistence

Redis >= 2.4 makes sure to avoid triggering an AOF rewrite when an RDB snapshotting operation is already in progress, or allowing a BGSAVE while the AOF rewrite is in progress. This prevents two Redis background processes from doing heavy disk I/O at the same time.

When snapshotting is in progress and the user explicitly requests a log rewrite operation using BGREWRITEAOF the server will reply with an OK status code telling the user the operation is scheduled, and the rewrite will start once the snapshotting is completed.

In the case both AOF and RDB persistence are enabled and Redis restarts the AOF file will be used to reconstruct the original dataset since it is guaranteed to be the most complete.

Backing up Redis data

Before starting this section, make sure to read the following sentence: **Make Sure to Backup Your Database**. Disks break, instances in the cloud disappear, and so forth: no backups means huge risk of data disappearing into /dev/null.

Redis is very data backup friendly since you can copy RDB files while the database is running: the RDB is never modified once produced, and while it gets produced it uses a temporary name and is renamed into its final destination atomically using rename(2) only when the new snapshot is complete.

This means that copying the RDB file is completely safe while the server is running. This is what we suggest:

- Create a cron job in your server creating hourly snapshots of the RDB file in one directory, and daily snapshots in a different directory.
- Every time the cron script runs, make sure to call the `find` command to make sure too old snapshots are deleted: for instance you can take hourly snapshots for the latest 48 hours, and daily snapshots for one or two months. Make sure to name the snapshots with data and time information.
- At least one time every day make sure to transfer an RDB snapshot *outside your data center* or at least *outside the physical machine* running your Redis instance.

Disaster recovery

Disaster recovery in the context of Redis is basically the same story as backups, plus the ability to transfer those backups in many different external data centers. This way data is secured even in the case of some catastrophic event affecting the main data center where Redis is running and producing its snapshots.

Since many Redis users are in the startup scene and thus don't have plenty of money to spend we'll review the most interesting disaster recovery techniques that don't have too high costs.

- Amazon S3 and other similar services are a good way for mounting your disaster recovery system. Simply transfer your daily or hourly RDB snapshot to S3 in an encrypted form. You can encrypt your data using gpg -c (in symmetric encryption mode). Make sure to store your password in many differnet safe places (for instance give a copy to the most important guys of your organization). It is recommanded to use multiple storage services for improved data safety.
- Transfer your snapshots using scp (part of ssh) to far servers. This is a fairly simple and safe route: get a small VPS in a place that is very far from you, install ssh there, and greate an ssh client key without passphrase, then make add it in the authorized_keys file of your small VPS. You are ready to transfer backups in an automated fashion. Get at least two VPS in two different providers for best results.

It is important to understand that this systems can easily fail if not coded in the right way. At least make absolutely sure that after the transfer is completed you are able to verify the file size (that should match the one of the file you copied) and possibly the SHA1 digest if you are using a VPS.

You also need some kind of independent alert system if the transfer of fresh backups is not working for some reason.

Request/Response protocols and RTT

Redis is a TCP server using the client-server model and what is called a *Request/Response* protocol.

This means that usually a request is accomplished with the following steps:

- The client sends a query to the server, and reads from the socket, usually in a blocking way, for the server response.
- The server processes the command and sends the response back to the client.

So for instance a four commands sequence is something like this:

- *Client:* INCR X
- *Server:* 1
- *Client:* INCR X
- *Server:* 2
- *Client:* INCR X
- *Server:* 3
- *Client:* INCR X
- *Server:* 4

Clients and Servers are connected via a networking link. Such a link can be very fast (a loopback interface) or very slow (a connection established over the Internet with many hops between the two hosts). Whatever the network latency is, there is a time for the packets to travel from the client to the server, and back from the server to the client to carry the reply.

This time is called RTT (Round Trip Time). It is very easy to see how this can affect the performances when a client needs to perform many requests in a row (for instance adding many elements to the same list, or populating a database with many keys). For instance if the RTT time is 250 milliseconds (in the case of a very slow link over the Internet), even if the server is able to process 100k requests per second, we'll be able to process at max four requests per second.

If the interface used is a loopback interface, the RTT is much shorter (for instance my host reports 0,044 milliseconds pinging 127.0.0.1), but it is still a lot if you need to perform many writes in a row.

Fortunately there is a way to improve this use cases.

Redis Pipelining

A Request/Response server can be implemented so that it is able to process new requests even if the client didn't already read the old responses. This way it is possible to send *multiple commands* to the server without waiting for the replies at all, and finally read the replies in a single step.

This is called pipelining, and is a technique widely in use since many decades. For instance many POP3 protocol implementations already supported this feature, dramatically speeding up the process of downloading new emails from the server.

Redis supports pipelining since the very early days, so whatever version you are running, you can use pipelining with Redis. This is an example using the raw netcat utility:

```
$ (echo -en "PING\r\nPING\r\nPING\r\n"; sleep 1) | nc localhost 6379
```

This time we are not paying the cost of RTT for every call, but just one time for the three commands.

To be very explicit, with pipelining the order of operations of our very first example will be the following:

- *Client:* INCR X
- *Client:* INCR X
- *Client:* INCR X
- *Client:* INCR X
- *Server:* 1
- *Server:* 2
- *Server:* 3
- *Server:* 4

IMPORTANT NOTE: while the client sends commands using pipelining, the server will be forced to queue the replies, using memory. So if you need to send many many commands with pipelining it's better to send this commands up to a given reasonable number, for instance 10k commands, read the replies, and send again other 10k commands and so forth. The speed will be nearly the same, but the additional memory used will be at max the amount needed to queue the replies for this 10k commands.

Some benchmark

In the following benchmark we'll use the Redis Ruby client, supporting pipelining, to test the speed improvement due to pipelining:

```
require 'rubygems'
```

Running the above simple script will provide this figures in my Mac OS X system, running over the loopback interface, where pipelining will provide the smallest improvement as the RTT is already pretty low:

```
without pipelining 1.185238 seconds
```

As you can see using pipelining we improved the transfer by a factor of five.

Pipelining VS Scripting

Using Redis scripting (available in Redis version 2.6 or greater) a number of use cases for pipelining can be addressed more efficiently using scripts that perform a lot of the work needed server side. A big advantage of scripting is that it is able to both read and write data with minimal latency, making operations like *read, compute, write* very fast (pipelining can't help in this scenario since the client needs the reply of the read command before it can call the write command).

Sometimes the application may also want to send EVAL or EVALSHA commands in a pipeline. This is entirely possible and Redis explicitly supports it with the SCRIPT LOAD command (it guarantees that EVALSHA can be called without the risk of failing).

This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by 

Problems with Redis? This is a good starting point.

This page tries to help you about what to do if you have issues with Redis. Part of the Redis project is helping people that are experiencing problems because we don't like to let people alone with their issues.

- If you have **latency problems** with Redis, that in some way appears to be idle for some time, read our Redis latency trubleshooting guide.
- Redis stable releases are usually very reliable, however in the rare event you are **experiencing crashes** the developers can help a lot more if you provide debugging informations. Please read our Debugging Redis guide.
- It happened multiple times that users experiencing problems with Redis actually had a server with **broken RAM**. Please test your RAM using **redis-server --test-memory** in case Redis is not stable in your system. Redis built-in memory test is fast and reasonably reliable, but if you can you should reboot your server and use memtest86.

For every other problem please drop a message to the Redis Google Group. We will be glad to help.

List of known critical bugs in previous Redis releases.

Note: this list may not be complete as we staretd it March 30, 2012, and did not included much historical data.

- Redis version up to 2.4.12 and 2.6.0-RC1: KEYS may not list all the keys, or may list duplicated keys, if keys with an expire set are present in the database. Issue #487.
- Redis version up to 2.4.10: SORT using GET or BY option with keys with an expire set may crash the server. Issue #460.
- Redis version up to 2.4.10: a bug in the aeWait() implementation in ae.c may result in a server crash under extremely hard to replicate conditions. Issue #267.
- Redis version up to 2.4.9: **memory leak in replication**. A memory leak was triggered by replicating a master contaning a database ID greatear than ID 9.
- Redis version up to 2.4.9: **chained replication bug**. In environments where a slave B is attached to another instance `A`, and the instance `A` is switched between master and slave using the SLAVEOF command, it is possilbe that `B` will not be correctly disconnected to force a resync when `A` changes status (and data set content).
- Redis version up to 2.4.7: **redis-check-aof does not work properly in 32 bit instances with AOF files bigger than 2GB**.
- Redis version up to 2.4.7: **Mixing replication and maxmemory produced bad results**. Specifically a master with maxmemory set with attached slaves could result into the master blocking and the dataset on the master to get completely erased. The reason was that key expiring produced more memory usabe because of the replication link DEL synthesizing, triggering the expiring of more keys.
- Redis versions up to 2.4.5: **Connection of multiple slaves at the same time could result into big master memory usage, and slave desync**. (See issue 141 for more details).

List of known bugs still present in latest 2.4 release.

- Redis version up to the current 2.4.x release: **Variadic list push commands and blocking list operations will not play well**. If you use LPUSH or RPUSH commands against a key that has other clients waiting for elements with blocking operations such as BLPOP, both the results of the computation the replication on slaves, and the AOF file commands produced, may not be correct. This bug is fixed in Redis 2.6 but unfortunately a too big refactoring was needed to fix the bug, large enough to make a back port more problematic than the bug itself.

List of known bugs still present in latest 2.6 release.

- There are no known important bugs in Redis 2.6.x

List of known Linux related bugs affecting Redis.

- Ubuntu 10.04 and 10.10 have serious bugs (especially 10.10) that cause slow downs if not just instance hangs. Please move away from the default kernels shipped with this distributions. Link to 10.04 bug. Link to 10.10 bug. Both bugs were reported many times in the context of EC2 instances, but other users confirmed that also native servers are affected (at least by one of the two).
- Certain versions of the Xen hypervisor are known to have very bad fork() performances. See the latency page for more information.

This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

Protocol specification

The Redis protocol is a compromise between the following things:

- Simple to implement
- Fast to parse by a computer
- Easy enough to parse by a human

Networking layer

A client connects to a Redis server creating a TCP connection to the port 6379. Every Redis command or data transmitted by the client and the server is terminated by `\r\n` (CRLF).

Requests

Redis accepts commands composed of different arguments. Once a command is received, it is processed and a reply is sent back to the client.

The new unified request protocol

The new unified protocol was introduced in Redis 1.2, but it became the standard way for talking with the Redis server in Redis 2.0.

In the unified protocol all the arguments sent to the Redis server are binary safe. This is the general form:

```
*<number of arguments> CR LF
```

See the following example:

```
*3
```

This is how the above command looks as a quoted string, so that it is possible to see the exact value of every byte in the query:

```
"*3\r\n$3\r\nSET\r\n$5\r\nmykey\r\n$7\r\nmyvalue\r\n"
```

As you will see in a moment this format is also used in Redis replies. The format used for every argument `$6\r\nmydata\r\n` is called a Bulk Reply. While the actual unified request protocol is what Redis uses to return list of items, and is called a Multi-bulk reply. It is just the sum of N different Bulk Replies prefixed by a `*<argc>\r\n` string where `<argc>` is the number of arguments (Bulk Replies) that will follow.

Replies

Redis will reply to commands with different kinds of replies. It is possible to check the kind of reply from the first byte sent by the server:

- With a single line reply the first byte of the reply will be "+"

- With an error message the first byte of the reply will be "-"
- With an integer number the first byte of the reply will be ":"
- With bulk reply the first byte of the reply will be "$"
- With multi-bulk reply the first byte of the reply will be "*"

Status reply

A status reply (or: single line reply) is in the form of a single line string starting with "+" terminated by "\r\n". For example:

```
+OK
```

The client library should return everything after the "+", that is, the string "OK" in this example.

Error reply

Errors are sent similar to status replies. The only difference is that the first byte is "-" instead of "+".

Error replies are only sent when something strange happened, for instance if you try to perform an operation against the wrong data type, or if the command does not exist and so forth. So an exception should be raised by the library client when an Error Reply is received.

Integer reply

This type of reply is just a CRLF terminated string representing an integer, prefixed by a ":" byte. For example ":0\r\n", or ":1000\r\n" are integer replies.

With commands like INCR or LASTSAVE using the integer reply to actually return a value there is no special meaning for the returned integer. It is just an incremental number for INCR, a UNIX time for LASTSAVE and so on.

Some commands like EXISTS will return 1 for true and 0 for false.

Other commands like SADD, SREM and SETNX will return 1 if the operation was actually done, 0 otherwise.

The following commands will reply with an integer reply: SETNX, DEL, EXISTS, INCR, INCRBY, DECR, DECRBY, DBSIZE, LASTSAVE, RENAMENX, MOVE, LLEN, SADD, SREM, SISMEMBER, SCARD

Bulk replies

Bulk replies are used by the server in order to return a single binary safe string.

```
C: GET mykey
```

The server sends as the first line a "$" byte followed by the number of bytes of the actual reply, followed by CRLF, then the actual data bytes are sent, followed by additional two bytes for the final CRLF. The exact sequence sent by the server is:

```
"$6\r\nfoobar\r\n"
```

If the requested value does not exist the bulk reply will use the special value -1 as data length, example:

```
C: GET nonexistingkey
```

The client library API should not return an empty string, but a nil object, when the requested object does not exist. For example a Ruby library should return 'nil' while a C library should return NULL (or set a special flag in the reply object), and so forth.

Multi-bulk replies

Commands like LRANGE need to return multiple values (every element of the list is a value, and LRANGE needs to return more than a single element). This is accomplished using multiple bulk writes, prefixed by an initial line indicating how many bulk writes will follow. The first byte of a multi bulk reply is always `*`. Example:

```
C: LRANGE mylist 0 3
```

As you can see the multi bulk reply is exactly the same format used in order to send commands to the Redis server using the unified protocol.

The first line the server sent is `*4\r\n` in order to specify that four bulk replies will follow. Then every bulk write is transmitted.

If the specified key does not exist, the key is considered to hold an empty list and the value `0` is sent as multi bulk count. Example:

```
C: LRANGE nokey 0 1
```

When the BLPOP command times out, it returns the nil multi bulk reply. This type of multi bulk has count `-1` and should be interpreted as a nil value. Example:

```
C: BLPOP key 1
```

A client library API *SHOULD* return a nil object and not an empty list when this happens. This is necessary to distinguish between an empty list and an error condition (for instance the timeout condition of the BLPOP command).

Nil elements in Multi-Bulk replies

Single elements of a multi bulk reply may have -1 length, in order to signal that this elements are missing and not empty strings. This can happen with the SORT command when used with the GET *pattern* option when the specified key is missing. Example of a multi bulk reply containing an empty element:

```
S: *3
```

The second element is nul. The client library should return something like this:

```
["foo",nil,"bar"]
```

Multiple commands and pipelining

A client can use the same connection in order to issue multiple commands. Pipelining is supported so multiple commands can be sent with a single write operation by the client, it is not needed to read the server reply in order to issue the next command. All the replies can be read at the end.

Usually Redis server and client will have a very fast link so this is not very important to support this feature in a client implementation, still if an application needs to issue a very large number of commands in short time to use pipelining can be much faster.

The old protocol for sending commands

Before of the Unified Request Protocol Redis used a different protocol to send commands, that is still supported since it is simpler to type by hand via telnet. In this protocol there are two kind of commands:

- Inline commands: simple commands where arguments are just space separated strings. No binary safeness is possible.
- Bulk commands: bulk commands are exactly like inline commands, but the last argument is handled in a special way in order to allow for a binary-safe last argument.

Inline Commands

The simplest way to send Redis a command is via **inline commands**. The following is an example of a server/client chat using an inline command (the server chat starts with S:, the client chat with C:)

```
C: PING
```

The following is another example of an INLINE command returning an integer:

```
C: EXISTS somekey
```

Since 'somekey' does not exist the server returned ':0'.

Note that the EXISTS command takes one argument. Arguments are separated by spaces.

Bulk commands

Some commands when sent as inline commands require a special form in order to support a binary safe last argument. This commands will use the last argument for a "byte count", then the bulk data is sent (that can be binary safe since the server knows how many bytes to read).

See for instance the following example:

```
C: SET mykey 6
```

The last argument of the command is '6'. This specify the number of DATA bytes that will follow, that is, the string "foobar". Note that even this bytes are terminated by two additional bytes of CRLF.

All the bulk commands are in this exact form: instead of the last argument the number of bytes that will follow is specified, followed by the bytes composing the argument itself, and CRLF. In order to be more clear for the programmer this is the string sent by the client in the above sample:

```
"SET mykey 6\r\nfoobar\r\n"
```

Redis has an internal list of what command is inline and what command is bulk, so you have to send this commands accordingly. It is strongly suggested to use the new Unified Request Protocol instead.

Related commands

- PSUBSCRIBE
- PUBLISH
- PUNSUBSCRIBE
- SUBSCRIBE
- UNSUBSCRIBE

Pub/Sub

SUBSCRIBE, UNSUBSCRIBE and PUBLISH implement the Publish/Subscribe messaging paradigm where (citing Wikipedia) 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. This decoupling of publishers and subscribers can allow for greater scalability and a more dynamic network topology.

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. The reply of the SUBSCRIBE and UNSUBSCRIBE operations are sent in the form of messages, so that the client can just read a coherent stream of messages where the first element indicates the type of message.

Format of pushed messages

A message is a Multi-bulk reply with three elements.

The first element is the kind of message:

- `subscribe`: means that we successfully subscribed to the channel given as the second element in the reply. The third argument represents the number of channels we are currently subscribed to.
- `unsubscribe`: means that we successfully unsubscribed from the channel given as second element in the reply. The third argument represents the number of channels we are currently subscribed to. When the last argument is zero, we are no longer subscribed to any channel, and the client can issue any kind of Redis command as we are outside the Pub/Sub state.
- `message`: it is a message received as result of a PUBLISH command issued by another client. The second element is the name of the originating channel, and the third argument is the actual message payload.

Wire protocol example

```
SUBSCRIBE first second
```

At this point, from another client we issue a PUBLISH operation against the channel named `second`:

```
> PUBLISH second Hello
```

This is what the first client receives:

```
*3
```

Now the client unsubscribes itself from all the channels using the UNSUBSCRIBE command without additional arguments:

```
UNSUBSCRIBE
```

Pattern-matching subscriptions

The Redis Pub/Sub implementation supports pattern matching. Clients may subscribe to glob-style patterns in order to receive all the messages sent to channel names matching a given pattern.

For instance:

```
PSUBSCRIBE news.*
```

Will receive all the messages sent to the channel `news.art.figurative`, `news.music.jazz`, etc. All the glob-style patterns are valid, so multiple wildcards are supported.

```
PUNSUBSCRIBE news.*
```

Will then unsubscribe the client from that pattern. No other subscriptions will be affected by this call.

Messages received as a result of pattern matching are sent in a different format:

- The type of the message is `pmessage`: it is a message received as result of a PUBLISH command issued by another client, matching a pattern-matching subscription. The second element is the original pattern matched, the third element is the name of the originating channel, and the last element the actual message payload.

Similarly to SUBSCRIBE and UNSUBSCRIBE, PSUBSCRIBE and PUNSUBSCRIBE commands are acknowledged by the system sending a message of type `psubscribe` and `punsubscribe` using the same format as the `subscribe` and `unsubscribe` message format.

Messages matching both a pattern and a channel subscription

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
```

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`.

The meaning of the subscription count with pattern matching

In `subscribe`, `unsubscribe`, `psubscribe` and `punsubscribe` message types, the last argument is the count of subscriptions still active. This number is actually the total number of channels and patterns the client is still subscribed to. So the client will exit the Pub/Sub state only when this count drops to zero as a result of unsubscription from all the channels and patterns.

Programming example

Pieter Noordhuis provided a great example using EventMachine and Redis to create a multi user high performance web chat.

Client library implementation hints

Because all the messages received contain the original subscription causing the message delivery (the channel in the case of message type, and the original pattern in the case of pmessage type) client libraries may bind the original subscription to callbacks (that can be anonymous functions, blocks, function pointers), using an hash table.

When a message is received an O(1) lookup can be done in order to deliver the message to the registered callback.

Replication

Redis replication is a very simple to use and configure master-slave replication that allows slave Redis servers to be exact copies of master servers. The following are some very important facts about Redis replication:

- A master can have multiple slaves.
- Slaves are able to accept other slaves connections. Aside from connecting a number of slaves to the same master, slaves can also be connected to other slaves in a graph-like structure.
- Redis replication is non-blocking on the master side, this means that the master will continue to serve queries when one or more slaves perform the first synchronization.
- Replication is non blocking on the slave side: while the slave is performing the first synchronization it can reply to queries using the old version of the data set, assuming you configured Redis to do so in redis.conf. Otherwise you can configure Redis slaves to send clients an error if the link with the master is down. However there is a moment where the old dataset must be deleted and the new one must be loaded by the slave where it will block incoming connections.
- Replications can be used both for scalability, in order to have multiple slaves for read-only queries (for example, heavy SORT operations can be offloaded to slaves, or simply for data redundancy.
- It is possible to use replication to avoid the saving process on the master side: just configure your master redis.conf to avoid saving (just comment all the "save" directives), then connect a slave configured to save from time to time.

How Redis replication works

If you set up a slave, upon connection it sends a SYNC command. And it doesn't matter if it's the first time it has connected or if it's a reconnection.

The master then starts background saving, and collects all new commands received that will modify the dataset. When the background saving is complete, the master transfers the database file to the slave, which saves it on disk, and then loads it into memory. The master will then send to the slave all accumulated commands, and all new commands received from clients that will modify the dataset. This is done as a stream of commands and is in the same format of the Redis protocol itself.

You can try it yourself via telnet. Connect to the Redis port while the server is doing some work and issue the SYNC command. You'll see a bulk transfer and then every command received by the master will be re-issued in the telnet session.

Slaves are able to automatically reconnect when the master <-> slave link goes down for some reason. If the master receives multiple concurrent slave synchronization requests, it performs a single background save in order to serve all of them.

When a master and a slave reconnects after the link went down, a full resync is performed.

Configuration

To configure replication is trivial: just add the following line to the slave configuration file:

```
slaveof 192.168.1.1 6379
```

Of course you need to replace 192.168.1.1 6379 with your master IP address (or hostname) and port. Alternatively, you can call the SLAVEOF command and the master host will start a sync with the slave.

Read only slave

Since Redis 2.6 slaves support a read-only mode that is enabled by default. This behavior is controlled by the `slave-read-only` option in the redis.conf file, and can be enabled and disabled at runtime using `CONFIG SET`.

Read only slaves will reject all the write commands, so that it is not possible to write to a slave because of a mistake. This does not mean that the feature is conceived to expose a slave instance to the internet or more generally to a network where untrusted clients exist, because administrative commands like `DEBUG` or `CONFIG` are still enabled. However security of read-only instances can be improved disabling commands in redis.conf using the `rename-command` directive.

You may wonder why it is possible to revert the default and have slave instances that can be target of write operations. The reason is that while this writes will be discarded if the slave and the master will resynchronize, or if the slave is restarted, often there is ephemeral data that is unimportant that can be stored into slaves. For instance clients may take information about reachability of master in the slave instance to coordinate a fail over strategy.

Setting a slave to authenticate to a master

If your master has a password via `requirepass`, it's trivial to configure the slave to use that password in all sync operations.

To do it on a running instance, use `redis-cli` and type:

```
config set masterauth <password>
```

To set it permanently, add this to your config file:

```
masterauth <password>
```

Redis Security

This document provides an introduction to the topic of security from the point of view of Redis: the access control provided by Redis, code security concerns, attacks that can be triggered from the outside by selecting malicious inputs and other similar topics are covered.

Redis general security model

Redis is designed to be accessed by trusted clients inside trusted environments. This means that usually it is not a good idea to expose the Redis instance directly to the internet or, in general, to an environment where untrusted clients can directly access the Redis TCP port or UNIX socket.

For instance, in the common context of a web application implemented using Redis as a database, cache, or messaging system, the clients inside the front-end (web side) of the application will query Redis to generate pages or to perform operations requested or triggered by the web application user.

In this case, the web application mediates access between Redis and untrusted clients (the user browsers accessing the web application).

This is a specific example, but, in general, untrusted access to Redis should always be mediated by a layer implementing ACLs, validating user input, and deciding what operations to perform against the Redis instance.

In general, Redis is not optimized for maximum security but for maximum performance and simplicity.

Network security

Access to the Redis port should be denied to everybody but trusted clients in the network, so the servers running Redis should be directly accessible only by the computers implementing the application using Redis.

In the common case of a single computer directly exposed to the internet, such as a virtualized Linux instance (Linode, EC2, ...), the Redis port should be firewalled to prevent access from the outside. Clients will still be able to access Redis using the loopback interface.

Note that it is possible to bind Redis to a single interface by adding a line like the following to the **redis.conf** file:

```
bind 127.0.0.1
```

Failing to protect the Redis port from the outside can have a big security impact because of the nature of Redis. For instance, a single **FLUSHALL** command can be used by an external attacker to delete the whole data set.

Authentication feature

While Redis does not try to implement Access Control, it provides a tiny layer of authentication that is optionally turned on editing the **redis.conf** file.

When the authorization layer is enabled, Redis will refuse any query by unauthenticated clients. A client can authenticate itself by sending the **AUTH** command followed by the password.

The password is set by the system administrator in clear text inside the redis.conf file. It should be long enough to prevent brute force attacks for two reasons:

- Redis is very fast at serving queries. Many passwords per second can be tested by an external client.
- The Redis password is stored inside the **redis.conf** file and inside the client configuration, so it does not need to be remembered by the system administrator, and thus it can be very long.

The goal of the authentication layer is to optionally provide a layer of redundancy. If firewalling or any other system implemented to protect Redis from external attackers fail, an external client will still not be able to access the Redis instance without knowledge of the authentication password.

The AUTH command, like every other Redis command, is sent unencrypted, so it does not protect against an attacker that has enough access to the network to perform eavesdropping.

Data encryption support

Redis does not support encryption. In order to implement setups where trusted parties can access a Redis instance over the internet or other untrusted networks, an additional layer of protection should be implemented, such as an SSL proxy.

Disabling of specific commands

It is possible to disable commands in Redis or to rename them into an unguessable name, so that normal clients are limited to a specified set of commands.

For instance, a virtualized server provider may offer a managed Redis instance service. In this context, normal users should probably not be able to call the Redis **CONFIG** command to alter the configuration of the instance, but the systems that provide and remove instances should be able to do so.

In this case, it is possible to either rename or completely shadow commands from the command table. This feature is available as a statement that can be used inside the redis.conf configuration file. For example:

```
rename-command CONFIG b840fc02d524045429941cc15f59e41cb7be6c52
```

In the above example, the **CONFIG** command was renamed into an unguessable name. It is also possible to completely disable it (or any other command) by renaming it to the empty string, like in the following example:

```
rename-command CONFIG ""
```

Attacks triggered by carefully selected inputs from external clients

There is a class of attacks that an attacker can trigger from the outside even without external access to the instance. An example of such attacks are the ability to insert data into Redis that triggers pathological (worst case) algorithm complexity on data structures implemented inside Redis internals.

For instance an attacker could supply, via a web form, a set of strings that is known to hash to the same bucket into an hash table in order to turn the O(1) expected time (the average time) to the O(N) worst case, consuming more CPU than expected, and ultimately causing a Denial of Service.

To prevent this specific attack, Redis uses a per-execution pseudo-random seed to the hash function.

Redis implements the SORT command using the qsort algorithm. Currently, the algorithm is not randomized, so it is possible to trigger a quadratic worst-case behavior by carefully selecting the right set of inputs.

String escaping and NoSQL injection

The Redis protocol has no concept of string escaping, so injection is impossible under normal circumstances using a normal client library. The protocol uses prefixed-length strings and is completely binary safe.

Lua scripts executed by the **EVAL** and **EVALSHA** commands follow the same rules, and thus those commands are also safe.

While it would be a very strange use case, the application should avoid composing the body of the Lua script using strings obtained from untrusted sources.

Code security

In a classical Redis setup, clients are allowed full access to the command set, but accessing the instance should never result in the ability to control the system where Redis is running.

Internally, Redis uses all the well known practices for writing secure code, to prevent buffer overflows, format bugs and other memory corruption issues. However, the ability to control the server configuration using the **CONFIG** command makes the client able to change the working dir of the program and the name of the dump file. This allows clients to write RDB Redis files at random paths, that is a security issue that may easily lead to the ability to run untrusted code as the same user as Redis is running.

Redis does not requires root privileges to run. It is recommended to run it as an unprivileged *redis* user that is only used for this purpose. The Redis authors are currently investigating the possibility of adding a new configuration parameter to prevent **CONFIG SET/GET dir** and other similar run-time configuration directives. This would prevent clients from forcing the server to write Redis dump files at arbitrary locations.

Redis Sentinel Documentation

Redis Sentinel is a system designed to help managing Redis instances. It performs the following three tasks:

- **Monitoring**. Sentinel constantly check if your master and slave instances are working as expected.
- **Notification**. Sentinel can notify the system administrator, or another computer program, via an API, that something is wrong with one of the monitored Redis instances.
- **Automatic failover**. If a master is not working as expected, Sentinel can start a failover process where a slave is promoted to master, the other additional slaves are reconfigured to use the new master, and the applications using the Redis server informed about the new address to use when connecting.

Redis Sentinel is a distributed system, this means that usually you want to run multiple Sentinel processes across your infrastructure, and this processes will use agreement protocols in order to understand if a master is down and to perform the failover.

Redis Sentinel is shipped as a stand-alone executable called `redis-sentinel` but actually it is a special execution mode of the Redis server itself, and can be also invoked using the `--sentinel` option of the normal `redis-sever` executable.

WARNING: Redis Sentinel is currently a work in progress. This document describes how to use what we is already implemented, and may change as the Sentinel implementation evolves.

Redis Sentinel is compatible with Redis 2.4.16 or greater, and redis 2.6.0-rc6 or greater.

Obtaining Sentinel

Currently Sentinel is part of the Redis *unstable* branch at github. To compile it you need to clone the *unstable* branch and compile Redis. You'll see a `redis-sentinel` executable in your `src` directory.

Alternatively you can use directly the `redis-server` executable itself, starting it in Sentinel mode as specified in the next paragraph.

Running Sentinel

If you are using the `redis-sentinel` executable (or if you have a symbolic link with that name to the `redis-server` executable) you can run Sentinel with the following command line:

```
redis-sentinel /path/to/sentinel.conf
```

Otherwise you can use directly the `redis-server` executable starting it in Sentinel mode:

```
redis-server /path/to/sentinel.conf --sentinel
```

Both ways work the same.

Configuring Sentinel

The Redis source distribution contains a file called `sentinel.conf` that is a self-documented example configuration file you can use to configure Sentinel, however a typical minimal configuration file looks like the following:

```
sentinel monitor mymaster 127.0.0.1 6379 2
```

The first line is used to tell Redis to monitor a master called *mymaster*, that is at address 127.0.0.1 and port 6379, with a level of agreement needed to detect this master as failing of 2 sentinels (if the agreement is not reached the automatic failover does not start).

The other options are almost always in the form:

```
sentinel <option_name> <master_name> <option_value>
```

And are used for the following purposes:

- `down-after-milliseconds` is the time in milliseconds an instance should not be reachable (either does not reply to our PINGs or it is replying with an error) for a Sentinel starting to think it is down. After this time has elapsed the Sentinel will mark an instance as **subjectively down** (also known as `SDOWN`), that is not enough to start the automatic failover. However if enough instances will think that there is a subjectively down condition, then the instance is marked as **objectively down**. The number of sentinels that needs to agree depends on the configured agreement for this master.
- `can-failover` tells this Sentinel if it should start a failover when an instance is detected as objectively down (also called `ODOWN` for simplicity). You may configure all the Sentinels to perform the failover if needed, or you may have a few Sentinels used only to reach the agreement, and a few more that are actually in charge to perform the failover.
- `parallel-syncs` sets the number of slaves that can be reconfigured to use the new master after a failover at the same time. The lower the number, the more time it will take for the failover process to complete, however if the slaves are configured to serve old data, you may not want all the slaves to resync at the same time with the new master, as while the replication process is mostly non blocking for a slave, there is a moment when it stops to load the bulk data from the master during a resync. You may make sure only one slave at a time is not reachable by setting this option to the value of 1.

The other options are described in the rest of this document and documented in the example sentinel.conf file shipped with the Redis distribution.

SDOWN and ODOWN

As already briefly mentioned in this document Redis Sentinel has two different concepts of *being down*, one is called a *Subjectively Down* condition (SDOWN) and is a down condition that is local to a given Sentinel instance. Another is called *Objectively Down* condition (ODOWN) and is reached when enough Sentinels (at least the number configured as the `quorum` parameter of the monitored master) have an SDOWN condition, and get feedbacks from other Sentinels using the `SENTINEL is-master-down-by-addr` command.

From the point of view of a Sentinel an SDOWN condition is reached if we don't receive a valid reply to PING requests for the number of seconds specified in the configuration as `is-master-down-after-milliseconds` parameter.

An acceptable reply to PING is one of the following:

- PING replied with +PONG.
- PING replied with -LOADING error.
- PING replied with -MASTERDOWN error.

Any other reply (or no reply) is considered non valid.

Note that SDOWN requires that no acceptable reply is received for the whole interval configured, so for instance if the interval is 30000 milliseconds (30 seconds) and we receive an acceptable ping reply every 29 seconds, the instance is considered to be working.

The ODOWN condition **only applies to masters**. For other kind of instances Sentinel don't require any agreement, so the ODOWN state is never reached for slaves and other sentinels.

The behavior of Redis Sentinel can be described by a set of rules that every Sentinel follows. The complete behavior of Sentinel as a distributed system composed of multiple Sentinels only results from this rules followed by every single Sentinel instance. The following is the first set of rules. In the course of this document more rules will be added in the appropriate sections.

Sentinel Rule #1: Every Sentinel sends a **PING** request to every known master, slave, and sentinel instance, every second.

Sentinel Rule #2: An instance is Subjectively Down (**SDOWN**) if the latest valid reply to **PING** was received more than `down-after-milliseconds` milliseconds ago. Acceptable PING replies are: +PONG, -LOADING, -MASTERDOWN.

Sentinel Rule #3: Every Sentinel is able to reply to the command **SENTINEL is-master-down-by-addr `<ip> <port>`**. This command replies true if the specified address is the one of a master instance, and the master is in **SDOWN** state.

Sentinel Rule #4: If a master is in **SDOWN** condition, every other Sentinel also monitoring this master, is queried for confirmation of this state, every second, using the **SENTINEL is-master-down-by-addr** command.

Sentinel Rule #5: If a master is in **SDOWN** condition, and enough other Sentinels (to reach the configured quorum) agree about the condition, with a reply to **SENTINEL is-master-down-by-addr** that is no older than five seconds, then the master is marked as Objectively Down (**ODOWN**).

Sentinel Rule #6: Every Sentinel sends an **INFO** request to every known master and slave instance, one time every 10 seconds. If a master is in **ODOWN** condition, its slaves are asked for **INFO** every second instead of being asked every 10 seconds.

Sentinel Rule #7: If the **first** INFO reply a Sentinel receives about a master shows that it is actually a slave, Sentinel will update the configuration to actually monitor the master reported by the INFO output instead. So it is safe to start Sentinel against slaves.

Sentinels and Slaves auto discovery

While Sentinels stay connected with other Sentinels in order to reciprocally check the availability of each other, and to exchange messages, you don't need to configure the other Sentinel addresses in every Sentinel instance you run, as Sentinel uses the Redis master Pub/Sub capabilities in order to discover the other Sentinels that are monitoring the same master.

This is obtained by sending *Hello Messages* into the channel named `__sentinel__:hello`.

Similarly you don't need to configure what is the list of the slaves attached to a master, as Sentinel will auto discover this list querying Redis.

Sentinel Rule #8: Every Sentinel publishes a message to every monitored master Pub/Sub channel `__sentinel__:hello`, every five seconds, announcing its presence with ip, port, runid, and ability to failover (accordingly to `can-failover` configuration directive in `sentinel.conf`).

Sentinel Rule #9: Every Sentinel is subscribed to the Pub/Sub channel `__sentinel__:hello` of every master, looking for unknown sentinels. When new sentinels are detected, we add them as sentinels of this master.

Sentinel Rule #10: Before adding a new sentinel to a master a Sentinel always checks if there is already a sentinel with the same runid or the same address (ip and port pair). In that case all the matching sentinels are removed, and the new added.

Sentinel API

By default Sentinel runs using TCP port 26379 (note that 6379 is the normal Redis port). Sentinels accept commands using the Redis protocol, so you can use `redis-cli` or any other unmodified Redis client in order to talk with Sentinel.

There are two ways to talk with Sentinel: it is possible to directly query it to check what is the state of the monitored Redis instances from its point of view, to see what other Sentinels it knows, and so forth.

An alternative is to use Pub/Sub to receive *push style* notifications from Sentinels, every time some event happens, like a failover, or an instance entering an error condition, and so forth.

Sentinel commands

The following is a list of accepted commands: * **PING** this command simply returns PONG. * **SENTINEL masters** show a list of monitored masters and their state. * **SENTINEL slaves <master name>** show a list of slaves for this master, and their state. * **SENTINEL is-master-down-by-addr <ip> <port>** return a two elements multi bulk reply where the first is 0 or 1 (0 if the master with that address is known and is in `SDOWN` state, 1 otherwise). The second element of the reply is the *subjective leader* for this master, that is, the `runid` of the Redis Sentinel instance that should perform the failover accordingly to the queried instance. * **SENTINEL get-master-addr-by-name <master name>** return the ip and port number of the master with that name. If a failover is in progress or terminated successfully for this master it returns the address and port of the promoted slave. * **SENTINEL reset <pattern>** this command will reset all the masters with matching name. The pattern argument is a glob-style pattern. The reset process clears any previous state in a master (including a failover in progress), and removes every slave and sentinel already discovered and associated with the master.

Pub/Sub Messages

A client can use a Sentinel as it was a Redis compatible Pub/Sub server (but you can't use PUBLISH) in order to SUBSCRIBE or PSUBSCRIBE to channels and get notified about specific events.

The channel name is the same as the name of the event. For instance the channel named `+sdown` will receive all the notifications related to instances entering an `SDOWN` condition.

To get all the messages simply subscribe using `PSUBSCRIBE *`.

The following is a list of channels and message formats you can receive using this API. The first word is the channel / event name, the rest is the format of the data.

Note: where *instance details* is specified it means that the following arguments are provided to identify the target instance:

```
<instance-type> <name> <ip> <port> @ <master-name> <master-ip> <master-port>
```

The part identifying the master (from the @ argument to the end) is optional and is only specified if the instance is not a master itself.

- **+reset-master** `<instance details>` -- The master was reset.
- **+slave** `<instance details>` -- A new slave was detected and attached.
- **+failover-state-reconf-slaves** `<instance details>` -- Failover state changed to `reconf-slaves` state.
- **+failover-detected** `<instance details>` -- A failover started by another Sentinel or any other external entity was detected (An attached slave turned into a master).
- **+slave-reconf-sent** `<instance details>` -- The leader sentinel sent the SLAVEOF command to this instance in order to reconfigure it for the new slave.
- **+slave-reconf-inprog** `<instance details>` -- The slave being reconfigured showed to be a slave of the new master ip:port pair, but the synchronization process is not yet complete.
- **+slave-reconf-done** `<instance details>` -- The slave is now synchronized with the new master.
- **-dup-sentinel** `<instance details>` -- One or more sentinels for the specified master were removed as duplicated (this happens for instance when a Sentinel instance is restarted).
- **+sentinel** `<instance details>` -- A new sentinel for this master was detected and attached.
- **+sdown** `<instance details>` -- The specified instance is now in Subjectively Down state.
- **-sdown** `<instance details>` -- The specified instance is no longer in Subjectively Down state.
- **+odown** `<instance details>` -- The specified instance is now in Objectively Down state.
- **-odown** `<instance details>` -- The specified instance is no longer in Objectively Down state.
- **+failover-takedown** `<instance details>` -- 25% of the configured failover timeout has elapsed, but this sentinel can't see any progress, and is the new leader. It starts to act as the new leader reconfiguring the remaining slaves to replicate with the new master.
- **+failover-triggered** `<instance details>` -- We are starting a new failover as a the leader sentinel.
- **+failover-state-wait-start** `<instance details>` -- New failover state is `wait-start`: we are waiting a fixed number of seconds, plus a random number of seconds before starting the failover.
- **+failover-state-select-slave** `<instance details>` -- New failover state is `select-slave`: we are trying to find a suitable slave for promotion.
- **no-good-slave** `<instance details>` -- There is no good slave to promote. Currently we'll try

after some time, but probably this will change and the state machine will abort the failover at all in this case.

- **selected-slave** `<instance details>` -- We found the specified good slave to promote.
- **failover-state-send-slaveof-noone** `<instance details>` -- We are trynig to reconfigure the promoted slave as master, waiting for it to switch.
- **failover-end-for-timeout** `<instance details>` -- The failover terminated for timeout. If we are the failover leader, we sent a *best effort* SLAVEOF command to all the slaves yet to reconfigure.
- **failover-end** `<instance details>` -- The failover terminated with success. All the slaves appears to be reconfigured to replicate with the new master.
- **switch-master** `<master name> <oldip> <oldport> <newip> <newport>` -- We are starting to monitor the new master, using the same name of the old one. The old master will be completely removed from our tables.
- **failover-abort-x-sdown** `<instance details>` -- The failover was undoed (aborted) because the promoted slave appears to be in extended SDOWN state.
- **-slave-reconf-undo** `<instance details>` -- The failover aborted so we sent a SLAVEOF command to the specified instance to reconfigure it back to the original master instance.
- **+tilt** -- Tilt mode entered.
- **-tilt** -- Tilt mode exited.

Sentinel failover

The failover process consists on the following steps:

- Recognize that the master is in ODOWN state.
- Understand who is the Sentinel that should start the failover, called **The Leader**. All the other Sentinels will be **The Observers**.
- The leader selects a slave to promote to master.
- The promoted slave is turned into a master with the command **SLAVEOF NO ONE**.
- The observers see that a slave was turned into a master, so they know the failover started. **Note:** this means that any event that turns one of the slaves of a monitored master into a master (`SLAVEOF NO ONE` command) will be sensed as the start of a failover process.
- All the other slaves attached to the original master are configured with the **SLAVEOF** command in order to start the replication process with the new master.
- The leader terminates the failover process when all the slaves are reconfigured. It removes the old master from the table of monitored masters and adds the new master, *under the same name* of the original master.
- The observers detect the end of the failover process when all the slaves are reconfigured. They remove the old master from the table and start monitoring the new master, exactly as the leader does.

The election of the Leader is performed using the same mechanism used to reach the ODOWN state, that is, the **SENTINEL is-master-down-by-addr** command. It returns the leader from the point of view of the queried Sentinel, we call it the **Subjective Leader**, and is selected using the following rule:

- We remove all the Sentinels that can't failover for configuration (this information is propagated using the Hello Channel to all the Sentinels).
- We remove all the Sentinels in SDOWN, disconnected, or with the last ping reply received more than `SENTINEL_INFO_VALIDITY_TIME` milliseconds ago (currently defined as 5 seconds).
- Of all the remaining instances, we get the one with the lowest `runid`, lexicographically (every Redis instance has a Run ID, that is an identifier of every single execution).

For a Sentinel to sense to be the **Objective Leader**, that is, the Sentinel that should start the failove process, the following conditions are needed.

- It thinks it is the subjective leader itself.
- It receives acknowledges from other Sentinels about the fact it is the leader: at least 50% plus one of all the Sentinels that were able to reply to the `SENTINEL is-master-down-by-addr` request shoudl agree it is the leader, and additionally we need a total level of agreement at least equal to the configured quorum of the master instance that we are going to failover.

Once a Sentinel things it is the Leader, the failover starts, but there is always a delay of five seconds plus an additional random delay. This is an additional layer of protection because if during this period we see another instance turning a slave into a master, we detect it as another instance staring the failover and turn ourselves into an observer instead.

Sentinel Rule #11: A **Good Slave** is a slave with the following requirements: * It is not in SDOWN nor in ODOWN condition. * We have a valid connection to it currently (not in DISCONNECTED state). * Latest PING reply we received from it is not older than five seconds. * Latest INFO reply we received from it is not older than five seconds. * The latest INFO reply reported that the link with the master is down for no more than the time elapsed since we saw the master entering SDOWN state, plus ten times the configured `down_after_milliseconds` parameter. So for instance if a Sentinel is configured to sense the SDOWN condition after 10 seconds, and the master is down since 50 seconds, we accept a slave as a Good Slave only if the replication link was disconnected less than `50+(10*10)` seconds (two minutes and half more or less).

Sentinel Rule #12: A **Subjective Leader** from the point of view of a Sentinel, is the Sentinel (including itself) with the lower runid monitoring a given master, that also replied to PING less than 5 seconds ago, reported to be able to do the failover via Pub/Sub hello channel, and is not in DISCONNECTED state.

Sentinel Rule #12: If a master is down we ask `SENTINEL is-master-down-by-addr` to every other connected Sentinel as explained in Sentinel Rule #4. This command will also reply with the runid of the **Subjective Leader** from the point of view of the asked Sentinel. A given Sentinel believes to be the **Objective Leader** of a master if it is reported to be the subjective leader by N Sentinels (including itself), where: * N must be equal or greater to the configured quorum for this master. * N mast be equal or greater to the majority of the voters (`num_votres/2+1`), considering only the Sentinels that also reported the master to be down.

Sentinel Rule #13: A Sentinel starts the failover as a **Leader** (that is, the Sentinel actually sending the commands to reconfigure the Redis servers) if the following conditions are true at the same time: * The master is in ODOWN condition. * The Sentinel is configured to perform the failover with `can-failover` set to yes. * There is at least a Good Slave from the point of view of the Sentinel. * The Sentinel believes to be the Objective Leader. * There is no failover in progress already detected for this master.

Sentinel Rule #14: A Sentinel detects a failover as an **Observer** (that is, the Sentinel just follows the failover generating the appropriate events in the log file and Pub/Sub interface, but without actively reconfiguring instances) if the following conditions are true at the same time: * There is no failover already in progress. * A slave instance of the monitored master turned into a master. However the failover **will NOT be sensed as started if the slave instance turns into a master and at the same time the runid has changed** from the previous one. This means the instance turned into a master because of a restart, and is not a valid condition to consider it a slave election.

Sentinel Rule #15: A Sentinel starting a failover as leader does not immediately starts it. It enters a state called **wait-start**, that lasts a random amount of time between 5 seconds and 15 seconds. During this time

Sentinel Rule #14 still applies: if a valid slave promotion is detected the failover as leader is aborted and the failover as observer is detected.

End of failover

The failover process is considered terminated from the point of view of a single Sentinel if:

- The promoted slave is not in SDOWN condition.
- A slave was promoted as new master.
- All the other slaves are configured to use the new master.

Note: Slaves that are in SDOWN state are ignored.

Also the failover state is considered terminate if:

- The promoted slave is not in SDOWN condition.
- A slave was promoted as new master.
- At least `failover-timeout` milliseconds elapsed since the last progress.

The `failover-timeout` value can be configured in sentinel.conf for every different slave.

Note that when a leader terminates a failover for timeout, it sends a SLAVEOF command in a best-effort way to all the slaves yet to be configured, in the hope that they'll receive the command and replicate with the new master eventually.

Sentinel Rule #16 A failover is considered complete if for a leader or observer if: * One slave was promoted to master (and the Sentinel can detect that this actually happened via INFO output), and all the additional slaves are all configured to replicate with the new slave (again, the sentinel needs to sense it using the INFO output). * There is already a correctly promoted slave, but the configured `failover-timeout` time has already elapsed without any progress in the reconfiguration of the additional slaves. In this case a leader sends a best effort SLAVEOF command is sent to all the not yet configured slaves. In both the two above conditions the promoted slave **must be reachable** (not in SDOWN state), otherwise a failover is never considered to be complete.

Leader failing during failover

If the leader fails when it has yet to promote the slave into a master, and it fails in a way that makes it in SDOWN state from the point of view of the other Sentinels, if enough Sentinels remained to reach the quorum the failover will automatically continue using a new leader (the subjective leader of all the remaining Sentinels will change because of the SDOWN state of the previous leader).

If the failover was already in progress and the slave was already promoted, and possibly a few other slaves were already reconfigured, an observer that is the new objective leader will continue the failover in case no progresses are made for more than 25% of the time specified by the `failover-timeout` configuration option.

Note that this is safe as multiple Sentinels trying to reconfigure slaves with duplicated SLAVEOF commands do not create any race condition, but at the same time we want to be sure that all the slaves are reconfigured in the case the original leader is no longer working.

Sentinel Rule #17 A Sentinel that is an observer for a failover in progress will turn itself into a failover leader, continuing the configuration of the additional slaves, if all the following conditions are true: * A failover is in progress, and this Sentinel is an observer. * It detects to be an objective leader (so likely the previous leader is no longer reachable by other sentinels). * At least 25% of the configured `failover-timeout` has elapsed without any progress in the observed failover process.

Promoted slave failing during failover

If the promoted slave has an active SDOWN condition, a Sentinel will never sense the failover as terminated.

Additionally if there is an *extended SDOWN condition* (that is an SDOWN that lasts for more than ten times `down-after-milliseconds` milliseconds) the failover is aborted (this happens for leaders and observers), and the master starts to be monitored again as usually, so that a new failover can start with a different slave in case the master is still failing.

Note that when this happens it is possible that there are a few slaves already configured to replicate from the (now failing) promoted slave, so when the leader sentinel aborts a failover it sends a SLAVEOF command to all the slaves already reconfigured or in the process of being reconfigured to switch the configuration back to the original master.

Sentinel Rule #18 A Sentinel will consider the failover process aborted, both when acting as leader and when acting as an observer, in the following conditions are true: * A failover is in progress and a slave to promote was already selected (or in the case of the observer was already detected as master). * The promoted slave is in **Extended SDOWN** condition (continually in SDOWN condition for at least ten times the configured `down-after-milliseconds`).

Manual interactions

- TODO: Manually triggering a failover with SENTINEL FAILOVER.
- TODO: Pausing Sentinels with SENTINEL PAUSE, RESUME.

The failback process

- TODO: Sentinel does not perform automatic Failback.
- TODO: Document correct steps for the failback.

Clients configuration update

Work in progress.

TILT mode

Redis Sentinel is heavily dependent on the computer time: for instance in order to understand if an instance is available it remembers the time of the latest successful reply to the PING command, and compares it with the current time to understand how old it is.

However if the computer time changes in an unexpected way, or if the computer is very busy, or the process blocked for some reason, Sentinel may start to behave in an unexpected way.

The TILT mode is a special "protection" mode that a Sentinel can enter when something odd is detected that can lower the reliability of the system. The Sentinel timer interrupt is normally called 10 times per second, so we expect that more or less 100 milliseconds will elapse between two calls to the timer interrupt.

What a Sentinel does is to register the previous time the timer interrupt was called, and compare it with the current call: if the time difference is negative or unexpectedly big (2 seconds or more) the TILT mode is entered (or if it was already entered the exit from the TILT mode postponed).

When in TILT mode the Sentinel will continue to monitor everything, but:

- It stops acting at all.
- It starts to reply negatively to `SENTINEL is-master-down-by-addr` requests as the ability to detect a failure is no longer trusted.

If everything appears to be normal for 30 second, the TILT mode is exited.

Handling of -BUSY state

(Warning: Yet not implemented)

The -BUSY error is returned when a script is running for more time than the configured script time limit. When this happens before triggering a fail over Redis Sentinel will try to send a "SCRIPT KILL" command, that will only succeed if the script was read-only.

Notifications via user script

Work in progress.

Suggested setup

Work in progress.

APPENDIX A - Implementation and algorithms

Duplicate Sentinels removal

In order to reach the configured quorum we absolutely want to make sure that the quorum is reached by different physical Sentinel instances. Under no circumstance we should get agreement from the same instance that for some reason appears to be two or multiple distinct Sentinel instances.

This is enforced by an aggressive removal of duplicated Sentinels: every time a Sentinel sends a message in the Hello Pub/Sub channel with its address and runid, if we can't find a perfect match (same runid and address) inside the Sentinels table for that master, we remove any other Sentinel with the same runid OR the same address. And later add the new Sentinel.

For instance if a Sentinel instance is restarted, the Run ID will be different, and the old Sentinel with the same IP address and port pair will be removed.

Selection of the Slave to promote

If a master has multiple slaves, the slave to promote to master is selected checking the slave priority (a new configuration option of Redis instances that is propagated via INFO output, still not implemented), and picking the one with lower priority value (it is an integer similar to the one of the MX field of the DNS system).

All the slaves that appears to be disconnected from the master for a long time are discarded.

If slaves with the same priority exist, the one with the lexicographically smaller Run ID is selected.

Note: because currently slave priority is not implemented, the selection is performed only discarding unreachable slaves and picking the one with the lower Run ID.

Sentinel Rule #19: A Sentinel performing the failover as leader will select the slave to promote, among the existing **Good Slaves** (See rule #11), taking the one with the lower slave priority. When priority is the same the slave with lexicographically lower runid is preferred.

APPENDIX B - Get started with Sentinel in five minutes

If you want to try Redis Sentinel, please follow this steps:

- Clone the *unstable* branch of the Redis repository at github (it is the default branch).
- Compile it with "make".
- Start a few normal Redis instances, using the `redis-server` compiled in the *unstable* branch. One master and one slave is enough.
- Use the `redis-sentinel` executable to start three instances of Sentinel, with `redis-sentinel /path/to/config`.

To create the three configurations just create three files where you put something like that:

```
port 26379
```

Note: where you see `port 26379`, use 26380 for the second Sentinel, and 26381 for the third Sentinel (any other different non colliding port will do of course). Also note that the `down-after-milliseconds` configuration option is set to just five seconds, that is a good value to play with Sentinel, but not good for production environments.

At this point you should see something like the following in every Sentinel you are running:

```
[4747] 23 Jul 14:49:15.883 * +slave slave 127.0.0.1:6380 127.0.0.1 6380 @ mymaster 127.0.0.1 63
```

To see how the failover works, just put down your slave (for instance sending `DEBUG SEGFAULT` to crash it) and see what happens.

This HOWTO is a work in progress, more information will be added in the near future.

This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

Redis Sponsors

All the work Salvatore Sanfilippo and Pieter Noordhuis are doing in order to develop Redis is sponsored by VMware. The Redis project no longer accepts money donations.

In the past Redis accepted donations from the following companies:

- Linode 15 January 2010, provided Virtual Machines for Redis testing in a virtualized environment.
- Slicehost 14 January 2010, provided Virtual Machines for Redis testing in a virtualized environment.
- Citrusbyte 18 Dec 2009, part of Virtual Memory. Citrusbyte is also the company developing the Redis-rb bindings for Redis and this very web site.
- Hitmeister 15 Dec 2009, part of Redis Cluster.
- Engine Yard 13 Dec 2009, for blocking POP (BLPOP) and part of the Virtual Memory implementation.

Also thanks to the following people or organizations that donated to the Project:

- Emil Vladev
- Brad Jasper
- Mrkris

We are grateful to VMware and to the companies and people that donated to the Redis project. Thank you.

The Redis.io domain is kindly donated to the project by I Want My Name.

This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

Related commands

- DISCARD
- EXEC
- MULTI
- UNWATCH
- WATCH

Transactions

MULTI, EXEC, DISCARD and WATCH are the foundation of transactions in Redis. They allow the execution of a group of commands in a single step, with two important guarantees:

- All the commands in a transaction are serialized and executed sequentially. It can never happen that a request issued by another client is served **in the middle** of the execution of a Redis transaction. This guarantees that the commands are executed as a single atomic operation.
- Either all of the commands or none are processed. The EXEC command triggers the execution of all the commands in the transaction, so if a client loses the connection to the server in the context of a transaction before calling the MULTI command none of the operations are performed, instead if the EXEC command is called, all the operations are performed. When using the append-only file Redis makes sure to use a single write(2) syscall to write the transaction on disk. However if the Redis server crashes or is killed by the system administrator in some hard way it is possible that only a partial number of operations are registered. Redis will detect this condition at restart, and will exit with an error. Using the `redis-check-aof` tool it is possible to fix the append only file that will remove the partial transaction so that the server can start again.

Redis 2.2 allows for an extra guarantee to the above two, in the form of optimistic locking in a way very similar to a check-and-set (CAS) operation. This is documented later on this page.

Usage

A Redis transaction is entered using the MULTI command. The command always replies with `OK`. At this point the user can issue multiple commands. Instead of executing these commands, Redis will queue them. All the commands are executed once EXEC is called.

Calling DISCARD instead will flush the transaction queue and will exit the transaction.

The following example increments keys `foo` and `bar` atomically.

```
> MULTI
```

As it is possible to see from the session above, MULTI returns an array of replies, where every element is the reply of a single command in the transaction, in the same order the commands were issued.

When a Redis connection is in the context of a MULTI request, all commands will reply with the string `QUEUED` unless they are syntactically incorrect. Some commands are still allowed to fail during execution time.

This is more clear on the protocol level. In the following example one command will fail when executed even if the syntax is right:

```
Trying 127.0.0.1...
```

MULTI returned two-element Bulk reply where one is an `OK` code and the other an `-ERR` reply. It's up to the client library to find a sensible way to provide the error to the user.

It's important to note that **even when a command fails, all the other commands in the queue are processed** â Redis will *not* stop the processing of commands.

Another example, again using the wire protocol with `telnet`, shows how syntax errors are reported ASAP instead:

```
MULTI
```

This time due to the syntax error the bad INCR command is not queued at all.

Errors inside a transaction

If you have a relational databases background, the fact that Redis commands can fail during a transaction, but still Redis will execute the rest of the transaction instead of rolling back, may look odd to you.

However there are good opinions for this behavior:

- Redis commands can fail only if called with a wrong syntax, or against keys holding the wrong data type: this means that in practical terms a failing command is the result of a programming errors, and a kind of error that is very likely to be detected during development, and not in production.
- Redis is internally simplified and faster because it does not need the ability to roll back.

An argument against Redis point of view is that bugs happen, however it should be noted that in general the roll back does not save you from programming errors. For instance if a query increments a key by 2 instead of 1, or increments the wrong key, there is no way for a rollback mechanism to help. Given that no one can save the programmer from his errors, and that the kind of errors required for a Redis command to fail are unlikely to enter in production, we selected the simpler and faster approach of not supporting roll backs on errors.

Discarding the command queue

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.

```
> SET foo 1
```

Optimistic locking using check-and-set

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

`WATCH`ed 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 multi-bulk reply to notify

that the transaction failed.

For example, imagine we have the need to atomically increment the value of a key by 1 (let's suppose Redis doesn't have INCR).

The first try may be the following:

```
val = GET mykey
```

This will work reliably only if we have a single client performing the operation in a given time. If multiple clients try to increment the key at about the same time there will be a race condition. For instance, client A and B will read the old value, for instance, 10. The value will be incremented to 11 by both the clients, and finally SET as the value of the key. So the final value will be 11 instead of 12.

Thanks to WATCH we are able to model the problem very well:

```
WATCH mykey
```

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.

We just have to repeat the operation hoping this time we'll not get a new race. This form of locking is called *optimistic locking* and is a very powerful form of locking. In many use cases, multiple clients will be accessing different keys, so collisions are unlikely â usually there's no need to repeat the operation.

WATCH explained

So what is WATCH really about? It is a command that will make the EXEC conditional: we are asking Redis to perform the transaction only if no other client modified any of the `WATCH`ed keys. Otherwise the transaction is not entered at all. (Note that if you WATCH a volatile key and Redis expires the key after you `WATCH`ed it, EXEC will still work. More on this.)

WATCH can be called multiple times. Simply all the WATCH calls will have the effects to watch for changes starting from the call, up to the moment EXEC is called. You can also send any number of keys to a single WATCH call.

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

It is also possible to use the UNWATCH command (without arguments) in order to flush all the watched keys. Sometimes this is useful as we optimistically lock a few keys, since possibly we need to perform a transaction to alter those keys, but after reading the current content of the keys we don't want to proceed. When this happens we just call UNWATCH so that the connection can already be used freely for new transactions.

Using WATCH to implement ZPOP

A good example to illustrate how WATCH can be used to create new atomic operations otherwise not supported by Redis is to implement ZPOP, that is a command that pops the element with the lower score from a sorted set in an atomic way. This is the simplest implementation:

```
WATCH zset
```

If EXEC fails (i.e. returns a Null multi-bulk reply) we just repeat the operation.

Redis scripting and transactions

A Redis script is transactional by definition, so everything you can do with a Redis transaction, you can also do with a script, and usually the script will be both simpler and faster.

This duplication is due to the fact that scripting was introduced in Redis 2.6 while transactions already existed long before. However we are unlikely to remove the support for transactions in the short time because it seems semantically opportune that even without resorting to Redis scripting it is still possible to avoid race conditions, especially since the implementation complexity of Redis transactions is minimal.

However it is not impossible that in a non immediate future we'll see that the whole user base is just using scripts. If this happens we may deprecate and finally remove transactions.

A case study: Design and implementation of a simple Twitter clone using only the Redis key-value store as database and PHP

In this article I'll explain the design and the implementation of a simple clone of Twitter written using PHP and Redis as only database. The programming community uses to look at key-value stores like special databases that can't be used as drop in replacement for a relational database for the development of web applications. This article will try to prove the contrary.

Our Twitter clone, called Retwis, is structurally simple, has very good performance, and can be distributed among N web servers and M Redis servers with very little effort. You can find the source code here.

We use PHP for the example since it can be read by everybody. The same (or... much better) results can be obtained using Ruby, Python, Erlang, and so on.

Note: Retwis-RB is a port of Retwis to Ruby and Sinatra written by Daniel Lucraft! With full source code included of course, the Git repository is linked in the footer of the web page. The rest of this article targets PHP, but Ruby programmers can also check the other source code, it conceptually very similar.

Note: Retwis-J is a port of Retwis to Java, using the Spring Data Framework, written by Costin Leau. The source code can be found on GitHub and there is comprehensive documentation available at springsource.org.

Key-value stores basics

The essence of a key-value store is the ability to store some data, called *value*, inside a key. This data can later be retrieved only if we know the exact key used to store it. There is no way to search something by value. In a sense, it is like a very large hash/dictionary, but it is persistent, i.e. when your application ends, the data doesn't go away. So for example I can use the command SET to store the value *bar* at key *foo*:

```
SET foo bar
```

Redis will store our data permanently, so we can later ask for "*What is the value stored at key foo?*" and Redis will reply with *bar*:

```
GET foo => bar
```

Other common operations provided by key-value stores are DEL used to delete a given key, and the associated value, SET-if-not-exists (called SETNX on Redis) that sets a key only if it does not already exist, and INCR that is able to atomically increment a number stored at a given key:

```
SET foo 10
```

Atomic operations

So far it should be pretty simple, but there is something special about INCR. Think about this, why to provide such an operation if we can do it ourselves with a bit of code? After all it is as simple as:

```
x = GET foo
```

The problem is that doing the increment this way will work as long as there is only a client working with the value *x* at a time. See what happens if two computers are accessing this data at the same time:

```
x = GET foo (yields 10)
```

Something is wrong with that! We incremented the value two times, but instead to go from 10 to 12 our key holds 11. This is because the INCR operation done with `GET / increment / SET` *is not an atomic operation*. Instead the INCR provided by Redis, Memcached, ..., are atomic implementations, the server will take care to protect the get-increment-set for all the time needed to complete in order to prevent simultaneous accesses.

What makes Redis different from other key-value stores is that it provides more operations similar to INCR that can be used together to model complex problems. This is why you can use Redis to write whole web applications without using an SQL database and without going crazy.

Beyond key-value stores

In this section we will see what Redis features we need to build our Twitter clone. The first thing to know is that Redis values can be more than strings. Redis supports Lists and Sets as values, and there are atomic operations to operate against this more advanced values so we are safe even with multiple accesses against the same key. Let's start from Lists:

```
LPUSH mylist a (now mylist holds one element list 'a')
```

LPUSH means *Left Push*, that is, add an element to the left (or to the head) of the list stored at *mylist*. If the key *mylist* does not exist it is automatically created by Redis as an empty list before the PUSH operation. As you can imagine, there is also the RPUSH operation that adds the element on the right of the list (on the tail).

This is very useful for our Twitter clone. Updates of users can be stored into a list stored at `username:updates` for instance. There are operations to get data or information from Lists of course. For instance LRANGE returns a range of the list, or the whole list.

```
LRANGE mylist 0 1 => c,b
```

LRANGE uses zero-based indexes, that is the first element is 0, the second 1, and so on. The command arguments are `LRANGE key first-index last-index`. The *last index* argument can be negative, with a special meaning: -1 is the last element of the list, -2 the penultimate, and so on. So in order to get the whole list we can use:

```
LRANGE mylist 0 -1 => c,b,a
```

Other important operations are LLEN that returns the length of the list, and LTRIM that is like LRANGE but instead of returning the specified range *trims* the list, so it is like *Get range from mylist, Set this range as new value* but atomic. We will use only this List operations, but make sure to check the Redis documentation to discover all the List operations supported by Redis.

The set data type

There is more than Lists, Redis also supports Sets, that are unsorted collection of elements. It is possible to add, remove, and test for existence of members, and perform intersection between different Sets. Of course it is possible to ask for the list or the number of elements of a Set. Some example will make it more clear. Keep in mind that SADD is the *add to set* operation, SREM is the *remove from set* operation, *sismember* is the *test if it is a member* operation, and SINTER is *perform intersection* operation. Other operations are SCARD that is used to get the cardinality (the number of elements) of a Set, and SMEMBERS that will return all the members of a Set.

```
SADD myset a
```

Note that SMEMBERS does not return the elements in the same order we added them, since Sets are *unsorted* collections of elements. When you want to store the order it is better to use Lists instead. Some more operations against Sets:

```
SADD mynewset b
```

SINTER can return the intersection between Sets but it is not limited to two sets, you may ask for intersection of 4,5 or 10000 Sets. Finally let's check how SISMEMBER works:

```
SISMEMBER myset foo => 1
```

Okay, I think we are ready to start coding!

Prerequisites

If you didn't download it already please grab the source code of Retwis. It's a simple tar.gz file with a few of PHP files inside. The implementation is very simple. You will find the PHP library client inside (redis.php) that is used to talk with the Redis server from PHP. This library was written by Ludovico Magnocavallo and you are free to reuse this in your own projects, but for updated version of the library please download the Redis distribution. (Note: there are now better PHP libraries available, check our clients page.

Another thing you probably want is a working Redis server. Just get the source, compile with make, and run with ./redis-server and you are done. No configuration is required at all in order to play with it or to run Retwis in your computer.

Data layout

Working with a relational database this is the stage were the database layout should be produced in form of tables, indexes, and so on. We don't have tables, so what should be designed? We need to identify what keys are needed to represent our objects and what kind of values this keys need to hold.

Let's start from Users. We need to represent this users of course, with the username, userid, password, followers and following users, and so on. The first question is, what should identify an user inside our system? The username can be a good idea since it is unique, but it is also too big, and we want to stay low on memory. So like if our DB was a relational one we can associate an unique ID to every user. Every other reference to this user will be done by id. That's very simple to do, because we have our atomic INCR operation! When we create a new user we can do something like this, assuming the user is called "antirez":

```
INCR global:nextUserId => 1000
```

We use the *global:nextUserId* key in order to always get an unique ID for every new user. Then we use this unique ID to populate all the other keys holding our user data. *This is a Design Pattern* with key-values stores! Keep it in mind. Besides the fields already defined, we need some more stuff in order to fully define an User. For example sometimes it can be useful to be able to get the user ID from the username, so we set this key too:

```
SET username:antirez:uid 1000
```

This may appear strange at first, but remember that we are only able to access data by key! It's not possible to tell Redis to return the key that holds a specific value. This is also *our strength*, this new paradigm is forcing us to organize the data so that everything is accessible by *primary key*, speaking with relational DBs language.

Following, followers and updates

There is another central need in our system. Every user has followers users and following users. We have a perfect data structure for this work! That is... Sets. So let's add this two new fields to our schema:

```
uid:1000:followers => Set of uids of all the followers users
```

Another important thing we need is a place were we can add the updates to display in the user home page. We'll need to access this data in chronological order later, from the most recent update to the older ones, so the perfect kind of Value for this work is a List. Basically every new update will be LPUSHed in the user updates key, and thanks to LRANGE we can implement pagination and so on. Note that we use the words *updates* and *posts* interchangeably, since updates are actually "little posts" in some way.

```
uid:1000:posts => a List of post ids, every new post is LPUSHed here.
```

Authentication

OK, we have more or less everything about the user, but authentication. We'll handle authentication in a simple but robust way: we don't want to use PHP sessions or other things like this, our system must be ready in order to be distributed among different servers, so we'll take the whole state in our Redis database. So all we need is a random string to set as the cookie of an authenticated user, and a key that will tell us what is the user ID of the client holding such a random string. We need two keys in order to make this thing working in a robust way:

```
SET uid:1000:auth fea5e81ac8ca77622bed1c2132a021f9
```

In order to authenticate an user we'll do this simple work (`login.php`): * Get the username and password via the login form * Check if the username:`<username>`:uid key actually exists * If it exists we have the user id, (i.e. 1000) * Check if uid:1000:password matches, if not, error message * Ok authenticated! Set "fea5e81ac8ca77622bed1c2132a021f9" (the value of uid:1000:auth) as "auth" cookie

This is the actual code:

```
include("retwis.php");
```

This happens every time the users log in, but we also need a function isLoggedIn in order to check if a given user is already authenticated or not. These are the logical steps preformed by the `isLoggedIn` function: *

Get the "auth" cookie from the user. If there is no cookie, the user is not logged in, of course. Let's call the value of this cookie `<authcookie>` * Check if auth:`<authcookie>` exists, and what the value (the user id) is (1000 in the example). * In order to be sure check that uid:1000:auth matches. * OK the user is authenticated, and we loaded a bit of information in the $User global variable.

The code is simpler than the description, possibly:

```
function isLoggedIn() {
```

`loadUserInfo` as separated function is an overkill for our application, but it's a good template for a complex application. The only thing it's missing from all the authentication is the logout. What we do on logout? That's simple, we'll just change the random string in uid:1000:auth, remove the old auth:`<oldauthstring>` and add a new auth:`<newauthstring>`.

Important: the logout procedure explains why we don't just authenticate the user after the lookup of auth:`<randomstring>`, but double check it against uid:1000:auth. The true authentication string is the latter, the auth:`<randomstring>` is just an authentication key that may even be volatile, or if there are bugs in the program or a script gets interrupted we may even end with multiple auth:`<something>` keys pointing to the same user id. The logout code is the following (logout.php):

```
include("retwis.php");
```

That is just what we described and should be simple to understand.

Updates

Updates, also known as posts, are even simpler. In order to create a new post on the database we do something like this:

```
INCR global:nextPostId => 10343
```

As you can see the user id and time of the post are stored directly inside the string, we don't need to lookup by time or user id in the example application so it is better to compact everything inside the post string.

After we create a post we obtain the post id. We need to LPUSH this post id in every user that's following the author of the post, and of course in the list of posts of the author. This is the file update.php that shows how this is performed:

```
include("retwis.php");
```

The core of the function is the `foreach`. We get using SMEMBERS all the followers of the current user, then the loop will LPUSH the post against the uid:`<userid>`:posts of every follower.

Note that we also maintain a timeline with all the posts. In order to do so what is needed is just to LPUSH the post against global:timeline. Let's face it, do you start thinking it was a bit strange to have to sort things added in chronological order using ORDER BY with SQL? I think so indeed.

Paginating updates

Now it should be pretty clear how we can user LRANGE in order to get ranges of posts, and render this posts

on the screen. The code is simple:

```
function showPost($id) {
```

`showPost` will simply convert and print a Post in HTML while `showUserPosts` get range of posts passing them to `showPosts`.

Following users

If user id 1000 (antirez) wants to follow user id 1001 (pippo), we can do this with just two SADD:

SADD uid:1000:following 1001 SADD uid:1001:followers 1000

Note the same pattern again and again, in theory with a relational database the list of following and followers is a single table with fields like `following_id` and `follower_id`. With queries you can extract the followers or following of every user. With a key-value DB that's a bit different as we need to set both the `1000 is following 1001` and `1001 is followed by 1000` relations. This is the price to pay, but on the other side accessing the data is simpler and ultra-fast. And having this things as separated sets allows us to do interesting stuff, for example using SINTER we can have the intersection of 'following' of two different users, so we may add a feature to our Twitter clone so that it is able to say you at warp speed, when you visit somebody' else profile, "you and foobar have 34 followers in common" and things like that.

You can find the code that sets or removes a following/follower relation at follow.php. It is trivial as you can see.

Making it horizontally scalable

Gentle reader, if you reached this point you are already an hero, thank you. Before to talk about scaling horizontally it is worth to check the performances on a single server. Retwis is *amazingly fast*, without any kind of cache. On a very slow and loaded server, apache benchmark with 100 parallel clients issuing 100000 requests measured the average pageview to take 5 milliseconds. This means you can serve millions of users every day with just a single Linux box, and this one was monkey asses slow! Go figure with more recent hardware.

So, first of all, probably you will not need more than one server for a lot of applications, even when you have a lot of users. But let's assume we *are* Twitter and need to handle a huge amount of traffic. What to do?

Hashing the key

The first thing to do is to hash the key and issue the request on different servers based on the key hash. There are a lot of well known algorithms to do so, for example check the Redis Ruby library client that implements *consistent hashing*, but the general idea is that you can turn your key into a number, and than take the reminder of the division of this number by the number of servers you have:

```
server_id = crc32(key) % number_of_servers
```

This has a lot of problems since if you add one server you need to move too much keys and so on, but this is the general idea even if you use a better hashing scheme like consistent hashing.

Ok, are key accesses distributed among the key space? Well, all the user data will be partitioned among different servers. There are no inter-keys operations used (like SINTER, otherwise you need to care that things you want to intersect will end in the same server. *This is why Redis unlike memcached does not force a specific hashing scheme, it's application specific*). Btw there are keys that are accessed more frequently.

Special keys

For example every time we post a new message, we *need* to increment the `global:nextPostId` key. How to fix this problem? A Single server will get a lot if increments. The simplest way to handle this is to have a dedicated server just for increments. This is probably an overkill btw unless you have really a lot of traffic. There is another trick. The ID does not really need to be an incremental number, but just *it needs to be unique*. So you can get a random string long enough to be unlikely (almost impossible, if it's md5-size) to collide, and you are done. We successfully eliminated our main problem to make it really horizontally scalable!

There is another one: global:timeline. There is no fix for this, if you need to take something in order you can split among different servers and *then merge* when you need to get the data back, or take it ordered and use a single key. Again if you really have so much posts per second, you can use a single server just for this. Remember that with commodity hardware Redis is able to handle 100000 writes for second, that's enough even for Twitter, I guess.

Please feel free to use the comments below for questions and feedbacks.

APPEND key value

Related commands

- **APPEND**
- BITCOUNT
- BITOP
- DECR
- DECRBY
- GET
- GETBIT
- GETRANGE
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- PSETEX
- SET
- SETBIT
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 2.0.0.

Time complexity: O(1). The amortized time complexity is O(1) assuming the appended value is small and the already present value is of any size, since the dynamic string library used by Redis will double the free space available on every reallocation.

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, so APPEND will be similar to SET in this special case.

Return value

Integer reply: the length of the string after the append operation.

Examples

redis> EXISTS mykey

```
(integer) 0
```

redis> APPEND mykey "Hello"

```
(integer) 5
```

redis> APPEND mykey " World"

```
(integer) 11
```

redis> GET mykey

```
"Hello World"
```

redis>

Pattern: Time series

The APPEND command can be used to create a very compact representation of a list of fixed-size samples, usually referred as *time series*. Every time a new sample arrives we can store it using the command

```
APPEND timeseries "fixed-size sample"
```

Accessing individual elements in the time series is not hard:

- STRLEN can be used in order to obtain the number of samples.
- GETRANGE allows for random access of elements. If our time series have associated time information we can easily implement a binary search to get range combining GETRANGE with the Lua scripting engine available in Redis 2.6.
- SETRANGE can be used to overwrite an existing time series.

The limitation of this pattern is that we are forced into an append-only mode of operation, there is no way to cut the time series to a given size easily because Redis currently lacks a command able to trim string objects. However the space efficiency of time series stored in this way is remarkable.

Hint: it is possible to switch to a different key based on the current Unix time, in this way it is possible to have just a relatively small amount of samples per key, to avoid dealing with very big keys, and to make this pattern more friendly to be distributed across many Redis instances.

An example sampling the temperature of a sensor using fixed-size strings (using a binary format is better in real implementations).

redis> APPEND ts "0043"

```
(integer) 4
```

redis> APPEND ts "0035"

```
(integer) 8
```

redis> GETRANGE ts 0 3

```
"0043"
```

redis> GETRANGE ts 4 7

```
"0035"
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by 

AUTH password

Related commands

- **AUTH**
- ECHO
- PING
- QUIT
- SELECT

Available since 1.0.0.

Request for authentication in a password-protected Redis server. Redis can be instructed to require a password before allowing clients to execute commands. This is done using the `requirepass` directive in the configuration file.

If `password` matches the password in the configuration file, the server replies with the `OK` status code and starts accepting commands. Otherwise, an error is returned and the clients needs to try a new password.

Note: because of the high performance nature of Redis, it is possible to try a lot of passwords in parallel in very short time, so make sure to generate a strong and very long password so that this attack is infeasible.

Return value

Status code reply

BGREWRITEAOF

Related commands

- **BGREWRITEAOF**
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- INFO
- LASTSAVE
- MONITOR
- SAVE
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- SYNC
- TIME

Available since 1.0.0.

Instruct Redis to start an Append Only File rewrite process. The rewrite will create a small optimized version of the current Append Only File.

If BGREWRITEAOF fails, no data gets lost as the old AOF will be untouched.

The rewrite will be only triggered by Redis if there is not already a background process doing persistence. Specifically:

- If a Redis child is creating a snapshot on disk, the AOF rewrite is *scheduled* but not started until the saving child producing the RDB file terminates. In this case the BGREWRITEAOF will still return an OK code, but with an appropriate message. You can check if an AOF rewrite is scheduled looking at the INFO command as of Redis 2.6.
- If an AOF rewrite is already in progress the command returns an error and no AOF rewrite will be scheduled for a later time.

Since Redis 2.4 the AOF rewrite is automatically triggered by Redis, however the BGREWRITEAOF command can be used to trigger a rewrite at any time.

Please refer to the persistence documentation for detailed information.

Return value

Status code reply: always `OK`.

BGSAVE

Related commands

- BGREWRITEAOF
- **BGSAVE**
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- INFO
- LASTSAVE
- MONITOR
- SAVE
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- SYNC
- TIME

Available since 1.0.0.

Save the DB in background. The OK code is immediately returned. Redis forks, the parent continues to serve the clients, the child saves the DB on disk then exits. A client my be able to check if the operation succeeded using the LASTSAVE command.

Please refer to the persistence documentation for detailed information.

Return value

Status code reply

BITCOUNT key [start] [end]

Related commands

- APPEND
- **BITCOUNT**
- BITOP
- DECR
- DECRBY
- GET
- GETBIT
- GETRANGE
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- PSETEX
- SET
- SETBIT
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 2.6.0.

Time complexity: O(N)

Count the number of set bits (population counting) in a string.

By default all the bytes contained in the string are examined. It is possible to specify the counting operation only in an interval passing the additional arguments *start* and *end*.

Like for the GETRANGE command start and end can contain negative values in order to index bytes starting from the end of the string, where -1 is the last byte, -2 is the penultimate, and so forth.

Non-existent keys are treated as empty strings, so the command will return zero.

Return value

Integer reply

The number of bits set to 1.

Examples

```
redis> SET mykey "foobar"
OK
redis> BITCOUNT mykey
(integer) 26
redis> BITCOUNT mykey 0 0
(integer) 4
redis> BITCOUNT mykey 1 1
(integer) 6
redis>
```

Pattern: real-time metrics using bitmaps

Bitmaps are a very space-efficient representation of certain kinds of information. One example is a Web application that needs the history of user visits, so that for instance it is possible to determine what users are good targets of beta features.

Using the SETBIT command this is trivial to accomplish, identifying every day with a small progressive integer. For instance day 0 is the first day the application was put online, day 1 the next day, and so forth.

Every time an user performs a page view, the application can register that in the current day the user visited the web site using the SETBIT command setting the bit corresponding to the current day.

Later it will be trivial to know the number of single days the user visited the web site simply calling the BITCOUNT command against the bitmap.

A similar pattern where user IDs are used instead of days is described in the article called "Fast easy realtime metrics using Redis bitmaps".

Performance considerations

In the above example of counting days, even after 10 years the application is online we still have just `365*10` bits of data per user, that is just 456 bytes per user. With this amount of data BITCOUNT is still as fast as any other O(1) Redis command like GET or INCR.

When the bitmap is big, there are two alternatives:

- Taking a separated key that is incremented every time the bitmap is modified. This can be very efficient and atomic using a small Redis Lua script.
- Running the bitmap incrementally using the BITCOUNT *start* and *end* optional parameters, accumulating the results client-side, and optionally caching the result into a key.

BITOP operation destkey key [key ...]

Related commands

- APPEND
- BITCOUNT
- **BITOP**
- DECR
- DECRBY
- GET
- GETBIT
- GETRANGE
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- PSETEX
- SET
- SETBIT
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 2.6.0.

Time complexity: O(N)

Perform a bitwise operation between multiple keys (containing string values) and store the result in the destination key.

The BITOP command supports four bitwise operations: **AND**, **OR**, **XOR** and **NOT**, thus the valid forms to call the command are:

- BITOP AND *destkey srckey1 srckey2 srckey3 ... srckeyN*
- BITOP OR *destkey srckey1 srckey2 srckey3 ... srckeyN*
- BITOP XOR *destkey srckey1 srckey2 srckey3 ... srckeyN*
- BITOP NOT *destkey srckey*

As you can see **NOT** is special as it only takes an input key, because it performs inversion of bits so it only makes sense as an unary operator.

The result of the operation is always stored at *destkey*.

Handling of strings with different lengths

When an operation is performed between strings having different lengths, all the strings shorter than the longest string in the set are treated as if they were zero-padded up to the length of the longest string.

The same holds true for non-existent keys, that are considered as a stream of zero bytes up to the length of the longest string.

Return value

Integer reply

The size of the string stored in the destination key, that is equal to the size of the longest input string.

Examples

redis> SET key1 "foobar"

```
OK
```

redis> SET key2 "abcdef"

```
OK
```

redis> BITOP AND dest key1 key2

```
ERR Don't know what to do for "bitop"
```

redis> GET dest

```
(nil)
```

redis>

Pattern: real time metrics using bitmaps

BITOP is a good complement to the pattern documented in the BITCOUNT command documentation. Different bitmaps can be combined in order to obtain a target bitmap where the population counting operation is performed.

See the article called "Fast easy realtime metrics using Redis bitmaps" for a interesting use cases.

Performance considerations

BITOP is a potentially slow command as it runs in O(N) time. Care should be taken when running it against long input strings.

For real-time metrics and statistics involving large inputs a good approach is to use a slave (with read-only option disabled) where the bit-wise operations are performed to avoid blocking the master instance.

BLPOP key [key ...] timeout

Related commands

- **BLPOP**
- BRPOP
- BRPOPLPUSH
- LINDEX
- LINSERT
- LLEN
- LPOP
- LPUSH
- LPUSHX
- LRANGE
- LREM
- LSET
- LTRIM
- RPOP
- RPOPLPUSH
- RPUSH
- RPUSHX

Available since 2.0.0.

Time complexity: O(1)

BLPOP is a blocking list pop primitive. It is the blocking version of LPOP because it blocks the connection when there are no elements to pop from any of the given lists. An element is popped from the head of the first list that is non-empty, with the given keys being checked in the order that they are given.

Non-blocking behavior

When BLPOP is called, if at least one of the specified keys contains a non-empty list, an element is popped from the head of the list and returned to the caller together with the `key` it was popped from.

Keys are checked in the order that they are given. Let's say that the key `list1` doesn't exist and `list2` and `list3` hold non-empty lists. Consider the following command:

```
BLPOP list1 list2 list3 0
```

BLPOP guarantees to return an element from the list stored at `list2` (since it is the first non empty list when checking `list1`, `list2` and `list3` in that order).

Blocking behavior

If none of the specified keys exist, BLPOP blocks the connection until another client performs an LPUSH or RPUSH operation against one of the keys.

Once new data is present on one of the lists, the client returns with the name of the key unblocking it and the popped value.

When BLPOP causes a client to block and a non-zero timeout is specified, the client will unblock returning a `nil` multi-bulk value when the specified timeout has expired without a push operation against at least one of the specified keys.

The timeout argument is interpreted as an integer value specifying the maximum number of seconds to block. A timeout of zero can be used to block indefinitely.

What key is served first? What client? What element? Priority ordering details.

- If the client tries to blocks for multiple keys, but at least one key contains elements, the returned key / element pair is the first key from left to right that has one or more elements. In this case the client is not blocked. So for instance `BLPOP key1 key2 key3 key4 0`, assuming that both `key2` and `key4` are non-empty, will always return an element from `key2`.
- If multiple clients are blocked for the same key, the first client to be served is the one that was waiting for more time (the first that blocked for the key). Once a client is unblocked it does not retain any priority, when it blocks again with the next call to BLPOP it will be served accordingly to the number of clients already blocked for the same key, that will all be served before it (from the first to the last that blocked).
- When a client is blocking for multiple keys at the same time, and elements are available at the same time in multiple keys (because of a transaction or a Lua script added elements to multiple lists), the client will be unblocked using the first key that received a push operation (assuming it has enough elements to serve our client, as there may be other clients as well waiting for this key). Basically after the execution of every command Redis will run a list of all the keys that received data AND that have at least a client blocked. The list is ordered by new element arrival time, from the first key that received data to the last. For every key processed, Redis will serve all the clients waiting for that key in a FIFO fashion, as long as there are elements in this key. When the key is empty or there are no longer clients waiting for this key, the next key that received new data in the previous command / transaction / script is processed, and so forth.

Behavior of `BLPOP` when multiple elements are pushed inside a list.

There are times when a list can receive multiple elements in the context of the same conceptual command:

- Variadic push operations such as `LPUSH mylist a b c`.
- After an EXEC of a MULTI block with multiple push operations against the same list.
- Executing a Lua Script with Redis 2.6 or newer.

When multiple elements are pushed inside a list where there are clients blocking, the behavior is different for Redis 2.4 and Redis 2.6 or newer.

For Redis 2.6 what happens is that the command performing multiple pushes is executed, and *only after* the execution of the command the blocked clients are served. Consider this sequence of commands.

```
Client A:   BLPOP foo 0
```

If the above condition happens using a Redis 2.6 server or greater, Client **A** will be served with the `c` element, because after the LPUSH command the list contains `c,b,a`, so taking an element from the left means to return `c`.

Instead Redis 2.4 works in a different way: clients are served *in the context* of the push operation, so as long as `LPUSH foo a b c` starts pushing the first element to the list, it will be delivered to the Client **A**, that will receive `a` (the first element pushed).

The behavior of Redis 2.4 creates a lot of problems when replicating or persisting data into the AOF file, so the much more generic and semantically simpler behaviour was introduced into Redis 2.6 to prevent problems.

Note that for the same reason a Lua script or a `MULTI/EXEC` block may push elements into a list and afterward **delete the list**. In this case the blocked clients will not be served at all and will continue to be blocked as long as no data is present on the list after the execution of a single command, transaction, or script.

`BLPOP` inside a `MULTI` / `EXEC` transaction

BLPOP can be used with pipelining (sending multiple commands and reading the replies in batch), however this setup makes sense almost solely when it is the last command of the pipeline.

Using BLPOP inside a MULTI / EXEC block does not make a lot of sense as it would require blocking the entire server in order to execute the block atomically, which in turn does not allow other clients to perform a push operation. For this reason the behavior of BLPOP inside MULTI / EXEC when the list is empty is to return a `nil` multi-bulk reply, which is the same thing that happens when the timeout is reached.

If you like science fiction, think of time flowing at infinite speed inside a MULTI / EXEC block...

Return value

Multi-bulk reply: specifically:

- A `nil` multi-bulk when no element could be popped and the timeout expired.
- A two-element multi-bulk with the first element being the name of the key where an element was popped and the second element being the value of the popped element.

Examples

```
redis> DEL list1 list2
```

Reliable queues

When BLPOP returns an element to the client, it also removes the element from the list. This means that the element only exists in the context of the client: if the client crashes while processing the returned element, it is lost forever.

This can be a problem with some application where we want a more reliable messaging system. When this is the case, please check the BRPOPLPUSH command, that is a variant of BLPOP that adds the returned element to a traget list before returing it to the client.

Pattern: Event notification

Using blocking list operations it is possible to mount different blocking primitives. For instance for some application you may need to block waiting for elements into a Redis Set, so that as far as a new element is added to the Set, it is possible to retrieve it without resort to polling. This would require a blocking version of SPOP that is not available, but using blocking list operations we can easily accomplish this task.

The consumer will do:

```
LOOP forever
```

While in the producer side we'll use simply:

```
MULTI
```

BRPOP key [key ...] timeout

Related commands

- BLPOP
- **BRPOP**
- BRPOPLPUSH
- LINDEX
- LINSERT
- LLEN
- LPOP
- LPUSH
- LPUSHX
- LRANGE
- LREM
- LSET
- LTRIM
- RPOP
- RPOPLPUSH
- RPUSH
- RPUSHX

Available since 2.0.0.

Time complexity: O(1)

BRPOP is a blocking list pop primitive. It is the blocking version of RPOP because it blocks the connection when there are no elements to pop from any of the given lists. An element is popped from the tail of the first list that is non-empty, with the given keys being checked in the order that they are given.

See the BLPOP documentation for the exact semantics, since BRPOP is identical to BLPOP with the only difference being that it pops elements from the tail of a list instead of popping from the head.

Return value

Multi-bulk reply: specifically:

- A `nil` multi-bulk when no element could be popped and the timeout expired.
- A two-element multi-bulk with the first element being the name of the key where an element was popped and the second element being the value of the popped element.

Examples

```
redis> DEL list1 list2
```

This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

BRPOPLPUSH source destination timeout

Related commands

- BLPOP
- BRPOP
- **BRPOPLPUSH**
- LINDEX
- LINSERT
- LLEN
- LPOP
- LPUSH
- LPUSHX
- LRANGE
- LREM
- LSET
- LTRIM
- RPOP
- RPOPLPUSH
- RPUSH
- RPUSHX

Available since 2.2.0.

Time complexity: O(1)

BRPOPLPUSH is the blocking variant of RPOPLPUSH. When `source` contains elements, this command behaves exactly like RPOPLPUSH. When `source` is empty, Redis will block the connection until another client pushes to it or until `timeout` is reached. A `timeout` of zero can be used to block indefinitely.

See RPOPLPUSH for more information.

Return value

Bulk reply: the element being popped from `source` and pushed to `destination`. If `timeout` is reached, a Null multi-bulk reply is returned.

Pattern: Reliable queue

Please see the pattern description in the RPOPLPUSH documentation.

Pattern: Circular list

Please see the pattern description in the RPOPLPUSH documentation.

This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by 

CLIENT KILL ip:port

Related commands

- BGREWRITEAOF
- BGSAVE
- **CLIENT KILL**
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- INFO
- LASTSAVE
- MONITOR
- SAVE
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- SYNC
- TIME

Available since 2.4.0.

Time complexity: O(N) where N is the number of client connections

The `CLIENT KILL` command closes a given client connection identified by ip:port.

The ip:port should match a line returned by the `CLIENT LIST` command.

Due to the single-treaded nature of Redis, it is not possible to kill a client connection while it is executing a command. From the client point of view, the connection can never be closed in the middle of the execution of a command. However, the client will notice the connection has been closed only when the next command is sent (and results in network error).

Return value

Status code reply: `OK` if the connection exists and has been closed

CLIENT LIST

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- **CLIENT LIST**
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- INFO
- LASTSAVE
- MONITOR
- SAVE
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- SYNC
- TIME

Available since 2.4.0.

Time complexity: O(N) where N is the number of client connections

The `CLIENT LIST` command returns information and statistics about the client connections server in a mostly human readable format.

Return value

Bulk reply: a unique string, formatted as follows:

- One client connection per line (separated by LF)
- Each line is composed of a succession of property=value fields separated by a space character.

Here is the meaning of the fields:

- addr: address/port of the client
- fd: file descriptor corresponding to the socket
- age: total duration of the connection in seconds
- idle: idle time of the connection in seconds
- flags: client flags (see below)

- db: current database ID
- sub: number of channel subscriptions
- psub: number of pattern matching subscriptions
- multi: number of commands in a MULTI/EXEC context
- qbuf: query buffer length (0 means no query pending)
- qbuf-free: free space of the query buffer (0 means the buffer is full)
- obl: output buffer length
- oll: output list length (replies are queued in this list when the buffer is full)
- omem: output buffer memory usage
- events: file descriptor events (see below)
- cmd: last command played

The client flags can be a combination of:

```
O: the client is a slave in MONITOR mode
```

The file descriptor events can be:

```
r: the client socket is readable (event loop)
```

Notes

New fields are regularly added for debugging purpose. Some could be removed in the future. A version safe Redis client using this command should parse the output accordingly (i.e. handling gracefully missing fields, skipping unknown fields).

CONFIG GET parameter

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- **CONFIG GET**
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- INFO
- LASTSAVE
- MONITOR
- SAVE
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- SYNC
- TIME

Available since 2.0.0.

The `CONFIG GET` command is used to read the configuration parameters of a running Redis server. Not all the configuration parameters are supported in Redis 2.4, while Redis 2.6 can read the whole configuration of a server using this command.

The symmetric command used to alter the configuration at run time is `CONFIG SET`.

`CONFIG GET` takes a single argument, which is a glob-style pattern. All the configuration parameters matching this parameter are reported as a list of key-value pairs. Example:

```
redis> config get *max-*-entries*
```

You can obtain a list of all the supported configuration parameters by typing `CONFIG GET *` in an open `redis-cli` prompt.

All the supported parameters have the same meaning of the equivalent configuration parameter used in the redis.conf file, with the following important differences:

- Where bytes or other quantities are specified, it is not possible to use the `redis.conf` abbreviated form (10k 2gb ... and so forth), everything should be specified as a well-formed 64-bit integer, in the base unit of the configuration directive.

- The save parameter is a single string of space-separated integers. Every pair of integers represent a seconds/modifications threshold.

For instance what in `redis.conf` looks like:

```
save 900 1
```

that means, save after 900 seconds if there is at least 1 change to the dataset, and after 300 seconds if there are at least 10 changes to the datasets, will be reported by `CONFIG GET` as "900 1 300 10".

Return value

The return type of the command is a Bulk reply.

CONFIG RESETSTAT

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- **CONFIG RESETSTAT**
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- INFO
- LASTSAVE
- MONITOR
- SAVE
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- SYNC
- TIME

Available since 2.0.0.

Time complexity: O(1)

Resets the statistics reported by Redis using the INFO command.

These are the counters that are reset:

- Keyspace hits
- Keyspace misses
- Number of commands processed
- Number of connections received
- Number of expired keys
- Number of rejected connections
- Latest fork(2) time
- The `aof_delayed_fsync` counter

Return value

Status code reply: always `OK`.

Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

CONFIG SET parameter value

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- **CONFIG SET**
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- INFO
- LASTSAVE
- MONITOR
- SAVE
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- SYNC
- TIME

Available since 2.0.0.

The `CONFIG SET` command is used in order to reconfigure the server at run time without the need to restart Redis. You can change both trivial parameters or switch from one to another persistence option using this command.

The list of configuration parameters supported by `CONFIG SET` can be obtained issuing a `CONFIG GET *` command, that is the symmetrical command used to obtain information about the configuration of a running Redis instance.

All the configuration parameters set using `CONFIG SET` are immediately loaded by Redis and will take effect starting with the next command executed.

All the supported parameters have the same meaning of the equivalent configuration parameter used in the redis.conf file, with the following important differences:

- Where bytes or other quantities are specified, it is not possible to use the `redis.conf` abbreviated form (10k 2gb ... and so forth), everything should be specified as a well-formed 64-bit integer, in the base unit of the configuration directive.
- The save parameter is a single string of space-separated integers. Every pair of integers represent a seconds/modifications threshold.

For instance what in `redis.conf` looks like:

```
save 900 1
```

that means, save after 900 seconds if there is at least 1 change to the dataset, and after 300 seconds if there are at least 10 changes to the datasets, should be set using `CONFIG SET` as "900 1 300 10".

It is possible to switch persistence from RDB snapshotting to append-only file (and the other way around) using the `CONFIG SET` command. For more information about how to do that please check the persistence page.

In general what you should know is that setting the `appendonly` parameter to `yes` will start a background process to save the initial append-only file (obtained from the in memory data set), and will append all the subsequent commands on the append-only file, thus obtaining exactly the same effect of a Redis server that started with AOF turned on since the start.

You can have both the AOF enabled with RDB snapshotting if you want, the two options are not mutually exclusive.

Return value

Status code reply: `OK` when the configuration was set properly. Otherwise an error is returned.

DBSIZE

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- **DBSIZE**
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- INFO
- LASTSAVE
- MONITOR
- SAVE
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- SYNC
- TIME

Available since 1.0.0.

Return the number of keys in the currently-selected database.

Return value

Integer reply

DEBUG OBJECT key

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- **DEBUG OBJECT**
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- INFO
- LASTSAVE
- MONITOR
- SAVE
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- SYNC
- TIME

Available since 1.0.0.

`DEBUG OBJECT` is a debugging command that should not be used by clients. Check the OBJECT command instead.

Status code reply

DEBUG SEGFAULT

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- **DEBUG SEGFAULT**
- FLUSHALL
- FLUSHDB
- INFO
- LASTSAVE
- MONITOR
- SAVE
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- SYNC
- TIME

Available since 1.0.0.

`DEBUG SEGFAULT` performs an invalid memory access that crashes Redis. It is used to simulate bugs during the development.

Status code reply

DECR key

Related commands

- APPEND
- BITCOUNT
- BITOP
- **DECR**
- DECRBY
- GET
- GETBIT
- GETRANGE
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- PSETEX
- SET
- SETBIT
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 1.0.0.

Time complexity: O(1)

Decrements 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. This operation is limited to **64 bit signed integers**.

See INCR for extra information on increment/decrement operations.

Return value

Integer reply: the value of `key` after the decrement

Examples

redis> SET mykey "10"

```
OK
```

redis> DECR mykey

```
(integer) 9
```

redis> SET mykey "2342934823904809480293482309 48"

```
OK
```

redis> DECR mykey

```
ERR value is not an integer or out of range
```

redis>

DECRBY key decrement

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- **DECRBY**
- GET
- GETBIT
- GETRANGE
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- PSETEX
- SET
- SETBIT
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 1.0.0.

Time complexity: O(1)

Decrements the number stored at `key` by `decrement`. 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. This operation is limited to 64 bit signed integers.

See INCR for extra information on increment/decrement operations.

Return value

Integer reply: the value of `key` after the decrement

Examples

redis> SET mykey "10"

```
OK
```

```
redis> DECRBY mykey 5

(integer) 5

redis>
```

Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by 

DEL key [key ...]

Related commands

- **DEL**
- DUMP
- EXISTS
- EXPIRE
- EXPIREAT
- KEYS
- MIGRATE
- MOVE
- OBJECT
- PERSIST
- PEXPIRE
- PEXPIREAT
- PTTL
- RANDOMKEY
- RENAME
- RENAMENX
- RESTORE
- SORT
- TTL
- TYPE

Available since 1.0.0.

Time complexity: O(N) where N is the number of keys that will be removed. When a key to remove holds a value other than a string, the individual complexity for this key is O(M) where M is the number of elements in the list, set, sorted set or hash. Removing a single key that holds a string value is O(1).

Removes the specified keys. A key is ignored if it does not exist.

Return value

Integer reply: The number of keys that were removed.

Examples

redis> SET key1 "Hello"

```
OK
```

redis> SET key2 "World"

```
OK
```

redis> DEL key1 key2 key3

```
(integer) 2
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

DISCARD

Related topics

- Transactions

Related commands

- **DISCARD**
- EXEC
- MULTI
- UNWATCH
- WATCH

Available since 2.0.0.

Flushes all previously queued commands in a transaction and restores the connection state to normal.

If WATCH was used, DISCARD unwatches all keys.

Return value

Status code reply: always `OK`.

DUMP key

Related commands

- DEL
- **DUMP**
- EXISTS
- EXPIRE
- EXPIREAT
- KEYS
- MIGRATE
- MOVE
- OBJECT
- PERSIST
- PEXPIRE
- PEXPIREAT
- PTTL
- RANDOMKEY
- RENAME
- RENAMENX
- RESTORE
- SORT
- TTL
- TYPE

Available since 2.6.0.

Time complexity: O(1) to access the key and additional O(N*M) to serialized it, where N is the number of Redis objects composing the value and M their average size. For small string values the time complexity is thus O(1)+O(1*M) where M is small, so simply O(1).

Serialize the value stored at key in a Redis-specific format and return it to the user. The returned value can be synthesized back into a Redis key using the RESTORE command.

The serialization format is opaque and non-standard, however it has a few semantical characteristics:

- It contains a 64-bit checksum that is used to make sure errors will be detected. The RESTORE command makes sure to check the checksum before synthesizing a key using the serialized value.
- Values are encoded in the same format used by RDB.
- An RDB version is encoded inside the serialized value, so that different Redis versions with incompatible RDB formats will refuse to process the serialized value.

The serialized value does NOT contain expire information. In order to capture the time to live of the current value the PTTL command should be used.

If `key` does not exist a nil bulk reply is returned.

Return value

Bulk reply: the serialized value.

Examples

redis> SET mykey 10

```
OK
```

redis> DUMP mykey

```
"\u0000\xC0\n\u0006\u0000\xF8r?\xC5\xFB\xFB_("
```

redis>

ECHO message

Related commands

- AUTH
- **ECHO**
- PING
- QUIT
- SELECT

Available since 1.0.0.

Returns `message`.

Return value

Bulk reply

Examples

redis> ECHO "Hello World!"

```
"Hello World!"
```

redis>

EVAL script numkeys key [key ...] arg [arg ...]

Related commands

- **EVAL**
- EVALSHA
- SCRIPT EXISTS
- SCRIPT FLUSH
- SCRIPT KILL
- SCRIPT LOAD

Available since 2.6.0.

Time complexity: Depends on the script that is executed.

Introduction to EVAL

EVAL and EVALSHA are used to evaluate scripts using the Lua interpreter built into Redis starting from version 2.6.0.

The first argument of EVAL is a Lua 5.1 script. The script does not need to define a Lua function (and should not). It is just a Lua program that will run in the context of the Redis server.

The second argument of EVAL is the number of arguments that follows the script (starting from the third argument) that represent Redis key names. This arguments can be accessed by Lua using the KEYS global variable in the form of a one-based array (so `KEYS[1]`, `KEYS[2]`, ...).

All the additional arguments should not represent key names and can be accessed by Lua using the `ARGV` global variable, very similarly to what happens with keys (so `ARGV[1]`, `ARGV[2]`, ...).

The following example should clarify what stated above:

```
> eval "return {KEYS[1],KEYS[2],ARGV[1],ARGV[2]}" 2 key1 key2 first second
```

Note: as you can see Lua arrays are returned as Redis multi bulk replies, that is a Redis return type that your client library will likely convert into an Array type in your programming language.

It is possible to call Redis commands from a Lua script using two different Lua functions:

- `redis.call()`
- `redis.pcall()`

`redis.call()` is similar to `redis.pcall()`, the only difference is that if a Redis command call will result into an error, `redis.call()` will raise a Lua error that in turn will force EVAL to return an error to the command caller, while `redis.pcall` will trap the error returning a Lua table representing the error.

The arguments of the `redis.call()` and `redis.pcall()` functions are simply all the arguments of a well formed Redis command:

```
> eval "return redis.call('set','foo','bar')" 0
```

The above script actually sets the key `foo` to the string `bar`. However it violates the EVAL command semantics as all the keys that the script uses should be passed using the KEYS array, in the following way:

```
> eval "return redis.call('set',KEYS[1],'bar')" 1 foo
```

The reason for passing keys in the proper way is that, before EVAL all the Redis commands could be analyzed before execution in order to establish what keys the command will operate on.

In order for this to be true for EVAL also keys must be explicit. This is useful in many ways, but especially in order to make sure Redis Cluster is able to forward your request to the appropriate cluster node (Redis Cluster is a work in progress, but the scripting feature was designed in order to play well with it). However this rule is not enforced in order to provide the user with opportunities to abuse the Redis single instance configuration, at the cost of writing scripts not compatible with Redis Cluster.

Lua scripts can return a value, that is converted from the Lua type to the Redis protocol using a set of conversion rules.

Conversion between Lua and Redis data types

Redis return values are converted into Lua data types when Lua calls a Redis command using call() or pcall(). Similarly Lua data types are converted into the Redis protocol when a Lua script returns a value, so that scripts can control what EVAL will return to the client.

This conversion between data types is designed in a way that if a Redis type is converted into a Lua type, and then the result is converted back into a Redis type, the result is the same as of the initial value.

In other words there is a one-to-one conversion between Lua and Redis types. The following table shows you all the conversions rules:

Redis to Lua conversion table.

- Redis integer reply -> Lua number
- Redis bulk reply -> Lua string
- Redis multi bulk reply -> Lua table (may have other Redis data types nested)
- Redis status reply -> Lua table with a single `ok` field containing the status
- Redis error reply -> Lua table with a single `err` field containing the error
- Redis Nil bulk reply and Nil multi bulk reply -> Lua false boolean type

Lua to Redis conversion table.

- Lua number -> Redis integer reply (the number is converted into an integer)
- Lua string -> Redis bulk reply
- Lua table (array) -> Redis multi bulk reply (truncated to the first nil inside the Lua array if any)
- Lua table with a single `ok` field -> Redis status reply
- Lua table with a single `err` field -> Redis error reply
- Lua boolean false -> Redis Nil bulk reply.

There is an additional Lua-to-Redis conversion rule that has no corresponding Redis to Lua conversion rule:

- Lua boolean true -> Redis integer reply with value of 1.

Also there are two important rules to note:

- Lua has a single numerical type, Lua numbers. There is no distinction between integers and floats. So we always convert Lua numbers into integer replies, removing the decimal part of the number if any. **If you want to return a float from Lua you should return it as a string**, exactly like Redis itself does (see for instance the ZSCORE command).
- There is no simple way to have nils inside Lua arrays, this is a result of Lua table semantics, so when Redis converts a Lua array into Redis protocol the conversion is stopped if a nil is encountered.

Here are a few conversion examples:

```
> eval "return 10" 0
```

The last example shows how it is possible to receive the exact return value of `redis.call()` or `redis.pcall()` from Lua that would be returned if the command was called directly.

In the following example we can see how floats and arrays with nils are handled:

```
> eval "return {1,2,3.3333,'foo',nil,'bar'}" 0
```

As you can see 3.333 is converted into 3, and the *bar* string is never returned as there is a nil before.

Helper functions to return Redis types

There are two helper functions to return Redis types from Lua.

- `redis.error_reply(error_string)` returns an error reply. This function simply returns the single field table with the `err` field set to the specified string for you.
- `redis.status_reply(status_string)` returns a status reply. This function simply returns the single field table with the `ok` field set to the specified string for you.

There is no difference between using the helper functions or directly returning the table with the specified format, so the following two forms are equivalent:

```
return {err="My Error"}
```

Atomicity of scripts

Redis uses the same Lua interpreter to run all the commands. Also Redis guarantees that a script is executed in an atomic way: no other script or Redis command will be executed while a script is being executed. This semantics is very similar to the one of MULTI / EXEC. From the point of view of all the other clients the effects of a script are either still not visible or already completed.

However this also means that executing slow scripts is not a good idea. It is not hard to create fast scripts, as the script overhead is very low, but if you are going to use slow scripts you should be aware that while the script is running no other client can execute commands since the server is busy.

Error handling

As already stated, calls to `redis.call()` resulting in a Redis command error will stop the execution of the script and will return the error, in a way that makes it obvious that the error was generated by a script:

```
> del foo
```

Using the `redis.pcall()` command no error is raised, but an error object is returned in the format specified above (as a Lua table with an `err` field). The script can pass the exact error to the user by returning the error object returned by `redis.pcall()`.

Bandwidth and EVALSHA

The EVAL command forces you to send the script body again and again. Redis does not need to recompile the script every time as it uses an internal caching mechanism, however paying the cost of the additional bandwidth may not be optimal in many contexts.

On the other hand, defining commands using a special command or via `redis.conf` would be a problem for a few reasons:

- Different instances may have different versions of a command implementation.
- Deployment is hard if there is to make sure all the instances contain a given command, especially in a distributed environment.
- Reading an application code the full semantic could not be clear since the application would call commands defined server side.

In order to avoid these problems while avoiding the bandwidth penalty, Redis implements the EVALSHA command.

EVALSHA works exactly like EVAL, but instead of having a script as the first argument it has the SHA1 digest of a script. The behavior is the following:

- If the server still remembers a script with a matching SHA1 digest, the script is executed.
- If the server does not remember a script with this SHA1 digest, a special error is returned telling the client to use EVAL instead.

Example:

```
> set foo bar
```

The client library implementation can always optimistically send EVALSHA under the hood even when the client actually calls EVAL, in the hope the script was already seen by the server. If the `NOSCRIPT` error is returned EVAL will be used instead.

Passing keys and arguments as additional EVAL arguments is also very useful in this context as the script string remains constant and can be efficiently cached by Redis.

Script cache semantics

Executed scripts are guaranteed to be in the script cache **forever**. This means that if an EVAL is performed against a Redis instance all the subsequent EVALSHA calls will succeed.

The only way to flush the script cache is by explicitly calling the SCRIPT FLUSH command, which will *completely flush* the scripts cache removing all the scripts executed so far. This is usually needed only when the instance is going to be instantiated for another customer or application in a cloud environment.

The reason why scripts can be cached for long time is that it is unlikely for a well written application to have enough different scripts to cause memory problems. Every script is conceptually like the implementation of a new command, and even a large application will likely have just a few hundred of them. Even if the application is modified many times and scripts will change, the memory used is negligible.

The fact that the user can count on Redis not removing scripts is semantically a very good thing. For instance an application with a persistent connection to Redis can be sure that if a script was sent once it is still in memory, so EVALSHA can be used against those scripts in a pipeline without the chance of an error being generated due to an unknown script (we'll see this problem in detail later).

The SCRIPT command

Redis offers a SCRIPT command that can be used in order to control the scripting subsystem. SCRIPT currently accepts three different commands:

- SCRIPT FLUSH. This command is the only way to force Redis to flush the scripts cache. It is most useful in a cloud environment where the same instance can be reassigned to a different user. It is also useful for testing client libraries' implementations of the scripting feature.
- SCRIPT EXISTS *sha1 sha2... shaN*. Given a list of SHA1 digests as arguments this command returns an array of 1 or 0, where 1 means the specific SHA1 is recognized as a script already present in the scripting cache, while 0 means that a script with this SHA1 was never seen before (or at least never seen after the latest SCRIPT FLUSH command).
- SCRIPT LOAD *script*. This command registers the specified script in the Redis script cache. The command is useful in all the contexts where we want to make sure that EVALSHA will not fail (for instance during a pipeline or MULTI/EXEC operation), without the need to actually execute the script.
- SCRIPT KILL. This command is the only way to interrupt a long-running script that reaches the configured maximum execution time for scripts. The SCRIPT KILL command can only be used with scripts that did not modify the dataset during their execution (since stopping a read-only script does not violate the scripting engine's guaranteed atomicity). See the next sections for more information about long running scripts.

Scripts as pure functions

A very important part of scripting is writing scripts that are pure functions. Scripts executed in a Redis instance are replicated on slaves by sending the script -- not the resulting commands. The same happens for the Append Only File. The reason is that sending a script to another Redis instance is much faster than sending the multiple commands the script generates, so if the client is sending many scripts to the master, converting the scripts into individual commands for the slave / AOF would result in too much bandwidth for the replication link or the Append Only File (and also too much CPU since dispatching a command received

via network is a lot more work for Redis compared to dispatching a command invoked by Lua scripts).

The only drawback with this approach is that scripts are required to have the following property:

- The script always evaluates the same Redis *write* commands with the same arguments given the same input data set. Operations performed by the script cannot depend on any hidden (non-explicit) information or state that may change as script execution proceeds or between different executions of the script, nor can it depend on any external input from I/O devices.

Things like using the system time, calling Redis random commands like RANDOMKEY, or using Lua random number generator, could result into scripts that will not always evaluate in the same way.

In order to enforce this behavior in scripts Redis does the following:

- Lua does not export commands to access the system time or other external state.
- Redis will block the script with an error if a script calls a Redis command able to alter the data set **after** a Redis *random* command like RANDOMKEY, SRANDMEMBER, TIME. This means that if a script is read-only and does not modify the data set it is free to call those commands. Note that a *random command* does not necessarily mean a command that uses random numbers: any non-deterministic command is considered a random command (the best example in this regard is the TIME command).
- Redis commands that may return elements in random order, like SMEMBERS (because Redis Sets are *unordered*) have a different behavior when called from Lua, and undergo a silent lexicographical sorting filter before returning data to Lua scripts. So `redis.call("smembers",KEYS[1])` will always return the Set elements in the same order, while the same command invoked from normal clients may return different results even if the key contains exactly the same elements.
- Lua pseudo random number generation functions `math.random` and `math.randomseed` are modified in order to always have the same seed every time a new script is executed. This means that calling `math.random` will always generate the same sequence of numbers every time a script is executed if `math.randomseed` is not used.

However the user is still able to write commands with random behavior using the following simple trick. Imagine I want to write a Redis script that will populate a list with N random integers.

I can start with this small Ruby program:

```
require 'rubygems'
```

Every time this script executed the resulting list will have exactly the following elements:

```
> lrange mylist 0 -1
```

In order to make it a pure function, but still be sure that every invocation of the script will result in different random elements, we can simply add an additional argument to the script that will be used in order to seed the Lua pseudo-random number generator. The new script is as follows:

```
RandomPushScript = <<EOF
```

What we are doing here is sending the seed of the PRNG as one of the arguments. This way the script output will be the same given the same arguments, but we are changing one of the arguments in every invocation, generating the random seed client-side. The seed will be propagated as one of the arguments both in the

replication link and in the Append Only File, guaranteeing that the same changes will be generated when the AOF is reloaded or when the slave processes the script.

Note: an important part of this behavior is that the PRNG that Redis implements as `math.random` and `math.randomseed` is guaranteed to have the same output regardless of the architecture of the system running Redis. 32-bit, 64-bit, big-endian and little-endian systems will all produce the same output.

Global variables protection

Redis scripts are not allowed to create global variables, in order to avoid leaking data into the Lua state. If a script needs to maintain state between calls (a pretty uncommon need) it should use Redis keys instead.

When global variable access is attempted the script is terminated and EVAL returns with an error:

```
redis 127.0.0.1:6379> eval 'a=10' 0
```

Accessing a *non existing* global variable generates a similar error.

Using Lua debugging functionality or other approaches like altering the meta table used to implement global protections in order to circumvent globals protection is not hard. However it is difficult to do it accidentally. If the user messes with the Lua global state, the consistency of AOF and replication is not guaranteed: don't do it.

Note for Lua newbies: in order to avoid using global variables in your scripts simply declare every variable you are going to use using the *local* keyword.

Available libraries

The Redis Lua interpreter loads the following Lua libraries:

- base lib.
- table lib.
- string lib.
- math lib.
- debug lib.
- cjson lib.
- cmsgpack lib.

Every Redis instance is *guaranteed* to have all the above libraries so you can be sure that the environment for your Redis scripts is always the same.

The CJSON library provides extremely fast JSON maniplation within Lua. All the other libraries are standard Lua libraries.

Emitting Redis logs from scripts

It is possible to write to the Redis log file from Lua scripts using the `redis.log` function.

```
redis.log(loglevel,message)
```

`loglevel` is one of:

- `redis.LOG_DEBUG`
- `redis.LOG_VERBOSE`
- `redis.LOG_NOTICE`
- `redis.LOG_WARNING`

They correspond directly to the normal Redis log levels. Only logs emitted by scripting using a log level that is equal or greater than the currently configured Redis instance log level will be emitted.

The `message` argument is simply a string. Example:

```
redis.log(redis.LOG_WARNING,"Something is wrong with this script.")
```

Will generate the following:

```
[32343] 22 Mar 15:21:39 # Something is wrong with this script.
```

Sandbox and maximum execution time

Scripts should never try to access the external system, like the file system or any other system call. A script should only operate on Redis data and passed arguments.

Scripts are also subject to a maximum execution time (five seconds by default). This default timeout is huge since a script should usually run in under a millisecond. The limit is mostly to handle accidental infinite loops created during development.

It is possible to modify the maximum time a script can be executed with millisecond precision, either via `redis.conf` or using the CONFIG GET / CONFIG SET command. The configuration parameter affecting max execution time is called `lua-time-limit`.

When a script reaches the timeout it is not automatically terminated by Redis since this violates the contract Redis has with the scripting engine to ensure that scripts are atomic. Interrupting a script means potentially leaving the dataset with half-written data. For this reasons when a script executes for more than the specified time the following happens:

- Redis logs that a script is running too long.
- It starts accepting commands again from other clients, but will reply with a BUSY error to all the clients sending normal commands. The only allowed commands in this status are `SCRIPT KILL` and `SHUTDOWN NOSAVE`.
- It is possible to terminate a script that executes only read-only commands using the `SCRIPT KILL` command. This does not violate the scripting semantic as no data was yet written to the dataset by the script.
- If the script already called write commands the only allowed command becomes `SHUTDOWN NOSAVE` that stops the server without saving the current data set on disk (basically the server is aborted).

EVALSHA in the context of pipelining

Care should be taken when executing EVALSHA in the context of a pipelined request, since even in a pipeline the order of execution of commands must be guaranteed. If EVALSHA will return a `NOSCRIPT` error the command can not be reissued later otherwise the order of execution is violated.

The client library implementation should take one of the following approaches:

- Always use plain EVAL when in the context of a pipeline.
- Accumulate all the commands to send into the pipeline, then check for EVAL commands and use the `SCRIPT EXISTS` command to check if all the scripts are already defined. If not, add `SCRIPT LOAD` commands on top of the pipeline as required, and use EVALSHA for all the EVAL calls.

EVALSHA sha1 numkeys key [key ...] arg [arg ...]

Related commands

- EVAL
- **EVALSHA**
- SCRIPT EXISTS
- SCRIPT FLUSH
- SCRIPT KILL
- SCRIPT LOAD

Available since 2.6.0.

Time complexity: Depends on the script that is executed.

Evaluates a script cached on the server side by its SHA1 digest. Scripts are cached on the server side using the `SCRIPT LOAD` command. The command is otherwise identical to EVAL.

EXEC

Related topics

- Transactions

Related commands

- DISCARD
- **EXEC**
- MULTI
- UNWATCH
- WATCH

Available since 1.2.0.

Executes all previously queued commands in a transaction and restores the connection state to normal.

When using WATCH, EXEC will execute commands only if the watched keys were not modified, allowing for a check-and-set mechanism.

Return value

Multi-bulk reply: each element being the reply to each of the commands in the atomic transaction.

When using WATCH, EXEC can return a Null multi-bulk reply if the execution was aborted.

EXISTS key

Related commands

- DEL
- DUMP
- **EXISTS**
- EXPIRE
- EXPIREAT
- KEYS
- MIGRATE
- MOVE
- OBJECT
- PERSIST
- PEXPIRE
- PEXPIREAT
- PTTL
- RANDOMKEY
- RENAME
- RENAMENX
- RESTORE
- SORT
- TTL
- TYPE

Available since 1.0.0.

Time complexity: O(1)

Returns if `key` exists.

Return value

Integer reply, specifically:

- `1` if the key exists.
- `0` if the key does not exist.

Examples

redis> SET key1 "Hello"

```
OK
```

redis> EXISTS key1

```
(integer) 1
```

redis> EXISTS key2

```
(integer) 0
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

EXPIRE key seconds

Related commands

- DEL
- DUMP
- EXISTS
- **EXPIRE**
- EXPIREAT
- KEYS
- MIGRATE
- MOVE
- OBJECT
- PERSIST
- PEXPIRE
- PEXPIREAT
- PTTL
- RANDOMKEY
- RENAME
- RENAMENX
- RESTORE
- SORT
- TTL
- TYPE

Available since 1.0.0.

Time complexity: O(1)

Set a timeout on `key`. After the timeout has expired, the key will automatically be deleted. A key with an associated timeout is often said to be *volatile* in Redis terminology.

The timeout is cleared only when the key is removed using the DEL command or overwritten using the SET or GETSET commands. This means that all the operations that conceptually *alter* the value stored at the key without replacing it with a new one will leave the timeout untouched. For instance, incrementing the value of a key with INCR, pushing a new value into a list with LPUSH, or altering the field value of a hash with HSET are all operations that will leave the timeout untouched.

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.

If a key is overwritten by RENAME, like in the case of an existing key `Key_A` that is overwritten by a call like `RENAME Key_B Key_A`, it does not matter if the original `Key_A` had a timeout associated or not, the new key `Key_A` will inherit all the characteristics of `Key_B`.

Refreshing expires

It is possible to call EXPIRE using as argument a key that already has an existing expire set. In this case the time to live of a key is *updated* to the new value. There are many useful applications for this, an example is documented in the *Navigation session* pattern section below.

Differences in Redis prior 2.1.3

In Redis versions prior **2.1.3** altering a key with an expire set using a command altering its value had the effect of removing the key entirely. This semantics was needed because of limitations in the replication layer that are now fixed.

Return value

Integer reply, specifically:

- `1` if the timeout was set.
- `0` if `key` does not exist or the timeout could not be set.

Examples

redis> SET mykey "Hello"

```
OK
```

redis> EXPIRE mykey 10

```
(integer) 1
```

redis> TTL mykey

```
(integer) 10
```

redis> SET mykey "Hello World"

```
OK
```

redis> TTL mykey

```
(integer) -1
```

redis>

Pattern: Navigation session

Imagine you have a web service and you are interested in the latest N pages *recently* visited by your users, such that each adiacent page view was not performed more than 60 seconds after the previous. Conceptually you may think at this set of page views as a *Navigation session* if your user, that may contain interesting information about what kind of products he or she is looking for currently, so that you can recommend related products.

You can easily model this pattern in Redis using the following strategy: every time the user does a page view you call the following commands:

```
MULTI
```

If the user will be idle more than 60 seconds, the key will be deleted and only subsequent page views that have less than 60 seconds of difference will be recorded.

This pattern is easily modified to use counters using INCR instead of lists using RPUSH.

Appendix: Redis expires

Keys with an expire

Normally Redis keys are created without an associated time to live. The key will simply live forever, unless it is removed by the user in an explicit way, for instance using the DEL command.

The EXPIRE family of commands is able to associate an expire to a given key, at the cost of some additional memory used by the key. When a key has an expire set, Redis will make sure to remove the key when the specified amount of time elapsed.

The key time to live can be updated or entirely removed using the EXPIRE and PERSIST command (or other strictly related commands).

Expire accuracy

In Redis 2.4 the expire might not be pin-point accurate, and it could be between zero to one seconds out.

Since Redis 2.6 the expire error is from 0 to 1 milliseconds.

Expires and persistence

Keys expiring information is stored as absolute Unix timestamps (in milliseconds in case of Redis version 2.6 or greater). This means that the time is flowing even when the Redis instance is not active.

For expires to work well, the computer time must be taken stable. If you move an RDB file from two computers with a big desync in their clocks, funny things may happen (like all the keys loaded to be expired at loading time).

Even running instances will always check the computer clock, so for instance if you set a key with a time to live of 1000 seconds, and then set your computer time 2000 seconds in the future, the key will be expired immediately, instead of lasting for 1000 seconds.

How Redis expires keys

Redis keys are expired in two ways: a passive way, and an active way.

A key is actively expired simply when some client tries to access it, and the key is found to be timed out.

Of course this is not enough as there are expired keys that will never be accessed again. This keys should be expired anyway, so periodically Redis test a few keys at random among keys with an expire set. All the keys that are already expired are deleted from the keyspace.

Specifically this is what Redis does 10 times per second:

1. Test 100 random keys from the set of keys with an associated expire.
2. Delete all the keys found expired.
3. If more than 25 keys were expired, start again from step 1.

This is a trivial probabilistic algorithm, basically the assumption is that our sample is representative of the whole key space, and we continue to expire until the percentage of keys that are likely to be expired is under 25%

This means that at any given moment the maximum amount of keys already expired that are using memory is at max equal to max amount of write operations per second divided by 4.

How expires are handled in the replication link and AOF file

In order to obtain a correct behavior without sacrificing consistency, when a key expires, a DEL operation is synthesized in both the AOF file and gains all the attached slaves. This way the expiration process is centralized in the master instance, and there is no chance of consistency errors.

However while the slaves connected to a master will not expire keys independently (but will wait for the DEL coming from the master), they'll still take the full state of the expires existing in the dataset, so when a slave is elected to a master it will be able to expire the keys independently, fully acting as a master.

EXPIREAT key timestamp

Related commands

- DEL
- DUMP
- EXISTS
- EXPIRE
- **EXPIREAT**
- KEYS
- MIGRATE
- MOVE
- OBJECT
- PERSIST
- PEXPIRE
- PEXPIREAT
- PTTL
- RANDOMKEY
- RENAME
- RENAMENX
- RESTORE
- SORT
- TTL
- TYPE

Available since 1.2.0.

Time complexity: O(1)

EXPIREAT has the same effect and semantic as EXPIRE, but instead of specifying the number of seconds representing the TTL (time to live), it takes an absolute Unix timestamp (seconds since January 1, 1970).

Please for the specific semantics of the command refer to the documentation of EXPIRE.

Background

EXPIREAT was introduced in order to convert relative timeouts to absolute timeouts for the AOF persistence mode. Of course, it can be used directly to specify that a given key should expire at a given time in the future.

Return value

Integer reply, specifically:

- `1` if the timeout was set.
- `0` if `key` does not exist or the timeout could not be set (see: EXPIRE).

Examples

redis> SET mykey "Hello"

```
OK
```

redis> EXISTS mykey

```
(integer) 1
```

redis> EXPIREAT mykey 1293840000

```
(integer) 1
```

redis> EXISTS mykey

```
(integer) 0
```

redis>

FLUSHALL

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- **FLUSHALL**
- FLUSHDB
- INFO
- LASTSAVE
- MONITOR
- SAVE
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- SYNC
- TIME

Available since 1.0.0.

Delete all the keys of all the existing databases, not just the currently selected one. This command never fails.

Return value

Status code reply

FLUSHDB

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- **FLUSHDB**
- INFO
- LASTSAVE
- MONITOR
- SAVE
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- SYNC
- TIME

Available since 1.0.0.

Delete all the keys of the currently selected DB. This command never fails.

Return value

Status code reply

GET key

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- DECRBY
- **GET**
- GETBIT
- GETRANGE
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- PSETEX
- SET
- SETBIT
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 1.0.0.

Time complexity: 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.

Return value

Bulk reply: the value of `key`, or `nil` when `key` does not exist.

Examples

redis> GET nonexisting

```
(nil)
```

redis> SET mykey "Hello"

```
OK
```

redis> GET mykey

```
"Hello"
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

GETBIT key offset

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- DECRBY
- GET
- **GETBIT**
- GETRANGE
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- PSETEX
- SET
- SETBIT
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 2.2.0.

Time complexity: O(1)

Returns the bit value at *offset* in the string value stored at *key*.

When *offset* is beyond the string length, the string is assumed to be a contiguous space with 0 bits. When *key* does not exist it is assumed to be an empty string, so *offset* is always out of range and the value is also assumed to be a contiguous space with 0 bits.

Return value

Integer reply: the bit value stored at *offset*.

Examples

redis> SETBIT mykey 7 1

```
(integer) 0
```

redis> GETBIT mykey 0

```
(integer) 0
```

redis> GETBIT mykey 7

```
(integer) 1
```

redis> GETBIT mykey 100

```
(integer) 0
```

redis>

GETRANGE key start end

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- DECRBY
- GET
- GETBIT
- **GETRANGE**
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- PSETEX
- SET
- SETBIT
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 2.4.0.

Time complexity: O(N) where N is the length of the returned string. The complexity is ultimately determined by the returned length, but because creating a substring from an existing string is very cheap, it can be considered O(1) for small strings.

Warning: this command was renamed to GETRANGE, it is called `SUBSTR` in Redis versions `<= 2.0`.

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.

Return value

Bulk reply

Examples

redis> SET mykey "This is a string"

```
OK
```

redis> GETRANGE mykey 0 3

```
"This"
```

redis> GETRANGE mykey -3 -1

```
"ing"
```

redis> GETRANGE mykey 0 -1

```
"This is a string"
```

redis> GETRANGE mykey 10 100

```
"string"
```

redis>

GETSET key value

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- DECRBY
- GET
- GETBIT
- GETRANGE
- **GETSET**
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- PSETEX
- SET
- SETBIT
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 1.0.0.

Time complexity: O(1)

Atomically sets `key` to `value` and returns the old value stored at `key`. Returns an error when `key` exists but does not hold a string value.

Design pattern

GETSET can be used together with INCR for counting with atomic reset. For example: a process may call INCR against the key `mycounter` every time some event occurs, but from time to time we need to get the value of the counter and reset it to zero atomically. This can be done using `GETSET mycounter "0"`:

redis> INCR mycounter

```
(integer) 1
```

redis> GETSET mycounter "0"

```
"1"
```

redis> GET mycounter

```
"0"
```

redis>

Return value

Bulk reply: the old value stored at `key`, or `nil` when `key` did not exist.

Examples

redis> SET mykey "Hello"

```
OK
```

redis> GETSET mykey "World"

```
"Hello"
```

redis> GET mykey

```
"World"
```

redis>

HDEL key field [field ...]

Related commands

- **HDEL**
- HEXISTS
- HGET
- HGETALL
- HINCRBY
- HINCRBYFLOAT
- HKEYS
- HLEN
- HMGET
- HMSET
- HSET
- HSETNX
- HVALS

Available since 2.0.0.

Time complexity: O(N) where N is the number of fields to be removed.

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`.

Return value

Integer reply: the number of fields that were removed from the hash, not including specified but non existing fields.

History

- `>= 2.4`: Accepts multiple `field` arguments. Redis versions older than 2.4 can only remove a field per call.

  To remove multiple fields from a hash in an atomic fashion in earlier versions, use a MULTI / EXEC block.

Examples

redis> HSET myhash field1 "foo"

```
(integer) 1
```

redis> HDEL myhash field1

```
(integer) 1
```

redis> HDEL myhash field2

```
(integer) 0
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

HEXISTS key field

Related commands

- HDEL
- **HEXISTS**
- HGET
- HGETALL
- HINCRBY
- HINCRBYFLOAT
- HKEYS
- HLEN
- HMGET
- HMSET
- HSET
- HSETNX
- HVALS

Available since 2.0.0.

Time complexity: O(1)

Returns if `field` is an existing field in the hash stored at `key`.

Return value

Integer reply, specifically:

- `1` if the hash contains `field`.
- `0` if the hash does not contain `field`, or `key` does not exist.

Examples

redis> HSET myhash field1 "foo"

```
(integer) 1
```

redis> HEXISTS myhash field1

```
(integer) 1
```

redis> HEXISTS myhash field2

```
(integer) 0
```

redis>

This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

HGET key field

Related commands

- HDEL
- HEXISTS
- **HGET**
- HGETALL
- HINCRBY
- HINCRBYFLOAT
- HKEYS
- HLEN
- HMGET
- HMSET
- HSET
- HSETNX
- HVALS

Available since 2.0.0.

Time complexity: O(1)

Returns the value associated with `field` in the hash stored at `key`.

Return value

Bulk reply: the value associated with `field`, or `nil` when `field` is not present in the hash or `key` does not exist.

Examples

redis> HSET myhash field1 "foo"

```
(integer) 1
```

redis> HGET myhash field1

```
"foo"
```

redis> HGET myhash field2

```
(nil)
```

redis>

HGETALL key

Related commands

- HDEL
- HEXISTS
- HGET
- **HGETALL**
- HINCRBY
- HINCRBYFLOAT
- HKEYS
- HLEN
- HMGET
- HMSET
- HSET
- HSETNX
- HVALS

Available since 2.0.0.

Time complexity: O(N) where N is the size of the hash.

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.

Return value

Multi-bulk reply: list of fields and their values stored in the hash, or an empty list when `key` does not exist.

Examples

redis> HSET myhash field1 "Hello"

```
(integer) 1
```

redis> HSET myhash field2 "World"

```
(integer) 1
```

redis> HGETALL myhash

```
1) "field1"
```

redis>

HINCRBY key field increment

Related commands

- HDEL
- HEXISTS
- HGET
- HGETALL
- **HINCRBY**
- HINCRBYFLOAT
- HKEYS
- HLEN
- HMGET
- HMSET
- HSET
- HSETNX
- HVALS

Available since 2.0.0.

Time complexity: 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.

The range of values supported by HINCRBY is limited to 64 bit signed integers.

Return value

Integer reply: the value at `field` after the increment operation.

Examples

Since the `increment` argument is signed, both increment and decrement operations can be performed:

redis> HSET myhash field 5

```
(integer) 1
```

redis> HINCRBY myhash field 1

```
(integer) 6
```

redis> HINCRBY myhash field -1

```
(integer) 5
```

redis> HINCRBY myhash field -10

```
(integer) -5
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

HINCRBYFLOAT key field increment

Related commands

- HDEL
- HEXISTS
- HGET
- HGETALL
- HINCRBY
- **HINCRBYFLOAT**
- HKEYS
- HLEN
- HMGET
- HMSET
- HSET
- HSETNX
- HVALS

Available since 2.6.0.

Time complexity: O(1)

Increment the specified `field` of an hash stored at `key`, and representing a floating point number, by the specified `increment`. If the field does not exist, it is set to `0` before performing the operation. An error is returned if one of the following conditions occur:

- The field contains a value of the wrong type (not a string).
- The current field content or the specified increment are not parsable as a double precision floating point number.

The exact behavior of this command is identical to the one of the INCRBYFLOAT command, please refer to the documentation of INCRBYFLOAT for further information.

Return value

Bulk reply: the value of `field` after the increment.

Examples

redis> HSET mykey field 10.50

```
(integer) 1
```

redis> HINCRBYFLOAT mykey field 0.1

```
"10.6"
```

redis> HSET mykey field 5.0e3

```
(integer) 0
```

redis> HINCRBYFLOAT mykey field 2.0e2

```
"5200"
```

redis>

Implementation details

The command is always propagated in the replication link and the Append Only File as a HSET operation, so that differences in the underlying floating point math implementation will not be sources of inconsistency.

HKEYS key

Related commands

- HDEL
- HEXISTS
- HGET
- HGETALL
- HINCRBY
- HINCRBYFLOAT
- **HKEYS**
- HLEN
- HMGET
- HMSET
- HSET
- HSETNX
- HVALS

Available since 2.0.0.

Time complexity: O(N) where N is the size of the hash.

Returns all field names in the hash stored at `key`.

Return value

Multi-bulk reply: list of fields in the hash, or an empty list when `key` does not exist.

Examples

redis> HSET myhash field1 "Hello"

```
(integer) 1
```

redis> HSET myhash field2 "World"

```
(integer) 1
```

redis> HKEYS myhash

```
1) "field1"
```

redis>

HLEN key

Related commands

- HDEL
- HEXISTS
- HGET
- HGETALL
- HINCRBY
- HINCRBYFLOAT
- HKEYS
- **HLEN**
- HMGET
- HMSET
- HSET
- HSETNX
- HVALS

Available since 2.0.0.

Time complexity: O(1)

Returns the number of fields contained in the hash stored at `key`.

Return value

Integer reply: number of fields in the hash, or `0` when `key` does not exist.

Examples

redis> HSET myhash field1 "Hello"

```
(integer) 1
```

redis> HSET myhash field2 "World"

```
(integer) 1
```

redis> HLEN myhash

```
(integer) 2
```

redis>

HMGET key field [field ...]

Related commands

- HDEL
- HEXISTS
- HGET
- HGETALL
- HINCRBY
- HINCRBYFLOAT
- HKEYS
- HLEN
- **HMGET**
- HMSET
- HSET
- HSETNX
- HVALS

Available since 2.0.0.

Time complexity: O(N) where N is the number of fields being requested.

Returns the values associated with the specified `fields` in the hash stored at `key`.

For every `field` that does not exist in the hash, a `nil` value is returned. Because a non-existing keys are treated as empty hashes, running HMGET against a non-existing `key` will return a list of `nil` values.

Return value

Multi-bulk reply: list of values associated with the given fields, in the same order as they are requested.

redis> HSET myhash field1 "Hello"

```
(integer) 1
```

redis> HSET myhash field2 "World"

```
(integer) 1
```

redis> HMGET myhash field1 field2 nofield

```
1) "Hello"
```

redis>

HMSET key field value [field value ...]

Related commands

- HDEL
- HEXISTS
- HGET
- HGETALL
- HINCRBY
- HINCRBYFLOAT
- HKEYS
- HLEN
- HMGET
- **HMSET**
- HSET
- HSETNX
- HVALS

Available since 2.0.0.

Time complexity: O(N) where N is the number of fields being set.

Sets the specified fields to their respective values in the hash stored at `key`. This command overwrites any existing fields in the hash. If `key` does not exist, a new key holding a hash is created.

Return value

Status code reply

Examples

redis> HMSET myhash field1 "Hello" field2 "World"

```
OK
```

redis> HGET myhash field1

```
"Hello"
```

redis> HGET myhash field2

```
"World"
```

redis>

HSET key field value

Related commands

- HDEL
- HEXISTS
- HGET
- HGETALL
- HINCRBY
- HINCRBYFLOAT
- HKEYS
- HLEN
- HMGET
- HMSET
- **HSET**
- HSETNX
- HVALS

Available since 2.0.0.

Time complexity: O(1)

Sets `field` in the hash stored at `key` to `value`. If `key` does not exist, a new key holding a hash is created. If `field` already exists in the hash, it is overwritten.

Return value

Integer reply, specifically:

- `1` if `field` is a new field in the hash and `value` was set.
- `0` if `field` already exists in the hash and the value was updated.

Examples

redis> HSET myhash field1 "Hello"

```
(integer) 1
```

redis> HGET myhash field1

```
"Hello"
```

redis>

HSETNX key field value

Related commands

- HDEL
- HEXISTS
- HGET
- HGETALL
- HINCRBY
- HINCRBYFLOAT
- HKEYS
- HLEN
- HMGET
- HMSET
- HSET
- **HSETNX**
- HVALS

Available since 2.0.0.

Time complexity: 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.

Return value

Integer reply, specifically:

- `1` if `field` is a new field in the hash and `value` was set.
- `0` if `field` already exists in the hash and no operation was performed.

Examples

redis> HSETNX myhash field "Hello"

```
(integer) 1
```

redis> HSETNX myhash field "World"

```
(integer) 0
```

redis> HGET myhash field

```
"Hello"
```

redis>

Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

HVALS key

Related commands

- HDEL
- HEXISTS
- HGET
- HGETALL
- HINCRBY
- HINCRBYFLOAT
- HKEYS
- HLEN
- HMGET
- HMSET
- HSET
- HSETNX
- **HVALS**

Available since 2.0.0.

Time complexity: O(N) where N is the size of the hash.

Returns all values in the hash stored at `key`.

Return value

Multi-bulk reply: list of values in the hash, or an empty list when `key` does not exist.

Examples

redis> HSET myhash field1 "Hello"

```
(integer) 1
```

redis> HSET myhash field2 "World"

```
(integer) 1
```

redis> HVALS myhash

```
1) "Hello"
```

redis>

INCR key

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- DECRBY
- GET
- GETBIT
- GETRANGE
- GETSET
- **INCR**
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- PSETEX
- SET
- SETBIT
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 1.0.0.

Time complexity: 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. This operation is limited to 64 bit signed integers.

Note: this is a string operation because Redis does not have a dedicated integer type. The string stored at the key is interpreted as a base-10 **64 bit signed integer** to execute the operation.

Redis stores integers in their integer representation, so for string values that actually hold an integer, there is no overhead for storing the string representation of the integer.

Return value

Integer reply: the value of `key` after the increment

Examples

redis> SET mykey "10"

```
OK
```

redis> INCR mykey

```
(integer) 11
```

redis> GET mykey

```
"11"
```

redis>

Pattern: Counter

The counter pattern is the most obvious thing you can do with Redis atomic increment operations. The idea is simply send an INCR command to Redis every time an operation occurs. For instance in a web application we may want to know how many page views this user did every day of the year.

To do so the web application may simply increment a key every time the user performs a page view, creating the key name concatenating the User ID and a string representing the current date.

This simple pattern can be extended in many ways:

- It is possible to use INCR and EXPIRE together at every page view to have a counter counting only the latest N page views separated by less than the specified amount of seconds.
- A client may use GETSET in order to atomically get the current counter value and reset it to zero.
- Using other atomic increment/decrement commands like DECR or INCRBY it is possible to handle values that may get bigger or smaller depending on the operations performed by the user. Imagine for instance the score of different users in an online game.

Pattern: Rate limiter

The rate limiter pattern is a special counter that is used to limit the rate at which an operation can be performed. The classical materialization of this pattern involves limiting the number of requests that can be performed against a public API.

We provide two implementations of this pattern using INCR, where we assume that the problem to solve is limiting the number of API calls to a maximum of *ten requests per second per IP address*.

Pattern: Rate limiter 1

The more simple and direct implementation of this pattern is the following:

```
FUNCTION LIMIT_API_CALL(ip)
```

Basically we have a counter for every IP, for every different second. But this counters are always incremented setting an expire of 10 seconds so that they'll be removed by Redis automatically when the current second is a

different one.

Note the used of MULTI and EXEC in order to make sure that we'll both increment and set the expire at every API call.

Pattern: Rate limiter 2

An alternative implementation uses a single counter, but is a bit more complex to get it right without race conditions. We'll examine different variants.

```
FUNCTION LIMIT_API_CALL(ip):
```

The counter is created in a way that it only will survive one second, starting from the first request performed in the current second. If there are more than 10 requests in the same second the counter will reach a value greater than 10, otherwise it will expire and start again from 0.

In the above code there is a race condition. If for some reason the client performs the INCR command but does not perform the EXPIRE the key will be leaked until we'll see the same IP address again.

This can be fixed easily turning the INCR with optional EXPIRE into a Lua script that is send using the EVAL command (only available since Redis version 2.6).

```
local current
```

There is a different way to fix this issue without using scripting, but using Redis lists instead of counters. The implementation is more complex and uses more advanced features but has the advantage of remembering the IP addresses of the clients currently performing an API call, that may be useful or not depending on the application.

```
FUNCTION LIMIT_API_CALL(ip)
```

The RPUSHX command only pushes the element if the key already exists.

Note that we have a race here, but it is not a problem: EXISTS may return false but the key may be created by another client before we create it inside the MULTI / EXEC block. However this race will just miss an API call under rare conditions, so the rate limiting will still work correctly.

INCRBY key increment

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- DECRBY
- GET
- GETBIT
- GETRANGE
- GETSET
- INCR
- **INCRBY**
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- PSETEX
- SET
- SETBIT
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 1.0.0.

Time complexity: O(1)

Increments the number stored at `key` by `increment`. 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. This operation is limited to 64 bit signed integers.

See INCR for extra information on increment/decrement operations.

Return value

Integer reply: the value of `key` after the increment

Examples

redis> SET mykey "10"

```
OK
```

```
redis> INCRBY mykey 5

(integer) 15

redis>
```

Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by 

INCRBYFLOAT key increment

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- DECRBY
- GET
- GETBIT
- GETRANGE
- GETSET
- INCR
- INCRBY
- **INCRBYFLOAT**
- MGET
- MSET
- MSETNX
- PSETEX
- SET
- SETBIT
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 2.6.0.

Time complexity: O(1)

Increment the string representing a floating point number stored at `key` by the specified `increment`. If the key does not exist, it is set to `0` before performing the operation. An error is returned if one of the following conditions occur:

- The key contains a value of the wrong type (not a string).
- The current key content or the specified increment are not parsable as a double precision floating point number.

If the command is successful the new incremented value is stored as the new value of the key (replacing the old one), and returned to the caller as a string.

Both the value already contained in the string key and the increment argument can be optionally provided in exponential notation, however the value computed after the increment is stored consistently in the same format, that is, an integer number followed (if needed) by a dot, and a variable number of digits representing the decimal part of the number. Trailing zeroes are always removed.

The precision of the output is fixed at 17 digits after the decimal point regardless of the actual internal precision of the computation.

Return value

Bulk reply: the value of `key` after the increment.

Examples

redis> SET mykey 10.50

```
OK
```

redis> INCRBYFLOAT mykey 0.1

```
"10.6"
```

redis> SET mykey 5.0e3

```
OK
```

redis> INCRBYFLOAT mykey 2.0e2

```
"5200"
```

redis>

Implementation details

The command is always propagated in the replication link and the Append Only File as a SET operation, so that differences in the underlying floating point math implementation will not be sources of inconsistency.


Commands Clients Documentation Community Download Issues
All Keys Strings Hashes Lists Sets Sorted Sets Pub/Sub Transactions Scripting Connection Server

- APPEND key value Append a value to a key
- AUTH password Authenticate to the server
- BGREWRITEAOF Asynchronously rewrite the append-only file
- BGSAVE Asynchronously save the dataset to disk
- BITCOUNT key [start] [end] Count set bits in a string
- BITOP operation destkey key [key ...] Perform bitwise operations between strings
- BLPOP key [key ...] timeout Remove and get the first element in a list, or block until one is available
- BRPOP key [key ...] timeout Remove and get the last element in a list, or block until one is available
- BRPOPLPUSH source destination timeout Pop a value from a list, push it to another list and return it; or block until one is available
- CLIENT KILL ip:port Kill the connection of a client
- CLIENT LIST Get the list of client connections
- CONFIG GET parameter Get the value of a configuration parameter
- CONFIG SET parameter value Set a configuration parameter to the given value
- CONFIG RESETSTAT Reset the stats returned by INFO
- DBSIZE Return the number of keys in the selected database
- DEBUG OBJECT key Get debugging information about a key
- DEBUG SEGFAULT Make the server crash
- DECR key Decrement the integer value of a key by one
- DECRBY key decrement Decrement the integer value of a key by the given number
- DEL key [key ...] Delete a key
- DISCARD Discard all commands issued after MULTI
- DUMP key Return a serialized version of the value stored at the specified key.
- ECHO message Echo the given string
- EVAL script numkeys key [key ...] arg [arg ...] Execute a Lua script server side
- EVALSHA sha1 numkeys key [key ...] arg [arg ...] Execute a Lua script server side
- EXEC Execute all commands issued after MULTI
- EXISTS key Determine if a key exists
- EXPIRE key seconds Set a key's time to live in seconds
- EXPIREAT key timestamp Set the expiration for a key as a UNIX timestamp
- FLUSHALL Remove all keys from all databases
- FLUSHDB Remove all keys from the current database
- GET key Get the value of a key
- GETBIT key offset Returns the bit value at offset in the string value stored at key
- GETRANGE key start end Get a substring of the string stored at a key
- GETSET key value Set the string value of a key and return its old value
- HDEL key field [field ...] Delete one or more hash fields
- HEXISTS key field Determine if a hash field exists
- HGET key field Get the value of a hash field
- HGETALL key Get all the fields and values in a hash
- HINCRBY key field increment Increment the integer value of a hash field by the given number
- HINCRBYFLOAT key field increment Increment the float value of a hash field by the given amount
- HKEYS key Get all the fields in a hash
- HLEN key Get the number of fields in a hash
- HMGET key field [field ...] Get the values of all the given hash fields
- HMSET key field value [field value ...] Set multiple hash fields to multiple values
- HSET key field value Set the string value of a hash field

- HSETNX key field value Set the value of a hash field, only if the field does not exist
- HVALS key Get all the values in a hash
- INCR key Increment the integer value of a key by one
- INCRBY key increment Increment the integer value of a key by the given amount
- INCRBYFLOAT key increment Increment the float value of a key by the given amount
- INFO Get information and statistics about the server
- KEYS pattern Find all keys matching the given pattern
- LASTSAVE Get the UNIX time stamp of the last successful save to disk
- LINDEX key index Get an element from a list by its index
- LINSERT key BEFORE|AFTER pivot value Insert an element before or after another element in a list
- LLEN key Get the length of a list
- LPOP key Remove and get the first element in a list
- LPUSH key value [value ...] Prepend one or multiple values to a list
- LPUSHX key value Prepend a value to a list, only if the list exists
- LRANGE key start stop Get a range of elements from a list
- LREM key count value Remove elements from a list
- LSET key index value Set the value of an element in a list by its index
- LTRIM key start stop Trim a list to the specified range
- MGET key [key ...] Get the values of all the given keys
- MIGRATE host port key destination-db timeout Atomically transfer a key from a Redis instance to another one.
- MONITOR Listen for all requests received by the server in real time
- MOVE key db Move a key to another database
- MSET key value [key value ...] Set multiple keys to multiple values
- MSETNX key value [key value ...] Set multiple keys to multiple values, only if none of the keys exist
- MULTI Mark the start of a transaction block
- OBJECT subcommand [arguments [arguments ...]] Inspect the internals of Redis objects
- PERSIST key Remove the expiration from a key
- PEXPIRE key milliseconds Set a key's time to live in milliseconds
- PEXPIREAT key milliseconds-timestamp Set the expiration for a key as a UNIX timestamp specified in milliseconds
- PING Ping the server
- PSETEX key milliseconds value Set the value and expiration in milliseconds of a key
- PSUBSCRIBE pattern [pattern ...] Listen for messages published to channels matching the given patterns
- PTTL key Get the time to live for a key in milliseconds
- PUBLISH channel message Post a message to a channel
- PUNSUBSCRIBE [pattern [pattern ...]] Stop listening for messages posted to channels matching the given patterns
- QUIT Close the connection
- RANDOMKEY Return a random key from the keyspace
- RENAME key newkey Rename a key
- RENAMENX key newkey Rename a key, only if the new key does not exist
- RESTORE key ttl serialized-value Create a key using the provided serialized value, previously obtained using DUMP.
- RPOP key Remove and get the last element in a list
- RPOPLPUSH source destination Remove the last element in a list, append it to another list and return it
- RPUSH key value [value ...] Append one or multiple values to a list
- RPUSHX key value Append a value to a list, only if the list exists
- SADD key member [member ...] Add one or more members to a set

- SAVE Synchronously save the dataset to disk
- SCARD key Get the number of members in a set
- SCRIPT EXISTS script [script ...] Check existence of scripts in the script cache.
- SCRIPT FLUSH Remove all the scripts from the script cache.
- SCRIPT KILL Kill the script currently in execution.
- SCRIPT LOAD script Load the specified Lua script into the script cache.
- SDIFF key [key ...] Subtract multiple sets
- SDIFFSTORE destination key [key ...] Subtract multiple sets and store the resulting set in a key
- SELECT index Change the selected database for the current connection
- SET key value Set the string value of a key
- SETBIT key offset value Sets or clears the bit at offset in the string value stored at key
- SETEX key seconds value Set the value and expiration of a key
- SETNX key value Set the value of a key, only if the key does not exist
- SETRANGE key offset value Overwrite part of a string at key starting at the specified offset
- SHUTDOWN [NOSAVE] [SAVE] Synchronously save the dataset to disk and then shut down the server
- SINTER key [key ...] Intersect multiple sets
- SINTERSTORE destination key [key ...] Intersect multiple sets and store the resulting set in a key
- SISMEMBER key member Determine if a given value is a member of a set
- SLAVEOF host port Make the server a slave of another instance, or promote it as master
- SLOWLOG subcommand [argument] Manages the Redis slow queries log
- SMEMBERS key Get all the members in a set
- SMOVE source destination member Move a member from one set to another
- SORT key [BY pattern] [LIMIT offset count] [GET pattern [GET pattern ...]] [ASC|DESC] [ALPHA] [STORE destination] Sort the elements in a list, set or sorted set
- SPOP key Remove and return a random member from a set
- SRANDMEMBER key [count] Get one or multiple random members from a set
- SREM key member [member ...] Remove one or more members from a set
- STRLEN key Get the length of the value stored in a key
- SUBSCRIBE channel [channel ...] Listen for messages published to the given channels
- SUNION key [key ...] Add multiple sets
- SUNIONSTORE destination key [key ...] Add multiple sets and store the resulting set in a key
- SYNC Internal command used for replication
- TIME Return the current server time
- TTL key Get the time to live for a key
- TYPE key Determine the type stored at key
- UNSUBSCRIBE [channel [channel ...]] Stop listening for messages posted to the given channels
- UNWATCH Forget about all watched keys
- WATCH key [key ...] Watch the given keys to determine execution of the MULTI/EXEC block
- ZADD key score member [score] [member] Add one or more members to a sorted set, or update its score if it already exists
- ZCARD key Get the number of members in a sorted set
- ZCOUNT key min max Count the members in a sorted set with scores within the given values
- ZINCRBY key increment member Increment the score of a member in a sorted set
- ZINTERSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX] Intersect multiple sorted sets and store the resulting sorted set in a new key
- ZRANGE key start stop [WITHSCORES] Return a range of members in a sorted set, by index
- ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count] Return a range of members in a sorted set, by score
- ZRANK key member Determine the index of a member in a sorted set
- ZREM key member [member ...] Remove one or more members from a sorted set

- ZREMRANGEBYRANK key start stop Remove all members in a sorted set within the given indexes
- ZREMRANGEBYSCORE key min max Remove all members in a sorted set within the given scores
- ZREVRANGE key start stop [WITHSCORES] Return a range of members in a sorted set, by index, with scores ordered from high to low
- ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset count] Return a range of members in a sorted set, by score, with scores ordered from high to low
- ZREVRANK key member Determine the index of a member in a sorted set, with scores ordered from high to low
- ZSCORE key member Get the score associated with the given member in a sorted set
- ZUNIONSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX] Add multiple sorted sets and store the resulting sorted set in a new key

This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

INFO

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- **INFO**
- LASTSAVE
- MONITOR
- SAVE
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- SYNC
- TIME

Available since 1.0.0.

The INFO command returns information and statistics about the server in a format that is simple to parse by computers and easy to read by humans.

The optional parameter can be used to select a specific section of information:

- `server`: General information about the Redis server
- `clients`: Client connections section
- `memory`: Memory consumption related information
- `persistence`: RDB and AOF related information
- `stats`: General statistics
- `replication`: Master/slave replication information
- `cpu`: CPU consumption statistics
- `commandstats`: Redis command statistics
- `cluster`: Redis Cluster section
- `keyspace`: Database related statistics

It can also take the following values:

- `all`: Return all sections
- `default`: Return only the default set of sections

When no parameter is provided, the `default` option is assumed.

Return value

Bulk reply: as a collection of text lines.

Lines can contain a section name (starting with a # character) or a property. All the properties are in the form of `field:value` terminated by `\r\n`.

redis> INFO

```
# Server
```

redis>

Notes

Please note depending on the version of Redis some of the fields have been added or removed. A robust client application should therefore parse the result of this command by skipping unknown properties, and gracefully handle missing fields.

Here is the description of fields for Redis >= 2.4.

Here is the meaning of all fields in the **server** section:

- `redis_version`: Version of the Redis server
- `redis_git_sha1`: Git SHA1
- `redis_git_dirty`: Git dirty flag
- `os`: Operating system hosting the Redis server
- `arch_bits`: Architecture (32 or 64 bits)
- `multiplexing_api`: event loop mechanism used by Redis
- `gcc_version`: Version of the GCC compiler used to compile the Redis server
- `process_id`: PID of the server process
- `run_id`: Random value identifying the Redis server (to be used by Sentinel and Cluster)
- `tcp_port`: TCP/IP listen port
- `uptime_in_seconds`: Number of seconds since Redis server start
- `uptime_in_days`: Same value expressed in days
- `lru_clock`: Clock incrementing every minute, for LRU management

Here is the meaning of all fields in the **clients** section:

- `connected_clients`: Number of client connections (excluding connections from slaves)
- `client_longest_output_list`: longest output list among current client connections
- `client_biggest_input_buf`: biggest input buffer among current client connections
- `blocked_clients`: Number of clients pending on a blocking call (BLPOP, BRPOP, BRPOPLPUSH)

Here is the meaning of all fields in the **memory** section:

- `used_memory`: total number of bytes allocated by Redis using its allocator (either standard **libc**, **jemalloc**, or an alternative allocator such as **tcmalloc**

- `used_memory_human`: Human readable representation of previous value
- `used_memory_rss`: Number of bytes that Redis allocated as seen by the operating system (a.k.a resident set size). This is the number reported by tools such as **top** and **ps**.
- `used_memory_peak`: Peak memory consumed by Redis (in bytes)
- `used_memory_peak_human`: Human readable representation of previous value
- `used_memory_lua`: Number of bytes used by the Lua engine
- `mem_fragmentation_ratio`: Ratio between `used_memory_rss` and `used_memory`
- `mem_allocator`: Memory allocator, chosen at compile time.

Ideally, the `used_memory_rss` value should be only slightly higher than `used_memory`. When rss >> used, a large difference means there is memory fragmentation (internal or external), which can be evaluated by checking `mem_fragmentation_ratio`. When used >> rss, it means part of Redis memory has been swapped off by the operating system: expect some significant latencies.

Because Redis does not have control over how its allocations are mapped to memory pages, high `used_memory_rss` is often the result of a spike in memory usage.

When Redis frees memory, the memory is given back to the allocator, and the allocator may or may not give the memory back to the system. There may be a discrepancy between the `used_memory` value and memory consumption as reported by the operating system. It may be due to the fact memory has been used and released by Redis, but not given back to the system. The `used_memory_peak` value is generally useful to check this point.

Here is the meaning of all fields in the **persistence** section:

- `loading`: Flag indicating if the load of a dump file is on-going
- `rdb_changes_since_last_save`: Number of changes since the last dump
- `rdb_bgsave_in_progress`: Flag indicating a RDB save is on-going
- `rdb_last_save_time`: Epoch-based timestamp of last successful RDB save
- `rdb_last_bgsave_status`: Status of the last RDB save operation
- `rdb_last_bgsave_time_sec`: Duration of the last RDB save operation in seconds
- `rdb_current_bgsave_time_sec`: Duration of the on-going RDB save operation if any
- `aof_enabled`: Flag indicating AOF logging is activated
- `aof_rewrite_in_progress`: Flag indicating a AOF rewrite operation is on-going
- `aof_rewrite_scheduled`: Flag indicating an AOF rewrite operation will be scheduled once the on-going RDB save is complete.
- `aof_last_rewrite_time_sec`: Duration of the last AOF rewrite operation in seconds
- `aof_current_rewrite_time_sec`: Duration of the on-going AOF rewrite operation if any
- `aof_last_bgrewrite_status`: Status of the last AOF rewrite operation

`changes_since_last_save` refers to the number of operations that produced some kind of changes in the dataset since the last time either SAVE or BGSAVE was called.

If AOF is activated, these additional fields will be added:

- `aof_current_size`: AOF current file size
- `aof_base_size`: AOF file size on latest startup or rewrite
- `aof_pending_rewrite`: Flag indicating an AOF rewrite operation will be scheduled once the on-going RDB save is complete.
- `aof_buffer_length`: Size of the AOF buffer
- `aof_rewrite_buffer_length`: Size of the AOF rewrite buffer

- `aof_pending_bio_fsync`: Number of fsync pending jobs in background I/O queue
- `aof_delayed_fsync`: Delayed fsync counter

If a load operation is on-going, these additional fields will be added:

- `loading_start_time`: Epoch-based timestamp of the start of the load operation
- `loading_total_bytes`: Total file size
- `loading_loaded_bytes`: Number of bytes already loaded
- `loading_loaded_perc`: Same value expressed as a percentage
- `loading_eta_seconds`: ETA in seconds for the load to be complete

Here is the meaning of all fields in the **stats** section:

- `total_connections_received`: Total number of connections accepted by the server
- `total_commands_processed`: Total number of commands processed by the server
- `instantaneous_ops_per_sec`: Number of commands processed per second
- `rejected_connections`: Number of connections rejected because of maxclients limit
- `expired_keys`: Total number of key expiration events
- `evicted_keys`: Number of evicted keys due to maxmemory limit
- `keyspace_hits`: Number of successful lookup of keys in the main dictionary
- `keyspace_misses`: Number of failed lookup of keys in the main dictionary
- `pubsub_channels`: Global number of pub/sub channels with client subscriptions
- `pubsub_patterns`: Global number of pub/sub pattern with client subscriptions
- `latest_fork_usec`: Duration of the latest fork operation in microseconds

Here is the meaning of all fields in the **replication** section:

- `role`: Value is "master" if the instance is slave of no one, or "slave" if the instance is enslaved to a master. Note that a slave can be master of another slave (daisy chaining).

If the instance is a slave, these additional fields are provided:

- `master_host`: Host or IP address of the master
- `master_port`: Master listening TCP port
- `master_link_status`: Status of the link (up/down)
- `master_last_io_seconds_ago`: Number of seconds since the last interaction with master
- `master_sync_in_progress`: Indicate the master is SYNCing to the slave

If a SYNC operation is on-going, these additional fields are provided:

- `master_sync_left_bytes`: Number of bytes left before SYNCing is complete
- `master_sync_last_io_seconds_ago`: Number of seconds since last transfer I/O during a SYNC operation

If the link between master and slave is down, an additional field is provided:

- `master_link_down_since_seconds`: Number of seconds since the link is down

The following field is always provided:

- `connected_slaves`: Number of connected slaves

For each slave, the following line is added:

- `slaveXXX`: id, ip address, port, state

Here is the meaning of all fields in the **cpu** section:

- `used_cpu_sys`: System CPU consumed by the Redis server
- `used_cpu_user`:User CPU consumed by the Redis server
- `used_cpu_sys_children`: System CPU consumed by the background processes
- `used_cpu_user_children`: User CPU consumed by the background processes

The **commandstats** section provides statistics based on the command type, including the number of calls, the total CPU time consumed by these commands, and the average CPU consumed per command execution.

For each command type, the following line is added:

- `cmdstat_XXX`:calls=XXX,usec=XXX,usec*per*call=XXX

The **cluster** section currently only contains a unique field:

- `cluster_enabled`: Indicate Redis cluster is enabled

The **keyspace** section provides statistics on the main dictionary of each database. The statistics are the number of keys, and the number of keys with an expiration.

For each database, the following line is added:

- `dbXXX`:keys=XXX,expires=XXX

KEYS pattern

Related commands

- DEL
- DUMP
- EXISTS
- EXPIRE
- EXPIREAT
- **KEYS**
- MIGRATE
- MOVE
- OBJECT
- PERSIST
- PEXPIRE
- PEXPIREAT
- PTTL
- RANDOMKEY
- RENAME
- RENAMENX
- RESTORE
- SORT
- TTL
- TYPE

Available since 1.0.0.

Time complexity: O(N) with N being the number of keys in the database, under the assumption that the key names in the database and the given pattern have limited length.

Returns all keys matching `pattern`.

While the time complexity for this operation is O(N), the constant times are fairly low. For example, Redis running on an entry level laptop can scan a 1 million key database in 40 milliseconds.

Warning: consider KEYS as a command that should only be used in production environments with extreme care. It may ruin performance when it is executed against large databases. This command is intended for debugging and special operations, such as changing your keyspace layout. Don't use KEYS in your regular application code. If you're looking for a way to find keys in a subset of your keyspace, consider using sets.

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`

Use \ to escape special characters if you want to match them verbatim.

Return value

Multi-bulk reply: list of keys matching `pattern`.

Examples

redis> MSET one 1 two 2 three 3 four 4

```
OK
```

redis> KEYS *o*

```
1) "one"
```

redis> KEYS t??

```
1) "two"
```

redis> KEYS *

```
1) "one"
```

redis>

LASTSAVE

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- INFO
- **LASTSAVE**
- MONITOR
- SAVE
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- SYNC
- TIME

Available since 1.0.0.

Return the UNIX TIME of the last DB save executed with success. A client may check if a BGSAVE command succeeded reading the LASTSAVE value, then issuing a BGSAVE command and checking at regular intervals every N seconds if LASTSAVE changed.

Return value

Integer reply: an UNIX time stamp.

LINDEX key index

Related commands

- BLPOP
- BRPOP
- BRPOPLPUSH
- **LINDEX**
- LINSERT
- LLEN
- LPOP
- LPUSH
- LPUSHX
- LRANGE
- LREM
- LSET
- LTRIM
- RPOP
- RPOPLPUSH
- RPUSH
- RPUSHX

Available since 1.0.0.

Time complexity: O(N) where N is the number of elements to traverse to get to the element at index. This makes asking for the first or the last element of the list O(1).

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.

When the value at `key` is not a list, an error is returned.

Return value

Bulk reply: the requested element, or `nil` when `index` is out of range.

Examples

redis> LPUSH mylist "World"

```
(integer) 1
```

redis> LPUSH mylist "Hello"

```
(integer) 2
```

redis> LINDEX mylist 0

```
"Hello"
```

redis> LINDEX mylist -1

```
"World"
```

redis> LINDEX mylist 3

```
(nil)
```

redis>

LINSERT key BEFORE|AFTER pivot value

Related commands

- BLPOP
- BRPOP
- BRPOPLPUSH
- LINDEX
- **LINSERT**
- LLEN
- LPOP
- LPUSH
- LPUSHX
- LRANGE
- LREM
- LSET
- LTRIM
- RPOP
- RPOPLPUSH
- RPUSH
- RPUSHX

Available since 2.2.0.

Time complexity: O(N) where N is the number of elements to traverse before seeing the value pivot. This means that inserting somewhere on the left end on the list (head) can be considered O(1) and inserting somewhere on the right end (tail) is O(N).

Inserts `value` 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.

Return value

Integer reply: the length of the list after the insert operation, or `-1` when the value `pivot` was not found.

Examples

redis> RPUSH mylist "Hello"

```
(integer) 1
```

redis> RPUSH mylist "World"

```
(integer) 2
```

redis> LINSERT mylist BEFORE "World" "There"

```
(integer) 3
```

redis> LRANGE mylist 0 -1

```
1) "Hello"
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

LLEN key

Related commands

- BLPOP
- BRPOP
- BRPOPLPUSH
- LINDEX
- LINSERT
- **LLEN**
- LPOP
- LPUSH
- LPUSHX
- LRANGE
- LREM
- LSET
- LTRIM
- RPOP
- RPOPLPUSH
- RPUSH
- RPUSHX

Available since 1.0.0.

Time complexity: 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.

Return value

Integer reply: the length of the list at `key`.

Examples

redis> LPUSH mylist "World"

```
(integer) 1
```

redis> LPUSH mylist "Hello"

```
(integer) 2
```

redis> LLEN mylist

```
(integer) 2
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by 

LPOP key

Related commands

- BLPOP
- BRPOP
- BRPOPLPUSH
- LINDEX
- LINSERT
- LLEN
- **LPOP**
- LPUSH
- LPUSHX
- LRANGE
- LREM
- LSET
- LTRIM
- RPOP
- RPOPLPUSH
- RPUSH
- RPUSHX

Available since 1.0.0.

Time complexity: O(1)

Removes and returns the first element of the list stored at `key`.

Return value

Bulk reply: the value of the first element, or `nil` when `key` does not exist.

Examples

redis> RPUSH mylist "one"

```
(integer) 1
```

redis> RPUSH mylist "two"

```
(integer) 2
```

redis> RPUSH mylist "three"

```
(integer) 3
```

redis> LPOP mylist

```
"one"
```

redis> LRANGE mylist 0 -1

```
1) "two"
```

redis>

LPUSH key value [value ...]

Related commands

- BLPOP
- BRPOP
- BRPOPLPUSH
- LINDEX
- LINSERT
- LLEN
- LPOP
- **LPUSH**
- LPUSHX
- LRANGE
- LREM
- LSET
- LTRIM
- RPOP
- RPOPLPUSH
- RPUSH
- RPUSHX

Available since 1.0.0.

Time complexity: 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.

It is possible to push multiple elements using a single command call just specifying multiple arguments at the end of the command. Elements are inserted one after the other to the head of the list, from the leftmost element to the rightmost element. So for instance the command `LPUSH mylist a b c` will result into a list containing `c` as first element, `b` as second element and `a` as third element.

Return value

Integer reply: the length of the list after the push operations.

History

- `>= 2.4`: Accepts multiple `value` arguments. In Redis versions older than 2.4 it was possible to push a single value per command.

Examples

redis> LPUSH mylist "world"

```
(integer) 1
```

redis> LPUSH mylist "hello"

```
(integer) 2
```

redis> LRANGE mylist 0 -1

```
1) "hello"
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

LPUSHX key value

Related commands

- BLPOP
- BRPOP
- BRPOPLPUSH
- LINDEX
- LINSERT
- LLEN
- LPOP
- LPUSH
- **LPUSHX**
- LRANGE
- LREM
- LSET
- LTRIM
- RPOP
- RPOPLPUSH
- RPUSH
- RPUSHX

Available since 2.2.0.

Time complexity: O(1)

Inserts `value` 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.

Return value

Integer reply: the length of the list after the push operation.

Examples

redis> LPUSH mylist "World"

```
(integer) 1
```

redis> LPUSHX mylist "Hello"

```
(integer) 2
```

redis> LPUSHX myotherlist "Hello"

```
(integer) 0
```

redis> LRANGE mylist 0 -1

```
1) "Hello"
```

redis> LRANGE myotherlist 0 -1

```
(empty list or set)
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

LRANGE key start stop

Related commands

- BLPOP
- BRPOP
- BRPOPLPUSH
- LINDEX
- LINSERT
- LLEN
- LPOP
- LPUSH
- LPUSHX
- **LRANGE**
- LREM
- LSET
- LTRIM
- RPOP
- RPOPLPUSH
- RPUSH
- RPUSHX

Available since 1.0.0.

Time complexity: O(S+N) where S is the start offset and N is the number of elements in the specified range.

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.

These offsets can also be negative numbers indicating offsets starting at the end of the list. For example, `-1` is the last element of the list, `-2` the penultimate, and so on.

Consistency with range functions in various programming languages

Note that if you have a list of numbers from 0 to 100, `LRANGE list 0 10` will return 11 elements, that is, the rightmost item is included. This **may or may not** be consistent with behavior of range-related functions in your programming language of choice (think Ruby's `Range.new`, `Array#slice` or Python's `range()` function).

Out-of-range indexes

Out of range indexes will not produce an error. If `start` is larger than the end of the list, an empty list is returned. If `stop` is larger than the actual end of the list, Redis will treat it like the last element of the list.

Return value

Multi-bulk reply: list of elements in the specified range.

Examples

redis> RPUSH mylist "one"

```
(integer) 1
```

redis> RPUSH mylist "two"

```
(integer) 2
```

redis> RPUSH mylist "three"

```
(integer) 3
```

redis> LRANGE mylist 0 0

```
1) "one"
```

redis> LRANGE mylist -3 2

```
1) "one"
```

redis> LRANGE mylist -100 100

```
1) "one"
```

redis> LRANGE mylist 5 10

```
(empty list or set)
```

redis>

LREM key count value

Related commands

- BLPOP
- BRPOP
- BRPOPLPUSH
- LINDEX
- LINSERT
- LLEN
- LPOP
- LPUSH
- LPUSHX
- LRANGE
- **LREM**
- LSET
- LTRIM
- RPOP
- RPOPLPUSH
- RPUSH
- RPUSHX

Available since 1.0.0.

Time complexity: O(N) where N is the length of the list.

Removes the first `count` occurrences of elements equal to `value` from the list stored at `key`. The `count` argument influences the operation in the following ways:

- `count > 0`: Remove elements equal to `value` moving from head to tail.
- `count < 0`: Remove elements equal to `value` moving from tail to head.
- `count = 0`: Remove all elements equal to `value`.

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

Note that non-existing keys are treated like empty lists, so when `key` does not exist, the command will always return `0`.

Return value

Integer reply: the number of removed elements.

Examples

redis> RPUSH mylist "hello"

```
(integer) 1
```

redis> RPUSH mylist "hello"

```
(integer) 2
```

redis> RPUSH mylist "foo"

```
(integer) 3
```

redis> RPUSH mylist "hello"

```
(integer) 4
```

redis> LREM mylist -2 "hello"

```
(integer) 2
```

redis> LRANGE mylist 0 -1

```
1) "hello"
```

redis>

LSET key index value

Related commands

- BLPOP
- BRPOP
- BRPOPLPUSH
- LINDEX
- LINSERT
- LLEN
- LPOP
- LPUSH
- LPUSHX
- LRANGE
- LREM
- **LSET**
- LTRIM
- RPOP
- RPOPLPUSH
- RPUSH
- RPUSHX

Available since 1.0.0.

Time complexity: O(N) where N is the length of the list. Setting either the first or the last element of the list is O(1).

Sets the list element at `index` to `value`. For more information on the `index` argument, see LINDEX.

An error is returned for out of range indexes.

Return value

Status code reply

Examples

redis> RPUSH mylist "one"

```
(integer) 1
```

redis> RPUSH mylist "two"

```
(integer) 2
```

redis> RPUSH mylist "three"

```
(integer) 3
```

```
redis> LSET mylist 0 "four"
OK
redis> LSET mylist -2 "five"
OK
redis> LRANGE mylist 0 -1
1) "four"
redis>
```

LTRIM key start stop

Related commands

- BLPOP
- BRPOP
- BRPOPLPUSH
- LINDEX
- LINSERT
- LLEN
- LPOP
- LPUSH
- LPUSHX
- LRANGE
- LREM
- LSET
- **LTRIM**
- RPOP
- RPOPLPUSH
- RPUSH
- RPUSHX

Available since 1.0.0.

Time complexity: O(N) where N is the number of elements to be removed by the operation.

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.

For example: `LTRIM foobar 0 2` will modify the list stored at `foobar` so that only the first three elements of the list will remain.

`start` and `end` can also be negative numbers indicating offsets from the end of the list, where `-1` is the last element of the list, `-2` the penultimate element and so on.

Out of range indexes will not produce an error: if `start` is larger than the end of the list, or `start > end`, the result will be an empty list (which causes `key` to be removed). If `end` is larger than the end of the list, Redis will treat it like the last element of the list.

A common use of LTRIM is together with LPUSH / RPUSH. For example:

```
LPUSH mylist someelement
```

This pair of commands will push a new element on the list, while making sure that the list will not grow larger than 100 elements. This is very useful when using Redis to store logs for example. It is important to note that when used in this way LTRIM is an O(1) operation because in the average case just one element is removed from the tail of the list.

Return value

Status code reply

Examples

redis> RPUSH mylist "one"

```
(integer) 1
```

redis> RPUSH mylist "two"

```
(integer) 2
```

redis> RPUSH mylist "three"

```
(integer) 3
```

redis> LTRIM mylist 1 -1

```
OK
```

redis> LRANGE mylist 0 -1

```
1) "two"
```

redis>

MGET key [key ...]

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- DECRBY
- GET
- GETBIT
- GETRANGE
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- **MGET**
- MSET
- MSETNX
- PSETEX
- SET
- SETBIT
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 1.0.0.

Time complexity: O(N) where N is the number of keys to retrieve.

Returns the values of all specified keys. For every key that does not hold a string value or does not exist, the special value `nil` is returned. Because of this, the operation never fails.

Return value

Multi-bulk reply: list of values at the specified keys.

Examples

redis> SET key1 "Hello"

```
OK
```

redis> SET key2 "World"

```
OK
```

redis> MGET key1 key2 nonexisting

```
1) "Hello"
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

MIGRATE host port key destination-db timeout

Related commands

- DEL
- DUMP
- EXISTS
- EXPIRE
- EXPIREAT
- KEYS
- **MIGRATE**
- MOVE
- OBJECT
- PERSIST
- PEXPIRE
- PEXPIREAT
- PTTL
- RANDOMKEY
- RENAME
- RENAMENX
- RESTORE
- SORT
- TTL
- TYPE

Available since 2.6.0.

Time complexity: This command actually executes a DUMP+DEL in the source instance, and a RESTORE in the target instance. See the pages of these commands for time complexity. Also an O(N) data transfer between the two instances is performed.

Atomically transfer a key from a source Redis instance to a destination Redis instance. On success the key is deleted from the original instance and is guaranteed to exist in the target instance.

The command is atomic and blocks the two instances for the time required to transfer the key, at any given time the key will appear to exist in a given instance or in the other instance, unless a timeout error occurs.

The command internally uses DUMP to generate the serialized version of the key value, and RESTORE in order to synthesize the key in the target instance. The source instance acts as a client for the target instance. If the target instance returns OK to the RESTORE command, the source instance deletes the key using DEL.

The timeout specifies the maximum idle time in any moment of the communication with the destination instance in milliseconds. This means that the operation does not need to be completed within the specified amount of milliseconds, but that the transfer should make progresses without blocking for more than the specified amount of milliseconds.

MIGRATE needs to perform I/O operations and to honor the specified timeout. When there is an I/O error during the transfer or if the timeout is reached the operation is aborted and the special error - `IOERR` returned.

When this happens the following two cases are possible:

- The key may be on both the instances.
- The key may be only in the source instance.

It is not possible for the key to get lost in the event of a timeout, but the client calling MIGRATE, in the event of a timeout error, should check if the key is *also* present in the target instance and act accordingly.

When any other error is returned (starting with `ERR`) MIGRATE guarantees that the key is still only present in the originating instance (unless a key with the same name was also *already* present on the target instance).

On success OK is returned.

Return value

Status code reply: The command returns OK on success.

MONITOR

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- INFO
- LASTSAVE
- **MONITOR**
- SAVE
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- SYNC
- TIME

Available since 1.0.0.

MONITOR is a debugging command that streams back every command processed by the Redis server. It can help in understanding what is happening to the database. This command can both be used via `redis-cli` and via `telnet`.

The ability to see all the requests processed by the server is useful in order to spot bugs in an application both when using Redis as a database and as a distributed caching system.

```
$ redis-cli monitor
```

Use `SIGINT` (Ctrl-C) to stop a MONITOR stream running via `redis-cli`.

```
$ telnet localhost 6379
```

Manually issue the QUIT command to stop a MONITOR stream running via `telnet`.

Cost of running MONITOR

Because MONITOR streams back **all** commands, its use comes at a cost. The following (totally unscientific) benchmark numbers illustrate what the cost of running MONITOR can be.

Benchmark result **without** MONITOR running:

```
$ src/redis-benchmark -c 10 -n 100000 -q
```

Benchmark result **with** MONITOR running (`redis-cli monitor > /dev/null`):

```
$ src/redis-benchmark -c 10 -n 100000 -q
```

In this particular case, running a single MONITOR client can reduce the throughput by more than 50%. Running more MONITOR clients will reduce throughput even more.

Return value

Non standard return value, just dumps the received commands in an infinite flow.

MOVE key db

Related commands

- DEL
- DUMP
- EXISTS
- EXPIRE
- EXPIREAT
- KEYS
- MIGRATE
- **MOVE**
- OBJECT
- PERSIST
- PEXPIRE
- PEXPIREAT
- PTTL
- RANDOMKEY
- RENAME
- RENAMENX
- RESTORE
- SORT
- TTL
- TYPE

Available since 1.0.0.

Time complexity: O(1)

Move `key` from the currently selected database (see SELECT) to the specified destination database. When `key` already exists in the destination database, or it does not exist in the source database, it does nothing. It is possible to use MOVE as a locking primitive because of this.

Return value

Integer reply, specifically:

- `1` if `key` was moved.
- `0` if `key` was not moved.

MSET key value [key value ...]

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- DECRBY
- GET
- GETBIT
- GETRANGE
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- **MSET**
- MSETNX
- PSETEX
- SET
- SETBIT
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 1.0.1.

Time complexity: O(N) where N is the number of keys to set.

Sets the given keys to their respective values. MSET replaces existing values with new values, just as regular SET. See MSETNX if you don't want to overwrite existing values.

MSET is atomic, so all given keys are set at once. It is not possible for clients to see that some of the keys were updated while others are unchanged.

Return value

Status code reply: always `OK` since MSET can't fail.

Examples

redis> MSET key1 "Hello" key2 "World"

```
OK
```

redis> GET key1

```
"Hello"
```

redis> GET key2

```
"World"
```

redis>

MSETNX key value [key value ...]

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- DECRBY
- GET
- GETBIT
- GETRANGE
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- **MSETNX**
- PSETEX
- SET
- SETBIT
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 1.0.1.

Time complexity: O(N) where N is the number of keys to set.

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

Because of this semantic MSETNX can be used in order to set different keys representing different fields of an unique logic object in a way that ensures that either all the fields or none at all are set.

MSETNX is atomic, so all given keys are set at once. It is not possible for clients to see that some of the keys were updated while others are unchanged.

Return value

Integer reply, specifically:

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

Examples

redis> MSETNX key1 "Hello" key2 "there"

```
(integer) 1
```

redis> MSETNX key2 "there" key3 "world"

```
(integer) 0
```

redis> MGET key1 key2 key3

```
1) "Hello"
```

redis>

MULTI

Related topics

- Transactions

Related commands

- DISCARD
- EXEC
- **MULTI**
- UNWATCH
- WATCH

Available since 1.2.0.

Marks the start of a transaction block. Subsequent commands will be queued for atomic execution using EXEC.

Return value

Status code reply: always `OK`.

OBJECT subcommand [arguments [arguments ...]]

Related commands

- DEL
- DUMP
- EXISTS
- EXPIRE
- EXPIREAT
- KEYS
- MIGRATE
- MOVE
- **OBJECT**
- PERSIST
- PEXPIRE
- PEXPIREAT
- PTTL
- RANDOMKEY
- RENAME
- RENAMENX
- RESTORE
- SORT
- TTL
- TYPE

Available since 2.2.3.

Time complexity: O(1) for all the currently implemented subcommands.

The OBJECT command allows to inspect the internals of Redis Objects associated with keys. It is useful for debugging or to understand if your keys are using the specially encoded data types to save space. Your application may also use the information reported by the OBJECT command to implement application level key eviction policies when using Redis as a Cache.

The OBJECT command supports multiple sub commands:

- `OBJECT REFCOUNT <key>` returns the number of references of the value associated with the specified key. This command is mainly useful for debugging.
- `OBJECT ENCODING <key>` returns the kind of internal representation used in order to store the value associated with a key.
- `OBJECT IDLETIME <key>` returns the number of seconds since the object stored at the specified key is idle (not requested by read or write operations). While the value is returned in seconds the actual resolution of this timer is 10 seconds, but may vary in future implementations.

Objects can be encoded in different ways:

- Strings can be encoded as `raw` (normal string encoding) or `int` (strings representing integers in a 64 bit signed interval are encoded in this way in order to save space).

- Lists can be encoded as `ziplist` or `linkedlist`. The `ziplist` is the special representation that is used to save space for small lists.
- Sets can be encoded as `intset` or `hashtable`. The `intset` is a special encoding used for small sets composed solely of integers.
- Hashes can be encoded as `zipmap` or `hashtable`. The `zipmap` is a special encoding used for small hashes.
- Sorted Sets can be encoded as `ziplist` or `skiplist` format. As for the List type small sorted sets can be specially encoded using `ziplist`, while the `skiplist` encoding is the one that works with sorted sets of any size.

All the specially encoded types are automatically converted to the general type once you perform an operation that makes it no possible for Redis to retain the space saving encoding.

Return value

Different return values are used for different subcommands.

- Subcommands `refcount` and `idletime` returns integers.
- Subcommand `encoding` returns a bulk reply.

If the object you try to inspect is missing, a null bulk reply is returned.

Examples

```
redis> lpush mylist "Hello World"
```

In the following example you can see how the encoding changes once Redis is no longer able to use the space saving encoding.

```
redis> set foo 1000
```

PERSIST key

Related commands

- DEL
- DUMP
- EXISTS
- EXPIRE
- EXPIREAT
- KEYS
- MIGRATE
- MOVE
- OBJECT
- **PERSIST**
- PEXPIRE
- PEXPIREAT
- PTTL
- RANDOMKEY
- RENAME
- RENAMENX
- RESTORE
- SORT
- TTL
- TYPE

Available since 2.2.0.

Time complexity: 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).

Return value

Integer reply, specifically:

- `1` if the timeout was removed.
- `0` if `key` does not exist or does not have an associated timeout.

Examples

redis> SET mykey "Hello"

```
OK
```

redis> EXPIRE mykey 10

```
(integer) 1
```

redis> TTL mykey

```
(integer) 10
```

redis> PERSIST mykey

```
(integer) 1
```

redis> TTL mykey

```
(integer) -1
```

redis>

PEXPIRE key milliseconds

Related commands

- DEL
- DUMP
- EXISTS
- EXPIRE
- EXPIREAT
- KEYS
- MIGRATE
- MOVE
- OBJECT
- PERSIST
- **PEXPIRE**
- PEXPIREAT
- PTTL
- RANDOMKEY
- RENAME
- RENAMENX
- RESTORE
- SORT
- TTL
- TYPE

Available since 2.6.0.

Time complexity: O(1)

This command works exactly like EXPIRE but the time to live of the key is specified in milliseconds instead of seconds.

Integer reply, specifically:

- `1` if the timeout was set.
- `0` if `key` does not exist or the timeout could not be set.

Examples

redis> SET mykey "Hello"

```
OK
```

redis> PEXPIRE mykey 1500

```
(integer) 1
```

redis> TTL mykey

```
(integer) 1
```

redis> PTTL mykey

```
(integer) 1497
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

PEXPIREAT key milliseconds-timestamp

Related commands

- DEL
- DUMP
- EXISTS
- EXPIRE
- EXPIREAT
- KEYS
- MIGRATE
- MOVE
- OBJECT
- PERSIST
- PEXPIRE
- **PEXPIREAT**
- PTTL
- RANDOMKEY
- RENAME
- RENAMENX
- RESTORE
- SORT
- TTL
- TYPE

Available since 2.6.0.

Time complexity: O(1)

PEXPIREAT has the same effect and semantic as EXPIREAT, but the Unix time at which the key will expire is specified in milliseconds instead of seconds.

Return value

Integer reply, specifically:

- `1` if the timeout was set.
- `0` if `key` does not exist or the timeout could not be set (see: EXPIRE).

Examples

redis> SET mykey "Hello"

```
OK
```

redis> PEXPIREAT mykey 1555555555005

```
(integer) 1
```

redis> TTL mykey

```
(integer) 205365180
```

redis> PTTL mykey

```
(integer) 205365180077
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

PING

Related commands

- AUTH
- ECHO
- **PING**
- QUIT
- SELECT

Available since 1.0.0.

Returns `PONG`. This command is often used to test if a connection is still alive, or to measure latency.

Return value

Status code reply

Examples

redis> PING

```
PONG
```

redis>

PSETEX key milliseconds value

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- DECRBY
- GET
- GETBIT
- GETRANGE
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- **PSETEX**
- SET
- SETBIT
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 2.6.0.

Time complexity: O(1)

PSETEX works exactly like SETEX with the sole difference that the expire time is specified in milliseconds instead of seconds.

Examples

redis> PSETEX mykey 1000 "Hello"

```
OK
```

redis> PTTL mykey

```
(integer) 998
```

redis> GET mykey

```
"Hello"
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by 

PSUBSCRIBE pattern [pattern ...]

Related topics

- Pub/Sub

Related commands

- **PSUBSCRIBE**
- PUBLISH
- PUNSUBSCRIBE
- SUBSCRIBE
- UNSUBSCRIBE

Available since 2.0.0.

Time complexity: O(N) where N is the number of patterns the client is already subscribed to.

Subscribes the client to the given patterns.

PTTL key

Related commands

- DEL
- DUMP
- EXISTS
- EXPIRE
- EXPIREAT
- KEYS
- MIGRATE
- MOVE
- OBJECT
- PERSIST
- PEXPIRE
- PEXPIREAT
- **PTTL**
- RANDOMKEY
- RENAME
- RENAMENX
- RESTORE
- SORT
- TTL
- TYPE

Available since 2.6.0.

Time complexity: O(1)

Like TTL this command returns the remaining time to live of a key that has an expire set, with the sole difference that TTL returns the amount of remaining time in seconds while PTTL returns it in milliseconds.

Return value

Integer reply: Time to live in milliseconds or `-1` when `key` does not exist or does not have a timeout.

Examples

redis> SET mykey "Hello"

```
OK
```

redis> EXPIRE mykey 1

```
(integer) 1
```

redis> PTTL mykey

```
(integer) 999
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

PUBLISH channel message

Related topics

- Pub/Sub

Related commands

- PSUBSCRIBE
- **PUBLISH**
- PUNSUBSCRIBE
- SUBSCRIBE
- UNSUBSCRIBE

Available since 2.0.0.

Time complexity: O(N+M) where N is the number of clients subscribed to the receiving channel and M is the total number of subscribed patterns (by any client).

Posts a message to the given channel.

Return value

Integer reply: the number of clients that received the message.

PUNSUBSCRIBE [pattern [pattern ...]]

Related topics

- Pub/Sub

Related commands

- PSUBSCRIBE
- PUBLISH
- **PUNSUBSCRIBE**
- SUBSCRIBE
- UNSUBSCRIBE

Available since 2.0.0.

Time complexity: O(N+M) where N is the number of patterns the client is already subscribed and M is the number of total patterns subscribed in the system (by any client).

Unsubscribes the client from the given patterns, or from all of them if none is given.

When no patters are specified, the client is unsubscribed from all the previously subscribed patterns. In this case, a message for every unsubscribed pattern will be sent to the client.

QUIT

Related commands

- AUTH
- ECHO
- PING
- **QUIT**
- SELECT

Available since 1.0.0.

Ask the server to close the connection. The connection is closed as soon as all pending replies have been written to the client.

Return value

Status code reply: always OK.

RANDOMKEY

Related commands

- DEL
- DUMP
- EXISTS
- EXPIRE
- EXPIREAT
- KEYS
- MIGRATE
- MOVE
- OBJECT
- PERSIST
- PEXPIRE
- PEXPIREAT
- PTTL
- **RANDOMKEY**
- RENAME
- RENAMENX
- RESTORE
- SORT
- TTL
- TYPE

Available since 1.0.0.

Time complexity: O(1)

Return a random key from the currently selected database.

Return value

Bulk reply: the random key, or `nil` when the database is empty.

RENAME key newkey

Related commands

- DEL
- DUMP
- EXISTS
- EXPIRE
- EXPIREAT
- KEYS
- MIGRATE
- MOVE
- OBJECT
- PERSIST
- PEXPIRE
- PEXPIREAT
- PTTL
- RANDOMKEY
- **RENAME**
- RENAMENX
- RESTORE
- SORT
- TTL
- TYPE

Available since 1.0.0.

Time complexity: O(1)

Renames `key` to `newkey`. It returns an error when the source and destination names are the same, or when `key` does not exist. If `newkey` already exists it is overwritten.

Return value

Status code reply

Examples

redis> SET mykey "Hello"

```
OK
```

redis> RENAME mykey myotherkey

```
OK
```

redis> GET myotherkey

```
"Hello"
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

RENAMENX key newkey

Related commands

- DEL
- DUMP
- EXISTS
- EXPIRE
- EXPIREAT
- KEYS
- MIGRATE
- MOVE
- OBJECT
- PERSIST
- PEXPIRE
- PEXPIREAT
- PTTL
- RANDOMKEY
- RENAME
- **RENAMENX**
- RESTORE
- SORT
- TTL
- TYPE

Available since 1.0.0.

Time complexity: O(1)

Renames `key` to `newkey` if `newkey` does not yet exist. It returns an error under the same conditions as RENAME.

Return value

Integer reply, specifically:

- `1` if `key` was renamed to `newkey`.
- `0` if `newkey` already exists.

Examples

redis> SET mykey "Hello"

```
OK
```

redis> SET myotherkey "World"

```
OK
```

redis> RENAMENX mykey myotherkey

```
(integer) 0
```

redis> GET myotherkey

```
"World"
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by 

RESTORE key ttl serialized-value

Related commands

- DEL
- DUMP
- EXISTS
- EXPIRE
- EXPIREAT
- KEYS
- MIGRATE
- MOVE
- OBJECT
- PERSIST
- PEXPIRE
- PEXPIREAT
- PTTL
- RANDOMKEY
- RENAME
- RENAMENX
- **RESTORE**
- SORT
- TTL
- TYPE

Available since 2.6.0.

Time complexity: O(1) to create the new key and additional O(N*M) to recostruct the serialized value, where N is the number of Redis objects composing the value and M their average size. For small string values the time complexity is thus O(1)+O(1*M) where M is small, so simply O(1). However for sorted set values the complexity is O(N*M*log(N)) because inserting values into sorted sets is O(log(N)).

Create a key associated with a value that is obtained by deserializing the provided serialized value (obtained via DUMP).

If `ttl` is 0 the key is created without any expire, otherwise the specified expire time (in milliseconds) is set.

RESTORE checks the RDB version and data checksum. If they don't match an error is returned.

Return value

Status code reply: The command returns OK on success.

Examples

```
redis> DEL mykey
```

Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

RPOP key

Related commands

- BLPOP
- BRPOP
- BRPOPLPUSH
- LINDEX
- LINSERT
- LLEN
- LPOP
- LPUSH
- LPUSHX
- LRANGE
- LREM
- LSET
- LTRIM
- **RPOP**
- RPOPLPUSH
- RPUSH
- RPUSHX

Available since 1.0.0.

Time complexity: O(1)

Removes and returns the last element of the list stored at `key`.

Return value

Bulk reply: the value of the last element, or `nil` when `key` does not exist.

Examples

redis> RPUSH mylist "one"

```
(integer) 1
```

redis> RPUSH mylist "two"

```
(integer) 2
```

redis> RPUSH mylist "three"

```
(integer) 3
```

redis> RPOP mylist

```
"three"
```

redis> LRANGE mylist 0 -1

```
1) "one"
```

redis>

RPOPLPUSH source destination

Related commands

- BLPOP
- BRPOP
- BRPOPLPUSH
- LINDEX
- LINSERT
- LLEN
- LPOP
- LPUSH
- LPUSHX
- LRANGE
- LREM
- LSET
- LTRIM
- RPOP
- **RPOPLPUSH**
- RPUSH
- RPUSHX

Available since 1.2.0.

Time complexity: O(1)

Atomically returns and removes the last element (tail) of the list stored at `source`, and pushes the element at the first element (head) 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 RPOPLPUSH 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 last element from the list and pushing it as first element of the list, so it can be considered as a list rotation command.

Return value

Bulk reply: the element being popped and pushed.

Examples

redis> RPUSH mylist "one"

```
(integer) 1
```

redis> RPUSH mylist "two"

```
(integer) 2
```

redis> RPUSH mylist "three"

```
(integer) 3
```

redis> RPOPLPUSH mylist myotherlist

```
"three"
```

redis> LRANGE mylist 0 -1

```
1) "one"
```

redis> LRANGE myotherlist 0 -1

```
1) "three"
```

redis>

Pattern: Reliable queue

Redis is often used as a messaging server to implement processing of background jobs or other kinds of messaging tasks. A simple form of queue is often obtained pushing values into a list in the producer side, and waiting for this values in the consumer side using RPOP (using polling), or BRPOP if the client is better served by a blocking operation.

However in this context the obtained queue is not *reliable* as messages can be lost, for example in the case there is a network problem or if the consumer crashes just after the message is received but it is still to process.

RPOPLPUSH (or BRPOPLPUSH for the blocking variant) offers a way to avoid this problem: the consumer fetches the message and at the same time pushes it into a *processing* list. It will use the LREM command in order to remove the message from the *processing* list once the message has been processed.

An additional client may monitor the *processing* list for items that remain there for too much time, and will push those timed out items into the queue again if needed.

Pattern: Circular list

Using RPOPLPUSH with the same source and destination key, a client can visit all the elements of an N-elements list, one after the other, in O(N) without transferring the full list from the server to the client using a single LRANGE operation.

The above pattern works even if the following two conditions: * There are multiple clients rotating the list: they'll fetch different elements, until all the elements of the list are visited, and the process restarts. * Even if other clients are actively pushing new items at the end of the list.

The above makes it very simple to implement a system where a set of items must be processed by N workers continuously as fast as possible. An example is a monitoring system that must check that a set of web sites are reachable, with the smallest delay possible, using a number of parallel workers.

Note that this implementation of workers is trivially scalable and reliable, because even if a message is lost the item is still in the queue and will be processed at the next iteration.

RPUSH key value [value ...]

Related commands

- BLPOP
- BRPOP
- BRPOPLPUSH
- LINDEX
- LINSERT
- LLEN
- LPOP
- LPUSH
- LPUSHX
- LRANGE
- LREM
- LSET
- LTRIM
- RPOP
- RPOPLPUSH
- **RPUSH**
- RPUSHX

Available since 1.0.0.

Time complexity: 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.

It is possible to push multiple elements using a single command call just specifying multiple arguments at the end of the command. Elements are inserted one after the other to the tail of the list, from the leftmost element to the rightmost element. So for instance the command `RPUSH mylist a b c` will result into a list containing `a` as first element, `b` as second element and `c` as third element.

Return value

Integer reply: the length of the list after the push operation.

History

- `>= 2.4`: Accepts multiple `value` arguments. In Redis versions older than 2.4 it was possible to push a single value per command.

Examples

redis> RPUSH mylist "hello"

```
(integer) 1
```

redis> RPUSH mylist "world"

```
(integer) 2
```

redis> LRANGE mylist 0 -1

```
1) "hello"
```

redis>

RPUSHX key value

Related commands

- BLPOP
- BRPOP
- BRPOPLPUSH
- LINDEX
- LINSERT
- LLEN
- LPOP
- LPUSH
- LPUSHX
- LRANGE
- LREM
- LSET
- LTRIM
- RPOP
- RPOPLPUSH
- RPUSH
- **RPUSHX**

Available since 2.2.0.

Time complexity: O(1)

Inserts `value` 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.

Return value

Integer reply: the length of the list after the push operation.

Examples

redis> RPUSH mylist "Hello"

```
(integer) 1
```

redis> RPUSHX mylist "World"

```
(integer) 2
```

redis> RPUSHX myotherlist "World"

```
(integer) 0
```

redis> LRANGE mylist 0 -1

```
1) "Hello"
```

redis> LRANGE myotherlist 0 -1

```
(empty list or set)
```

redis>

SADD key member [member ...]

Related commands

- **SADD**
- SCARD
- SDIFF
- SDIFFSTORE
- SINTER
- SINTERSTORE
- SISMEMBER
- SMEMBERS
- SMOVE
- SPOP
- SRANDMEMBER
- SREM
- SUNION
- SUNIONSTORE

Available since 1.0.0.

Time complexity: O(N) where N is the number of members to be added.

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.

An error is returned when the value stored at `key` is not a set.

Return value

Integer reply: the number of elements that were added to the set, not including all the elements already present into the set.

History

- `>= 2.4`: Accepts multiple `member` arguments. Redis versions before 2.4 are only able to add a single member per call.

Examples

redis> SADD myset "Hello"

```
(integer) 1
```

redis> SADD myset "World"

```
(integer) 1
```

redis> SADD myset "World"

```
(integer) 0
```

redis> SMEMBERS myset

```
1) "World"
```

redis>

SAVE

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- INFO
- LASTSAVE
- MONITOR
- **SAVE**
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- SYNC
- TIME

Available since 1.0.0.

The SAVE commands performs a **synchronous** save of the dataset producing a *point in time* snapshot of all the data inside the Redis instance, in the form of an RDB file.

You almost never want to call SAVE in production environments where it will block all the other clients. Instead usually BGSAVE is used. However in case of issues preventing Redis to create the background saving child (for instance errors in the fork(2) system call), the SAVE command can be a good last resort to perform the dump of the latest dataset.

Please refer to the persistence documentation for detailed information.

Return value

Status code reply: The commands returns OK on success.

SCARD key

Related commands

- SADD
- **SCARD**
- SDIFF
- SDIFFSTORE
- SINTER
- SINTERSTORE
- SISMEMBER
- SMEMBERS
- SMOVE
- SPOP
- SRANDMEMBER
- SREM
- SUNION
- SUNIONSTORE

Available since 1.0.0.

Time complexity: O(1)

Returns the set cardinality (number of elements) of the set stored at `key`.

Return value

Integer reply: the cardinality (number of elements) of the set, or `0` if `key` does not exist.

Examples

redis> SADD myset "Hello"

```
(integer) 1
```

redis> SADD myset "World"

```
(integer) 1
```

redis> SCARD myset

```
(integer) 2
```

redis>

SCRIPT EXISTS script [script ...]

Related commands

- EVAL
- EVALSHA
- **SCRIPT EXISTS**
- SCRIPT FLUSH
- SCRIPT KILL
- SCRIPT LOAD

Available since 2.6.0.

Time complexity: O(N) with N being the number of scripts to check (so checking a single script is an O(1) operation).

Returns information about the existence of the scripts in the script cache.

This command accepts one or more SHA1 digests and returns a list of ones or zeros to signal if the scripts are already defined or not inside the script cache. This can be useful before a pipelining operation to ensure that scripts are loaded (and if not, to load them using `SCRIPT LOAD`) so that the pipelining operation can be performed solely using EVALSHA instead of EVAL to save bandwidth.

Please refer to the EVAL documentation for detailed information about Redis Lua scripting.

Return value

Multi-bulk reply The command returns an array of integers that correspond to the specified SHA1 digest arguments. For every corresponding SHA1 digest of a script that actually exists in the script cache, an 1 is returned, otherwise 0 is returned.

redis> SCRIPT LOAD "return 1"

```
ERR Unknown or disabled command 'SCRIPT'
```

redis> SCRIPT EXISTS e0e1f9fabfc9d4800c877a703b823ac0578ff8db ffffffffffffffffffffffffffffffffffffffff

```
ERR Unknown or disabled command 'SCRIPT'
```

redis>

SCRIPT FLUSH

Related commands

- EVAL
- EVALSHA
- SCRIPT EXISTS
- **SCRIPT FLUSH**
- SCRIPT KILL
- SCRIPT LOAD

Available since 2.6.0.

Time complexity: O(N) with N being the number of scripts in cache

Flush the Lua scripts cache.

Please refer to the EVAL documentation for detailed information about Redis Lua scripting.

Return value

Status code reply

SCRIPT KILL

Related commands

- EVAL
- EVALSHA
- SCRIPT EXISTS
- SCRIPT FLUSH
- **SCRIPT KILL**
- SCRIPT LOAD

Available since 2.6.0.

Time complexity: O(1)

Kills the currently executing Lua script, assuming no write operation was yet performed by the script.

This command is mainly useful to kill a script that is running for too much time(for instance because it entered an infinite loop because of a bug). The script will be killed and the client currently blocked into EVAL will see the command returning with an error.

If the script already performed write operations it can not be killed in this way because it would violate Lua script atomicity contract. In such a case only `SHUTDOWN NOSAVE` is able to kill the script, killing the Redis process in an hard way preventing it to persist with half-written information.

Please refer to the EVAL documentation for detailed information about Redis Lua scripting.

Return value

Status code reply

SCRIPT LOAD script

Related commands

- EVAL
- EVALSHA
- SCRIPT EXISTS
- SCRIPT FLUSH
- SCRIPT KILL
- **SCRIPT LOAD**

Available since 2.6.0.

Time complexity: O(N) with N being the length in bytes of the script body.

Load a script into the scripts cache, without executing it. After the specified command is loaded into the script cache it will be callable using EVALSHA with the correct SHA1 digest of the script, exactly like after the first successful invocation of EVAL.

The script is guaranteed to stay in the script cache forever (unless `SCRIPT FLUSH` is called).

The command works in the same way even if the script was already present in the script cache.

Please refer to the EVAL documentation for detailed information about Redis Lua scripting.

Return value

Bulk reply This command returns the SHA1 digest of the script added into the script cache.

SDIFF key [key ...]

Related commands

- SADD
- SCARD
- **SDIFF**
- SDIFFSTORE
- SINTER
- SINTERSTORE
- SISMEMBER
- SMEMBERS
- SMOVE
- SPOP
- SRANDMEMBER
- SREM
- SUNION
- SUNIONSTORE

Available since 1.0.0.

Time complexity: O(N) where N is the total number of elements in all given sets.

Returns the members of the set resulting from the difference between the first set and all the successive sets.

For example:

```
key1 = {a,b,c,d}
```

Keys that do not exist are considered to be empty sets.

Return value

Multi-bulk reply: list with members of the resulting set.

Examples

redis> SADD key1 "a"

```
(integer) 1
```

redis> SADD key1 "b"

```
(integer) 1
```

redis> SADD key1 "c"

```
(integer) 1
```

```
redis> SADD key2 "c"
(integer) 1
redis> SADD key2 "d"
(integer) 1
redis> SADD key2 "e"
(integer) 1
redis> SDIFF key1 key2
1) "b"
redis>
```

SDIFFSTORE destination key [key ...]

Related commands

- SADD
- SCARD
- SDIFF
- **SDIFFSTORE**
- SINTER
- SINTERSTORE
- SISMEMBER
- SMEMBERS
- SMOVE
- SPOP
- SRANDMEMBER
- SREM
- SUNION
- SUNIONSTORE

Available since 1.0.0.

Time complexity: O(N) where N is the total number of elements in all given sets.

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.

Return value

Integer reply: the number of elements in the resulting set.

SELECT index

Related commands

- AUTH
- ECHO
- PING
- QUIT
- **SELECT**

Available since 1.0.0.

Select the DB with having the specified zero-based numeric index. New connections always use DB 0.

Return value

Status code reply

SET key value

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- DECRBY
- GET
- GETBIT
- GETRANGE
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- PSETEX
- **SET**
- SETBIT
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 1.0.0.

Time complexity: O(1)

Set `key` to hold the string `value`. If `key` already holds a value, it is overwritten, regardless of its type.

Return value

Status code reply: always `OK` since SET can't fail.

Examples

redis> SET mykey "Hello"

```
OK
```

redis> GET mykey

```
"Hello"
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by 

SETBIT key offset value

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- DECRBY
- GET
- GETBIT
- GETRANGE
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- PSETEX
- SET
- **SETBIT**
- SETEX
- SETNX
- SETRANGE
- STRLEN

Available since 2.2.0.

Time complexity: O(1)

Sets or clears the bit at *offset* in the string value stored at *key*.

The bit is either set or cleared depending on *value*, which can be either 0 or 1. When *key* does not exist, a new string value is created. The string is grown to make sure it can hold a bit at *offset*. The *offset* argument is required to be greater than or equal to 0, and smaller than 2^{32} (this limits bitmaps to 512MB). When the string at *key* is grown, added bits are set to 0.

Warning: When setting the last possible bit (*offset* equal to 2^{32} -1) and the string value stored at *key* does not yet hold a string value, or holds a small string value, Redis needs to allocate all intermediate memory which can block the server for some time. On a 2010 MacBook Pro, setting bit number 2^{32} -1 (512MB allocation) takes ~300ms, setting bit number 2^{30} -1 (128MB allocation) takes ~80ms, setting bit number 2^{28} -1 (32MB allocation) takes ~30ms and setting bit number 2^{26} -1 (8MB allocation) takes ~8ms. Note that once this first allocation is done, subsequent calls to SETBIT for the same *key* will not have the allocation overhead.

Return value

Integer reply: the original bit value stored at *offset*.

Examples

redis> SETBIT mykey 7 1

```
(integer) 0
```

redis> SETBIT mykey 7 0

```
(integer) 1
```

redis> GET mykey

```
"\u0000"
```

redis>

SETEX key seconds value

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- DECRBY
- GET
- GETBIT
- GETRANGE
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- PSETEX
- SET
- SETBIT
- **SETEX**
- SETNX
- SETRANGE
- STRLEN

Available since 2.0.0.

Time complexity: O(1)

Set `key` to hold the string `value` and set `key` to timeout after a given number of seconds. This command is equivalent to executing the following commands:

```
SET mykey value
```

SETEX is atomic, and can be reproduced by using the previous two commands inside an MULTI / EXEC block. It is provided as a faster alternative to the given sequence of operations, because this operation is very common when Redis is used as a cache.

An error is returned when `seconds` is invalid.

Return value

Status code reply

Examples

redis> SETEX mykey 10 "Hello"

```
OK
```

redis> TTL mykey

```
(integer) 10
```

redis> GET mykey

```
"Hello"
```

redis>

SETNX key value

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- DECRBY
- GET
- GETBIT
- GETRANGE
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- PSETEX
- SET
- SETBIT
- SETEX
- **SETNX**
- SETRANGE
- STRLEN

Available since 1.0.0.

Time complexity: O(1)

Set `key` to hold string `value` if `key` does not exist. In that case, it is equal to SET. When `key` already holds a value, no operation is performed. SETNX is short for "**SET** if **N** ot e **X** ists".

Return value

Integer reply, specifically:

- `1` if the key was set
- `0` if the key was not set

Examples

redis> SETNX mykey "Hello"

```
(integer) 1
```

```
redis> SETNX mykey "World"

(integer) 0

redis> GET mykey

"Hello"

redis>
```

Design pattern: Locking with `SETNX`

SETNX can be used as a locking primitive. For example, to acquire the lock of the key `foo`, the client could try the following:

```
SETNX lock.foo <current Unix time + lock timeout + 1>
```

If SETNX returns `1` the client acquired the lock, setting the `lock.foo` key to the Unix time at which the lock should no longer be considered valid. The client will later use `DEL lock.foo` in order to release the lock.

If SETNX returns `0` the key is already locked by some other client. We can either return to the caller if it's a non blocking lock, or enter a loop retrying to hold the lock until we succeed or some kind of timeout expires.

Handling deadlocks

In the above locking algorithm there is a problem: what happens if a client fails, crashes, or is otherwise not able to release the lock? It's possible to detect this condition because the lock key contains a UNIX timestamp. If such a timestamp is equal to the current Unix time the lock is no longer valid.

When this happens we can't just call DEL against the key to remove the lock and then try to issue a SETNX, as there is a race condition here, when multiple clients detected an expired lock and are trying to release it.

- C1 and C2 read `lock.foo` to check the timestamp, because they both received `0` after executing SETNX, as the lock is still held by C3 that crashed after holding the lock.
- C1 sends `DEL lock.foo`
- C1 sends `SETNX lock.foo` and it succeeds
- C2 sends `DEL lock.foo`
- C2 sends `SETNX lock.foo` and it succeeds
- **ERROR**: both C1 and C2 acquired the lock because of the race condition.

Fortunately, it's possible to avoid this issue using the following algorithm. Let's see how C4, our sane client, uses the good algorithm:

- C4 sends `SETNX lock.foo` in order to acquire the lock
- The crashed client C3 still holds it, so Redis will reply with `0` to C4.
- C4 sends `GET lock.foo` to check if the lock expired. If it is not, it will sleep for some time and retry from the start.
- Instead, if the lock is expired because the Unix time at `lock.foo` is older than the current Unix time, C4 tries to perform:

```
GETSET lock.foo <current Unix timestamp + lock timeout + 1>
```

- Because of the GETSET semantic, C4 can check if the old value stored at `key` is still an expired timestamp. If it is, the lock was acquired.
- If another client, for instance C5, was faster than C4 and acquired the lock with the GETSET operation, the C4 GETSET operation will return a non expired timestamp. C4 will simply restart from the first step. Note that even if C4 set the key a bit a few seconds in the future this is not a problem.

Important note: In order to make this locking algorithm more robust, a client holding a lock should always check the timeout didn't expire before unlocking the key with DEL because client failures can be complex, not just crashing but also blocking a lot of time against some operations and trying to issue DEL after a lot of time (when the LOCK is already held by another client).

SETRANGE key offset value

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- DECRBY
- GET
- GETBIT
- GETRANGE
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- PSETEX
- SET
- SETBIT
- SETEX
- SETNX
- **SETRANGE**
- STRLEN

Available since 2.2.0.

Time complexity: O(1), not counting the time taken to copy the new string in place. Usually, this string is very small so the amortized complexity is O(1). Otherwise, complexity is O(M) with M being the length of the value argument.

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 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*.

Note that the maximum offset that you can set is 2^{29} -1 (536870911), as Redis Strings are limited to 512 megabytes. If you need to grow beyond this size, you can use multiple keys.

Warning: When setting the last possible byte and the string value stored at *key* does not yet hold a string value, or holds a small string value, Redis needs to allocate all intermediate memory which can block the server for some time. On a 2010 MacBook Pro, setting byte number 536870911 (512MB allocation) takes ~300ms, setting byte number 134217728 (128MB allocation) takes ~80ms, setting bit number 33554432 (32MB allocation) takes ~30ms and setting bit number 8388608 (8MB allocation) takes ~8ms. Note that once this first allocation is done, subsequent calls to SETRANGE for the same *key* will not have the allocation overhead.

Patterns

Thanks to SETRANGE and the analogous GETRANGE commands, you can use Redis strings as a linear array with O(1) random access. This is a very fast and efficient storage in many real world use cases.

Return value

Integer reply: the length of the string after it was modified by the command.

Examples

Basic usage:

redis> SET key1 "Hello World"

```
OK
```

redis> SETRANGE key1 6 "Redis"

```
(integer) 11
```

redis> GET key1

```
"Hello Redis"
```

redis>
Example of zero padding:

redis> SETRANGE key2 6 "Redis"

```
(integer) 11
```

redis> GET key2

```
"\u0000\u0000\u0000\u0000\u0000\u0000Redis"
```

redis>

SHUTDOWN [NOSAVE] [SAVE]

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- INFO
- LASTSAVE
- MONITOR
- SAVE
- **SHUTDOWN**
- SLAVEOF
- SLOWLOG
- SYNC
- TIME

Available since 1.0.0.

The command behavior is the following:

- Stop all the clients.
- Perform a blocking SAVE if at least one **save point** is configured.
- Flush the Append Only File if AOF is enabled.
- Quit the server.

If persistence is enabled this commands makes sure that Redis is switched off without the lost of any data. This is not guaranteed if the client uses simply SAVE and then QUIT because other clients may alter the DB data between the two commands.

Note: A Redis instance that is configured for not persisting on disk (no AOF configured, nor "save" directive) will not dump the RDB file on SHUTDOWN, as usually you don't want Redis instances used only for caching to block on when shutting down.

SAVE and NOSAVE modifiers

It is possible to specify an optional modifier to alter the behavior of the command. Specifically:

- **SHUTDOWN SAVE** will force a DB saving operation even if no save points are configured.
- **SHUTDOWN NOSAVE** will prevent a DB saving operation even if one or more save points are configured. (You can think at this variant as an hypothetical **ABORT** command that just stops the server).

Return value

Status code reply on error. On success nothing is returned since the server quits and the connection is closed.

SINTER key [key ...]

Related commands

- SADD
- SCARD
- SDIFF
- SDIFFSTORE
- **SINTER**
- SINTERSTORE
- SISMEMBER
- SMEMBERS
- SMOVE
- SPOP
- SRANDMEMBER
- SREM
- SUNION
- SUNIONSTORE

Available since 1.0.0.

Time complexity: O(N*M) worst case where N is the cardinality of the smallest set and M is the number of sets.

Returns the members of the set resulting from the intersection of all the given sets.

For example:

```
key1 = {a,b,c,d}
```

Keys that do not exist are considered to be empty sets. With one of the keys being an empty set, the resulting set is also empty (since set intersection with an empty set always results in an empty set).

Return value

Multi-bulk reply: list with members of the resulting set.

Examples

redis> SADD key1 "a"

```
(integer) 1
```

redis> SADD key1 "b"

```
(integer) 1
```

```
redis> SADD key1 "c"
(integer) 1
redis> SADD key2 "c"
(integer) 1
redis> SADD key2 "d"
(integer) 1
redis> SADD key2 "e"
(integer) 1
redis> SINTER key1 key2
1) "c"
redis>
```

SINTERSTORE destination key [key ...]

Related commands

- SADD
- SCARD
- SDIFF
- SDIFFSTORE
- SINTER
- **SINTERSTORE**
- SISMEMBER
- SMEMBERS
- SMOVE
- SPOP
- SRANDMEMBER
- SREM
- SUNION
- SUNIONSTORE

Available since 1.0.0.

Time complexity: O(N*M) worst case where N is the cardinality of the smallest set and M is the number of sets.

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.

Return value

Integer reply: the number of elements in the resulting set.

SISMEMBER key member

Related commands

- SADD
- SCARD
- SDIFF
- SDIFFSTORE
- SINTER
- SINTERSTORE
- **SISMEMBER**
- SMEMBERS
- SMOVE
- SPOP
- SRANDMEMBER
- SREM
- SUNION
- SUNIONSTORE

Available since 1.0.0.

Time complexity: O(1)

Returns if `member` is a member of the set stored at `key`.

Return value

Integer reply, specifically:

- `1` if the element is a member of the set.
- `0` if the element is not a member of the set, or if `key` does not exist.

Examples

redis> SADD myset "one"

```
(integer) 1
```

redis> SISMEMBER myset "one"

```
(integer) 1
```

redis> SISMEMBER myset "two"

```
(integer) 0
```

redis>

Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

SLAVEOF host port

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- INFO
- LASTSAVE
- MONITOR
- SAVE
- SHUTDOWN
- **SLAVEOF**
- SLOWLOG
- SYNC
- TIME

Available since 1.0.0.

The SLAVEOF command can change the replication settings of a slave on the fly. If a Redis server is already acting as slave, the command SLAVEOF NO ONE will turn off the replication, turning the Redis server into a MASTER. In the proper form SLAVEOF hostname port will make the server a slave of another server listening at the specified hostname and port.

If a server is already a slave of some master, SLAVEOF hostname port will stop the replication against the old server and start the synchronization against the new one, discarding the old dataset.

The form SLAVEOF NO ONE will stop replication, turning the server into a MASTER, but will not discard the replication. So, if the old master stops working, it is possible to turn the slave into a master and set the application to use this new master in read/write. Later when the other Redis server is fixed, it can be reconfigured to work as a slave.

Return value

Status code reply

The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

SLOWLOG subcommand [argument]

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- INFO
- LASTSAVE
- MONITOR
- SAVE
- SHUTDOWN
- SLAVEOF
- **SLOWLOG**
- SYNC
- TIME

Available since 2.2.12.

This command is used in order to read and reset the Redis slow queries log.

Redis slow log overview

The Redis Slow Log is a system to log queries that exceeded a specified execution time. The execution time does not include I/O operations like talking with the client, sending the reply and so forth, but just the time needed to actually execute the command (this is the only stage of command execution where the thread is blocked and can not serve other requests in the meantime).

You can configure the slow log with two parameters: *slowlog-log-slower-than* tells Redis what is the execution time, in microseconds, to exceed in order for the command to get logged. Note that a negative number disables the slow log, while a value of zero forces the logging of every command. *slowlog-max-len* is the length of the slow log. The minimum value is zero. When a new command is logged and the slow log is already at its maximum length, the oldest one is removed from the queue of logged commands in order to make space.

The configuration can be done by editing `redis.conf` or while the server is running using the `CONFIG GET` and `CONFIG SET` commands.

Reading the slow log

The slow log is accumulated in memory, so no file is written with information about the slow command executions. This makes the slow log remarkably fast at the point that you can enable the logging of all the commands (setting the *slowlog-log-slower-than* config parameter to zero) with minor performance hit.

To read the slow log the **SLOWLOG GET** command is used, that returns every entry in the slow log. It is possible to return only the N most recent entries passing an additional argument to the command (for instance **SLOWLOG GET 10**).

Note that you need a recent version of redis-cli in order to read the slow log output, since it uses some features of the protocol that were not formerly implemented in redis-cli (deeply nested multi bulk replies).

Output format

```
redis 127.0.0.1:6379> slowlog get 2
```

Every entry is composed of four fields:

- A unique progressive identifier for every slow log entry.
- The unix timestamp at which the logged command was processed.
- The amount of time needed for its execution, in microseconds.
- The array composing the arguments of the command.

The entry's unique ID can be used in order to avoid processing slow log entries multiple times (for instance you may have a script sending you an email alert for every new slow log entry).

The ID is never reset in the course of the Redis server execution, only a server restart will reset it.

Obtaining the current length of the slow log

It is possible to get just the length of the slow log using the command **SLOWLOG LEN**.

Resetting the slow log.

You can reset the slow log using the **SLOWLOG RESET** command. Once deleted the information is lost forever.

SMEMBERS key

Related commands

- SADD
- SCARD
- SDIFF
- SDIFFSTORE
- SINTER
- SINTERSTORE
- SISMEMBER
- **SMEMBERS**
- SMOVE
- SPOP
- SRANDMEMBER
- SREM
- SUNION
- SUNIONSTORE

Available since 1.0.0.

Time complexity: O(N) where N is the set cardinality.

Returns all the members of the set value stored at `key`.

This has the same effect as running SINTER with one argument `key`.

Return value

Multi-bulk reply: all elements of the set.

Examples

redis> SADD myset "Hello"

```
(integer) 1
```

redis> SADD myset "World"

```
(integer) 1
```

redis> SMEMBERS myset

```
1) "World"
```

redis>

This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

SMOVE source destination member

Related commands

- SADD
- SCARD
- SDIFF
- SDIFFSTORE
- SINTER
- SINTERSTORE
- SISMEMBER
- SMEMBERS
- **SMOVE**
- SPOP
- SRANDMEMBER
- SREM
- SUNION
- SUNIONSTORE

Available since 1.0.0.

Time complexity: O(1)

Move `member` from the set at `source` to the set at `destination`. This operation is atomic. In every given moment the element will appear to be a member of `source` **or** `destination` for other clients.

If the source set does not exist or does not contain the specified element, no operation is performed and `0` is returned. Otherwise, the element is removed from the source set and added to the destination set. When the specified element already exists in the destination set, it is only removed from the source set.

An error is returned if `source` or `destination` does not hold a set value.

Return value

Integer reply, specifically:

- `1` if the element is moved.
- `0` if the element is not a member of `source` and no operation was performed.

Examples

redis> SADD myset "one"

```
(integer) 1
```

redis> SADD myset "two"

```
(integer) 1
```

redis> SADD myotherset "three"

```
(integer) 1
```

redis> SMOVE myset myotherset "two"

```
(integer) 1
```

redis> SMEMBERS myset

```
1) "one"
```

redis> SMEMBERS myotherset

```
1) "two"
```

redis>

SORT key [BY pattern] [LIMIT offset count] [GET pattern [GET pattern ...]] [ASC|DESC] [ALPHA] [STORE destination]

Related commands

- DEL
- DUMP
- EXISTS
- EXPIRE
- EXPIREAT
- KEYS
- MIGRATE
- MOVE
- OBJECT
- PERSIST
- PEXPIRE
- PEXPIREAT
- PTTL
- RANDOMKEY
- RENAME
- RENAMENX
- RESTORE
- **SORT**
- TTL
- TYPE

Available since 1.0.0.

Time complexity: O(N+M*log(M)) where N is the number of elements in the list or set to sort, and M the number of returned elements. When the elements are not sorted, complexity is currently O(N) as there is a copy step that will be avoided in next releases.

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
```

Assuming `mylist` is a list of numbers, this command will return the same list with the elements sorted from small to large. 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
```

Redis is UTF-8 aware, assuming you correctly set the !LC_COLLATE environment variable.

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
```

Sorting by external keys

Sometimes you want to sort elements using external keys as weights to compare instead of comparing the actual elements in the list, set or sorted set. Let's say the list mylist contains the elements 1, 2 and 3 representing unique IDs of objects stored in object_1, object_2 and object_3. When these objects have associated weights stored in weight_1, weight_2 and weight_3, SORT can be instructed to use these weights to sort mylist with the following statement:

```
SORT mylist BY weight_*
```

The BY option takes a pattern (equal to weight_* in this example) that is used to generate the keys that are used for sorting. These key names are obtained substituting the first occurrence of * with the actual value of the element in the list (1, 2 and 3 in this example).

Skip sorting the elements

The BY option can also take a non-existent key, which causes SORT to skip the sorting operation. This is useful if you want to retrieve external keys (see the GET option below) without the overhead of sorting.

```
SORT mylist BY nosort
```

Retrieving external keys

Our previous example returns just the sorted IDs. In some cases, it is more useful to get the actual objects instead of their IDs (object_1, object_2 and object_3). Retrieving external keys based on the elements in a list, set or sorted set can be done with the following command:

```
SORT mylist BY weight_* GET object_*
```

The GET option can be used multiple times in order to get more keys for every element of the original list, set or sorted set.

It is also possible to GET the element itself using the special pattern #:

```
SORT mylist BY weight_* GET object_* GET #
```

Storing the result of a SORT operation

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 BY weight_* STORE resultkey
```

An interesting pattern using `SORT ... STORE` consists in associating an EXPIRE timeout to the resulting key so that in applications where the result of a SORT operation can be cached for some time. Other clients will use the cached list instead of calling SORT for every request. When the key will timeout, an updated version of the cache can be created by calling `SORT ... STORE` again.

Note that for correctly implementing this pattern it is important to avoid multiple clients rebuilding the cache at the same time. Some kind of locking is needed here (for instance using SETNX).

Using hashes in `BY` and `GET`

It is possible to use `BY` and `GET` options against hash fields with the following syntax:

```
SORT mylist BY weight_*->fieldname GET object_*->fieldname
```

The string `->` is used to separate the key name from the hash field name. The key is substituted as documented above, and the hash stored at the resulting key is accessed to retrieve the specified hash field.

Return value

Multi-bulk reply: list of sorted elements.

SPOP key

Related commands

- SADD
- SCARD
- SDIFF
- SDIFFSTORE
- SINTER
- SINTERSTORE
- SISMEMBER
- SMEMBERS
- SMOVE
- **SPOP**
- SRANDMEMBER
- SREM
- SUNION
- SUNIONSTORE

Available since 1.0.0.

Time complexity: O(1)

Removes and returns a random element from the set value stored at `key`.

This operation is similar to SRANDMEMBER, that returns a random element from a set but does not remove it.

Return value

Bulk reply: the removed element, or `nil` when `key` does not exist.

Examples

redis> SADD myset "one"

```
(integer) 1
```

redis> SADD myset "two"

```
(integer) 1
```

redis> SADD myset "three"

```
(integer) 1
```

redis> SPOP myset

```
"two"
```

redis> SMEMBERS myset

```
1) "one"
```

redis>

SRANDMEMBER key [count]

Related commands

- SADD
- SCARD
- SDIFF
- SDIFFSTORE
- SINTER
- SINTERSTORE
- SISMEMBER
- SMEMBERS
- SMOVE
- SPOP
- **SRANDMEMBER**
- SREM
- SUNION
- SUNIONSTORE

Available since 1.0.0.

Time complexity: Without the count argument O(1), otherwise O(N) where N is the absolute value of the passed count.

When called with just the `key` argument, return a random element from the set value stored at `key`.

Starting from Redis version 2.6, when called with the additional `count` argument, return an array of `count` **distinct elements** if `count` is positive. 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 numer of returned elements is the absolute value of the specified `count`.

When called with just the key argument, the operation is similar to SPOP, however while SPOP also removes the randomly selected element from the set, SRANDMEMBER will just return a random element without altering the original set in any way.

Return value

Bulk reply: without the additional `count` argument the command returns a Bulk Reply with the randomly selected element, or `nil` when `key` does not exist. Multi-bulk reply: when the additional `count` argument is passed the command returns an array of elements, or an empty array when `key` does not exist.

Examples

redis> SADD myset one two three

```
(integer) 3
```

redis> SRANDMEMBER myset

```
"one"
```

redis> SRANDMEMBER myset 2

```
1) "one"
```

redis> SRANDMEMBER myset -5

```
1) "three"
```

redis>

Specification of the behavior when count is passed

When a count argument is passed and is positive, the elements are returned as if every selected element is removed from the set (like the extraction of numbers in the game of Bingo). However elements are **not removed** from the Set. So basically:

- No repeated elements are returned.
- If count is bigger than the number of elements inside the Set, the command will only return the whole set without additional elements.

When instead the count is negative, the behavior changes and the extraction happens as if you put the extracted element inside the bag again after every extraction, so repeated elements are possible, and the number of elements requested is always returned as we can repeat the same elements again and again, with the exception of an empty Set (non existing key) that will always produce an empty array as a result.

Distribution of returned elements

The distribution of the returned elements is far from perfect when the number of elements in the set is small, this is due to the fact that we used an approximated random element function that does not really guarantees good distribution.

The algorithm used, that is implemented inside dict.c, samples the hash table buckets to find a non-empty one. Once a non empty bucket is found, since we use chaining in our hash table implementation, the number of elements inside the bucked is checked and a random element is selected.

This means that if you have two non-empty buckets in the entire hash table, and one has three elements while one has just one, the element that is alone in its bucket will be returned with much higher probability.

SREM key member [member ...]

Related commands

- SADD
- SCARD
- SDIFF
- SDIFFSTORE
- SINTER
- SINTERSTORE
- SISMEMBER
- SMEMBERS
- SMOVE
- SPOP
- SRANDMEMBER
- **SREM**
- SUNION
- SUNIONSTORE

Available since 1.0.0.

Time complexity: O(N) where N is the number of members to be removed.

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`.

An error is returned when the value stored at `key` is not a set.

Return value

Integer reply: the number of members that were removed from the set, not including non existing members.

History

- `>= 2.4`: Accepts multiple `member` arguments. Redis versions older than 2.4 can only remove a set member per call.

Examples

redis> SADD myset "one"

```
(integer) 1
```

redis> SADD myset "two"

```
(integer) 1
```

redis> SADD myset "three"

```
(integer) 1
```

redis> SREM myset "one"

```
(integer) 1
```

redis> SREM myset "four"

```
(integer) 0
```

redis> SMEMBERS myset

```
1) "two"
```

redis>

STRLEN key

Related commands

- APPEND
- BITCOUNT
- BITOP
- DECR
- DECRBY
- GET
- GETBIT
- GETRANGE
- GETSET
- INCR
- INCRBY
- INCRBYFLOAT
- MGET
- MSET
- MSETNX
- PSETEX
- SET
- SETBIT
- SETEX
- SETNX
- SETRANGE
- **STRLEN**

Available since 2.2.0.

Time complexity: O(1)

Returns the length of the string value stored at `key`. An error is returned when `key` holds a non-string value.

Return value

Integer reply: the length of the string at `key`, or `0` when `key` does not exist.

Examples

redis> SET mykey "Hello world"

```
OK
```

redis> STRLEN mykey

```
(integer) 11
```

redis> STRLEN nonexisting

```
(integer) 0
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

SUBSCRIBE channel [channel ...]

Related topics

- Pub/Sub

Related commands

- PSUBSCRIBE
- PUBLISH
- PUNSUBSCRIBE
- **SUBSCRIBE**
- UNSUBSCRIBE

Available since 2.0.0.

Time complexity: O(N) where N is the number of channels to subscribe to.

Subscribes the client to the specified channels.

Once the client enters the subscribed state it is not supposed to issue any other commands, except for additional SUBSCRIBE, PSUBSCRIBE, UNSUBSCRIBE and PUNSUBSCRIBE commands.

SUNION key [key ...]

Related commands

- SADD
- SCARD
- SDIFF
- SDIFFSTORE
- SINTER
- SINTERSTORE
- SISMEMBER
- SMEMBERS
- SMOVE
- SPOP
- SRANDMEMBER
- SREM
- **SUNION**
- SUNIONSTORE

Available since 1.0.0.

Time complexity: O(N) where N is the total number of elements in all given sets.

Returns the members of the set resulting from the union of all the given sets.

For example:

```
key1 = {a,b,c,d}
```

Keys that do not exist are considered to be empty sets.

Return value

Multi-bulk reply: list with members of the resulting set.

Examples

redis> SADD key1 "a"

```
(integer) 1
```

redis> SADD key1 "b"

```
(integer) 1
```

redis> SADD key1 "c"

```
(integer) 1
```

```
redis> SADD key2 "c"
(integer) 1
redis> SADD key2 "d"
(integer) 1
redis> SADD key2 "e"
(integer) 1
redis> SUNION key1 key2
1) "b"
redis>
```

SUNIONSTORE destination key [key ...]

Related commands

- SADD
- SCARD
- SDIFF
- SDIFFSTORE
- SINTER
- SINTERSTORE
- SISMEMBER
- SMEMBERS
- SMOVE
- SPOP
- SRANDMEMBER
- SREM
- SUNION
- **SUNIONSTORE**

Available since 1.0.0.

Time complexity: O(N) where N is the total number of elements in all given sets.

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.

Return value

Integer reply: the number of elements in the resulting set.

SYNC

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- INFO
- LASTSAVE
- MONITOR
- SAVE
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- **SYNC**
- TIME

Available since 1.0.0.

Examples

Return value

TIME

Related commands

- BGREWRITEAOF
- BGSAVE
- CLIENT KILL
- CLIENT LIST
- CONFIG GET
- CONFIG RESETSTAT
- CONFIG SET
- DBSIZE
- DEBUG OBJECT
- DEBUG SEGFAULT
- FLUSHALL
- FLUSHDB
- INFO
- LASTSAVE
- MONITOR
- SAVE
- SHUTDOWN
- SLAVEOF
- SLOWLOG
- SYNC
- **TIME**

Available since 2.6.0.

Time complexity: O(1)

The TIME command returns the current server time as a two items lists: a Unix timestamp and the amount of microseconds already elapsed in the current second. Basically the interface is very similar to the one of the `gettimeofday` system call.

Return value

Multi-bulk reply, specifically:

A multi bulk reply containing two elements:

- unix time in seconds.
- microseconds.

Examples

redis> TIME

```
1) "1350190384"
```

redis> TIME

```
1) "1350190384"
```

redis>

TTL key

Related commands

- DEL
- DUMP
- EXISTS
- EXPIRE
- EXPIREAT
- KEYS
- MIGRATE
- MOVE
- OBJECT
- PERSIST
- PEXPIRE
- PEXPIREAT
- PTTL
- RANDOMKEY
- RENAME
- RENAMENX
- RESTORE
- SORT
- **TTL**
- TYPE

Available since 1.0.0.

Time complexity: O(1)

Returns the remaining time to live of a key that has a timeout. This introspection capability allows a Redis client to check how many seconds a given key will continue to be part of the dataset.

Return value

Integer reply: TTL in seconds or `-1` when `key` does not exist or does not have a timeout.

Examples

redis> SET mykey "Hello"

```
OK
```

redis> EXPIRE mykey 10

```
(integer) 1
```

redis> TTL mykey

```
(integer) 10
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

TYPE key

Related commands

- DEL
- DUMP
- EXISTS
- EXPIRE
- EXPIREAT
- KEYS
- MIGRATE
- MOVE
- OBJECT
- PERSIST
- PEXPIRE
- PEXPIREAT
- PTTL
- RANDOMKEY
- RENAME
- RENAMENX
- RESTORE
- SORT
- TTL
- **TYPE**

Available since 1.0.0.

Time complexity: 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` and `hash`.

Return value

Status code reply: type of `key`, or `none` when `key` does not exist.

Examples

redis> SET key1 "value"

```
OK
```

redis> LPUSH key2 "value"

```
(integer) 1
```

redis> SADD key3 "value"

```
(integer) 1
```

redis> TYPE key1

```
string
```

redis> TYPE key2

```
list
```

redis> TYPE key3

```
set
```

redis>

UNSUBSCRIBE [channel [channel ...]]

Related topics

- Pub/Sub

Related commands

- PSUBSCRIBE
- PUBLISH
- PUNSUBSCRIBE
- SUBSCRIBE
- **UNSUBSCRIBE**

Available since 2.0.0.

Time complexity: O(N) where N is the number of clients already subscribed to a channel.

Unsubscribes the client from the given channels, or from all of them if none is given.

When no channels are specified, the client is unsubscribed from all the previously subscribed channels. In this case, a message for every unsubscribed channel will be sent to the client.

UNWATCH

Related topics

- Transactions

Related commands

- DISCARD
- EXEC
- MULTI
- **UNWATCH**
- WATCH

Available since 2.2.0.

Time complexity: O(1)

Flushes all the previously watched keys for a transaction.

If you call EXEC or DISCARD, there's no need to manually call UNWATCH.

Return value

Status code reply: always `OK`.

WATCH key [key ...]

Related topics

- Transactions

Related commands

- DISCARD
- EXEC
- MULTI
- UNWATCH
- **WATCH**

Available since 2.2.0.

Time complexity: O(1) for every key.

Marks the given keys to be watched for conditional execution of a transaction.

Return value

Status code reply: always `OK`.

ZADD key score member [score] [member]

Related commands

- **ZADD**
- ZCARD
- ZCOUNT
- ZINCRBY
- ZINTERSTORE
- ZRANGE
- ZRANGEBYSCORE
- ZRANK
- ZREM
- ZREMRANGEBYRANK
- ZREMRANGEBYSCORE
- ZREVRANGE
- ZREVRANGEBYSCORE
- ZREVRANK
- ZSCORE
- ZUNIONSTORE

Available since 1.2.0.

Time complexity: O(log(N)) where N is the number of elements in the sorted set.

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. If `key` does not exist, a new sorted set with the specified members as sole members is created, like if the sorted set was empty. If the key exists but does not hold a sorted set, an error is returned.

The score values should be the string representation of a numeric value, and accepts double precision floating point numbers.

For an introduction to sorted sets, see the data types page on sorted sets.

Return value

Integer reply, specifically:

- The number of elements added to the sorted sets, not including elements already existing for which the score was updated.

History

- `>= 2.4`: Accepts multiple elements. In Redis versions older than 2.4 it was possible to add or update a single member per call.

Examples

redis> ZADD myzset 1 "one"

```
(integer) 1
```

redis> ZADD myzset 1 "uno"

```
(integer) 1
```

redis> ZADD myzset 2 "two"

```
(integer) 1
```

redis> ZADD myzset 3 "two"

```
(integer) 0
```

redis> ZRANGE myzset 0 -1 WITHSCORES

```
1) "one"
```

redis>

ZCARD key

Related commands

- ZADD
- **ZCARD**
- ZCOUNT
- ZINCRBY
- ZINTERSTORE
- ZRANGE
- ZRANGEBYSCORE
- ZRANK
- ZREM
- ZREMRANGEBYRANK
- ZREMRANGEBYSCORE
- ZREVRANGE
- ZREVRANGEBYSCORE
- ZREVRANK
- ZSCORE
- ZUNIONSTORE

Available since 1.2.0.

Time complexity: O(1)

Returns the sorted set cardinality (number of elements) of the sorted set stored at `key`.

Return value

Integer reply: the cardinality (number of elements) of the sorted set, or `0` if `key` does not exist.

Examples

redis> ZADD myzset 1 "one"

```
(integer) 1
```

redis> ZADD myzset 2 "two"

```
(integer) 1
```

redis> ZCARD myzset

```
(integer) 2
```

redis>

This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

ZCOUNT key min max

Related commands

- ZADD
- ZCARD
- **ZCOUNT**
- ZINCRBY
- ZINTERSTORE
- ZRANGE
- ZRANGEBYSCORE
- ZRANK
- ZREM
- ZREMRANGEBYRANK
- ZREMRANGEBYSCORE
- ZREVRANGE
- ZREVRANGEBYSCORE
- ZREVRANK
- ZSCORE
- ZUNIONSTORE

Available since 2.0.0.

Time complexity: O(log(N)+M) with N being the number of elements in the sorted set and M being the number of elements between min and max.

Returns the number of elements in the sorted set at `key` with a score between `min` and `max`.

The `min` and `max` arguments have the same semantic as described for ZRANGEBYSCORE.

Return value

Integer reply: the number of elements in the specified score range.

Examples

redis> ZADD myzset 1 "one"

```
(integer) 1
```

redis> ZADD myzset 2 "two"

```
(integer) 1
```

redis> ZADD myzset 3 "three"

```
(integer) 1
```

redis> ZCOUNT myzset -inf +inf

```
(integer) 3
```

redis> ZCOUNT myzset (1 3

```
(integer) 2
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

ZINCRBY key increment member

Related commands

- ZADD
- ZCARD
- ZCOUNT
- **ZINCRBY**
- ZINTERSTORE
- ZRANGE
- ZRANGEBYSCORE
- ZRANK
- ZREM
- ZREMRANGEBYRANK
- ZREMRANGEBYSCORE
- ZREVRANGE
- ZREVRANGEBYSCORE
- ZREVRANK
- ZSCORE
- ZUNIONSTORE

Available since 1.2.0.

Time complexity: O(log(N)) where N is the number of elements in the sorted set.

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.

An error is returned when `key` exists but does not hold a sorted set.

The `score` value should be the string representation of a numeric value, and accepts double precision floating point numbers. It is possible to provide a negative value to decrement the score.

Return value

Bulk reply: the new score of `member` (a double precision floating point number), represented as string.

Examples

redis> ZADD myzset 1 "one"

```
(integer) 1
```

redis> ZADD myzset 2 "two"

```
(integer) 1
```

redis> ZINCRBY myzset 2 "one"

```
"3"
```

redis> ZRANGE myzset 0 -1 WITHSCORES

```
1) "two"
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

ZINTERSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]

Related commands

- ZADD
- ZCARD
- ZCOUNT
- ZINCRBY
- **ZINTERSTORE**
- ZRANGE
- ZRANGEBYSCORE
- ZRANK
- ZREM
- ZREMRANGEBYRANK
- ZREMRANGEBYSCORE
- ZREVRANGE
- ZREVRANGEBYSCORE
- ZREVRANK
- ZSCORE
- ZUNIONSTORE

Available since 2.0.0.

Time complexity: O(N*K)+O(M*log(M)) worst case with N being the smallest input sorted set, K being the number of input sorted sets and M being the number of elements in the resulting sorted set.

Computes the intersection of `numkeys` sorted sets given by the specified keys, and stores the result in `destination`. It is mandatory to provide the number of input keys (`numkeys`) before passing the input keys and the other (optional) arguments.

By default, the resulting score of an element is the sum of its scores in the sorted sets where it exists. Because intersection requires an element to be a member of every given sorted set, this results in the score of every element in the resulting sorted set to be equal to the number of input sorted sets.

For a description of the `WEIGHTS` and `AGGREGATE` options, see ZUNIONSTORE.

If `destination` already exists, it is overwritten.

Return value

Integer reply: the number of elements in the resulting sorted set at `destination`.

Examples

redis> ZADD zset1 1 "one"

```
(integer) 1
```

redis> ZADD zset1 2 "two"

```
(integer) 1
```

redis> ZADD zset2 1 "one"

```
(integer) 1
```

redis> ZADD zset2 2 "two"

```
(integer) 1
```

redis> ZADD zset2 3 "three"

```
(integer) 1
```

redis> ZINTERSTORE out 2 zset1 zset2 WEIGHTS 2 3

```
(integer) 2
```

redis> ZRANGE out 0 -1 WITHSCORES

```
1) "one"
```

redis>

ZRANGE key start stop [WITHSCORES]

Related commands

- ZADD
- ZCARD
- ZCOUNT
- ZINCRBY
- ZINTERSTORE
- **ZRANGE**
- ZRANGEBYSCORE
- ZRANK
- ZREM
- ZREMRANGEBYRANK
- ZREMRANGEBYSCORE
- ZREVRANGE
- ZREVRANGEBYSCORE
- ZREVRANK
- ZSCORE
- ZUNIONSTORE

Available since 1.2.0.

Time complexity: O(log(N)+M) with N being the number of elements in the sorted set and M the number of elements returned.

Returns the specified range of elements in the sorted set stored at `key`. The elements are considered to be ordered from the lowest to the highest score. Lexicographical order is used for elements with equal score.

See ZREVRANGE when you need the elements ordered from highest to lowest score (and descending lexicographical order for elements with equal score).

Both `start` and `stop` are zero-based indexes, where `0` is the first element, `1` is the next element and so on. They can also be negative numbers indicating offsets from the end of the sorted set, with `-1` being the last element of the sorted set, `-2` the penultimate element and so on.

Out of range indexes will not produce an error. If `start` is larger than the largest index in the sorted set, or `start > stop`, an empty list is returned. If `stop` is larger than the end of the sorted set Redis will treat it like it is the last element of the sorted set.

It is possible to pass the `WITHSCORES` option in order to return the scores of the elements together with the elements. The returned list will contain `value1,score1,...,valueN,scoreN` instead of `value1,...,valueN`. Client libraries are free to return a more appropriate data type (suggestion: an array with (value, score) arrays/tuples).

Return value

Multi-bulk reply: list of elements in the specified range (optionally with their scores).

Examples

redis> ZADD myzset 1 "one"

```
(integer) 1
```

redis> ZADD myzset 2 "two"

```
(integer) 1
```

redis> ZADD myzset 3 "three"

```
(integer) 1
```

redis> ZRANGE myzset 0 -1

```
1) "one"
```

redis> ZRANGE myzset 2 3

```
1) "three"
```

redis> ZRANGE myzset -2 -1

```
1) "two"
```

redis>

ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT offset count]

Related commands

- ZADD
- ZCARD
- ZCOUNT
- ZINCRBY
- ZINTERSTORE
- ZRANGE
- **ZRANGEBYSCORE**
- ZRANK
- ZREM
- ZREMRANGEBYRANK
- ZREMRANGEBYSCORE
- ZREVRANGE
- ZREVRANGEBYSCORE
- ZREVRANK
- ZSCORE
- ZUNIONSTORE

Available since 1.0.5.

Time complexity: O(log(N)+M) with N being the number of elements in the sorted set and M the number of elements being returned. If M is constant (e.g. always asking for the first 10 elements with LIMIT), you can consider it O(log(N)).

Returns all the elements in the sorted set at `key` with a score between `min` and `max` (including elements with score equal to `min` or `max`). The elements are considered to be ordered from low to high scores.

The elements having the same score are returned in lexicographical order (this follows from a property of the sorted set implementation in Redis and does not involve further computation).

The optional `LIMIT` argument can be used to only get a range of the matching elements (similar to *SELECT LIMIT offset, count* in SQL). Keep in mind that if `offset` is large, the sorted set needs to be traversed for `offset` elements before getting to the elements to return, which can add up to O(N) time complexity.

The optional `WITHSCORES` argument makes the command return both the element and its score, instead of the element alone. This option is available since Redis 2.0.

Exclusive intervals and infinity

`min` and `max` can be `-inf` and `+inf`, so that you are not required to know the highest or lowest score in the sorted set to get all elements from or up to a certain score.

By default, the interval specified by `min` and `max` is closed (inclusive). It is possible to specify an open interval (exclusive) by prefixing the score with the character `(`. For example:

```
ZRANGEBYSCORE zset (1 5
```

Will return all elements with `1 < score <= 5` while:

```
ZRANGEBYSCORE zset (5 (10
```

Will return all the elements with `5 < score < 10` (5 and 10 excluded).

Return value

Multi-bulk reply: list of elements in the specified score range (optionally with their scores).

Examples

redis> ZADD myzset 1 "one"

```
(integer) 1
```

redis> ZADD myzset 2 "two"

```
(integer) 1
```

redis> ZADD myzset 3 "three"

```
(integer) 1
```

redis> ZRANGEBYSCORE myzset -inf +inf

```
1) "one"
```

redis> ZRANGEBYSCORE myzset 1 2

```
1) "one"
```

redis> ZRANGEBYSCORE myzset (1 2

```
1) "two"
```

redis> ZRANGEBYSCORE myzset (1 (2

```
(empty list or set)
```

redis>

ZRANK key member

Related commands

- ZADD
- ZCARD
- ZCOUNT
- ZINCRBY
- ZINTERSTORE
- ZRANGE
- ZRANGEBYSCORE
- **ZRANK**
- ZREM
- ZREMRANGEBYRANK
- ZREMRANGEBYSCORE
- ZREVRANGE
- ZREVRANGEBYSCORE
- ZREVRANK
- ZSCORE
- ZUNIONSTORE

Available since 2.0.0.

Time complexity: 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`.

Use ZREVRANK to get the rank of an element with the scores ordered from high to low.

Return value

- If `member` exists in the sorted set, Integer reply: the rank of `member`.
- If `member` does not exist in the sorted set or `key` does not exist, Bulk reply: `nil`.

Examples

redis> ZADD myzset 1 "one"

```
(integer) 1
```

redis> ZADD myzset 2 "two"

```
(integer) 1
```

redis> ZADD myzset 3 "three"

```
(integer) 1
```

redis> ZRANK myzset "three"

```
(integer) 2
```

redis> ZRANK myzset "four"

```
(nil)
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

ZREM key member [member ...]

Related commands

- ZADD
- ZCARD
- ZCOUNT
- ZINCRBY
- ZINTERSTORE
- ZRANGE
- ZRANGEBYSCORE
- ZRANK
- **ZREM**
- ZREMRANGEBYRANK
- ZREMRANGEBYSCORE
- ZREVRANGE
- ZREVRANGEBYSCORE
- ZREVRANK
- ZSCORE
- ZUNIONSTORE

Available since 1.2.0.

Time complexity: O(M*log(N)) with N being the number of elements in the sorted set and M the number of elements to be removed.

Removes the specified members from the sorted set stored at `key`. Non existing members are ignored.

An error is returned when `key` exists and does not hold a sorted set.

Return value

Integer reply, specifically:

- The number of members removed from the sorted set, not including non existing members.

History

- `>= 2.4`: Accepts multiple elements. In Redis versions older than 2.4 it was possible to remove a single member per call.

Examples

redis> ZADD myzset 1 "one"

```
(integer) 1
```

```
redis> ZADD myzset 2 "two"
(integer) 1
redis> ZADD myzset 3 "three"
(integer) 1
redis> ZREM myzset "two"
(integer) 1
redis> ZRANGE myzset 0 -1 WITHSCORES
1) "one"
redis>
```

ZREMRANGEBYRANK key start stop

Related commands

- ZADD
- ZCARD
- ZCOUNT
- ZINCRBY
- ZINTERSTORE
- ZRANGE
- ZRANGEBYSCORE
- ZRANK
- ZREM
- **ZREMRANGEBYRANK**
- ZREMRANGEBYSCORE
- ZREVRANGE
- ZREVRANGEBYSCORE
- ZREVRANK
- ZSCORE
- ZUNIONSTORE

Available since 2.0.0.

Time complexity: O(log(N)+M) with N being the number of elements in the sorted set and M the number of elements removed by the operation.

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. These indexes can be negative numbers, where they indicate offsets starting at the element with the highest score. For example: `-1` is the element with the highest score, `-2` the element with the second highest score and so forth.

Return value

Integer reply: the number of elements removed.

Examples

redis> ZADD myzset 1 "one"

```
(integer) 1
```

redis> ZADD myzset 2 "two"

```
(integer) 1
```

redis> ZADD myzset 3 "three"

```
(integer) 1
```

redis> ZREMRANGEBYRANK myzset 0 1

```
(integer) 2
```

redis> ZRANGE myzset 0 -1 WITHSCORES

```
1) "three"
```

redis>
Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by vmware

ZREMRANGEBYSCORE key min max

Related commands

- ZADD
- ZCARD
- ZCOUNT
- ZINCRBY
- ZINTERSTORE
- ZRANGE
- ZRANGEBYSCORE
- ZRANK
- ZREM
- ZREMRANGEBYRANK
- **ZREMRANGEBYSCORE**
- ZREVRANGE
- ZREVRANGEBYSCORE
- ZREVRANK
- ZSCORE
- ZUNIONSTORE

Available since 1.2.0.

Time complexity: O(log(N)+M) with N being the number of elements in the sorted set and M the number of elements removed by the operation.

Removes all elements in the sorted set stored at `key` with a score between `min` and `max` (inclusive).

Since version 2.1.6, `min` and `max` can be exclusive, following the syntax of ZRANGEBYSCORE.

Return value

Integer reply: the number of elements removed.

Examples

redis> ZADD myzset 1 "one"

```
(integer) 1
```

redis> ZADD myzset 2 "two"

```
(integer) 1
```

redis> ZADD myzset 3 "three"

```
(integer) 1
```

redis> ZREMRANGEBYSCORE myzset -inf (2

```
(integer) 1
```

redis> ZRANGE myzset 0 -1 WITHSCORES

```
1) "two"
```

redis>

ZREVRANGE key start stop [WITHSCORES]

Related commands

- ZADD
- ZCARD
- ZCOUNT
- ZINCRBY
- ZINTERSTORE
- ZRANGE
- ZRANGEBYSCORE
- ZRANK
- ZREM
- ZREMRANGEBYRANK
- ZREMRANGEBYSCORE
- **ZREVRANGE**
- ZREVRANGEBYSCORE
- ZREVRANK
- ZSCORE
- ZUNIONSTORE

Available since 1.2.0.

Time complexity: O(log(N)+M) with N being the number of elements in the sorted set and M the number of elements returned.

Returns the specified range of elements in the sorted set stored at `key`. The elements are considered to be ordered from the highest to the lowest score. Descending lexicographical order is used for elements with equal score.

Apart from the reversed ordering, ZREVRANGE is similar to ZRANGE.

Return value

Multi-bulk reply: list of elements in the specified range (optionally with their scores).

Examples

redis> ZADD myzset 1 "one"

```
(integer) 1
```

redis> ZADD myzset 2 "two"

```
(integer) 1
```

redis> ZADD myzset 3 "three"

```
(integer) 1
```

redis> ZREVRANGE myzset 0 -1

```
1) "three"
```

redis> ZREVRANGE myzset 2 3

```
1) "one"
```

redis> ZREVRANGE myzset -2 -1

```
1) "two"
```

redis>

ZREVRANGEBYSCORE key max min [WITHSCORES] [LIMIT offset count]

Related commands

- ZADD
- ZCARD
- ZCOUNT
- ZINCRBY
- ZINTERSTORE
- ZRANGE
- ZRANGEBYSCORE
- ZRANK
- ZREM
- ZREMRANGEBYRANK
- ZREMRANGEBYSCORE
- ZREVRANGE
- **ZREVRANGEBYSCORE**
- ZREVRANK
- ZSCORE
- ZUNIONSTORE

Available since 2.2.0.

Time complexity: O(log(N)+M) with N being the number of elements in the sorted set and M the number of elements being returned. If M is constant (e.g. always asking for the first 10 elements with LIMIT), you can consider it O(log(N)).

Returns all the elements in the sorted set at `key` with a score between `max` and `min` (including elements with score equal to `max` or `min`). In contrary to the default ordering of sorted sets, for this command the elements are considered to be ordered from high to low scores.

The elements having the same score are returned in reverse lexicographical order.

Apart from the reversed ordering, ZREVRANGEBYSCORE is similar to ZRANGEBYSCORE.

Return value

Multi-bulk reply: list of elements in the specified score range (optionally with their scores).

Examples

redis> ZADD myzset 1 "one"

```
(integer) 1
```

```
redis> ZADD myzset 2 "two"
(integer) 1
redis> ZADD myzset 3 "three"
(integer) 1
redis> ZREVRANGEBYSCORE myzset +inf -inf
1) "three"
redis> ZREVRANGEBYSCORE myzset 2 1
1) "two"
redis> ZREVRANGEBYSCORE myzset 2 (1
1) "two"
redis> ZREVRANGEBYSCORE myzset (2 (1
(empty list or set)
redis>
```

Comments powered by Disqus
This website is open source software developed by Citrusbyte.
The Redis logo was designed by Carlos Prioglio. See more credits.

Sponsored by 

ZREVRANK key member

Related commands

- ZADD
- ZCARD
- ZCOUNT
- ZINCRBY
- ZINTERSTORE
- ZRANGE
- ZRANGEBYSCORE
- ZRANK
- ZREM
- ZREMRANGEBYRANK
- ZREMRANGEBYSCORE
- ZREVRANGE
- ZREVRANGEBYSCORE
- **ZREVRANK**
- ZSCORE
- ZUNIONSTORE

Available since 2.0.0.

Time complexity: O(log(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`.

Use ZRANK to get the rank of an element with the scores ordered from low to high.

Return value

- If `member` exists in the sorted set, Integer reply: the rank of `member`.
- If `member` does not exist in the sorted set or `key` does not exist, Bulk reply: `nil`.

Examples

redis> ZADD myzset 1 "one"

```
(integer) 1
```

redis> ZADD myzset 2 "two"

```
(integer) 1
```

redis> ZADD myzset 3 "three"

```
(integer) 1
```

redis> ZREVRANK myzset "one"

```
(integer) 2
```

redis> ZREVRANK myzset "four"

```
(nil)
```

redis>

ZSCORE key member

Related commands

- ZADD
- ZCARD
- ZCOUNT
- ZINCRBY
- ZINTERSTORE
- ZRANGE
- ZRANGEBYSCORE
- ZRANK
- ZREM
- ZREMRANGEBYRANK
- ZREMRANGEBYSCORE
- ZREVRANGE
- ZREVRANGEBYSCORE
- ZREVRANK
- **ZSCORE**
- ZUNIONSTORE

Available since 1.2.0.

Time complexity: O(1)

Returns the score of `member` in the sorted set at `key`.

If `member` does not exist in the sorted set, or `key` does not exist, `nil` is returned.

Return value

Bulk reply: the score of `member` (a double precision floating point number), represented as string.

Examples

redis> ZADD myzset 1 "one"

```
(integer) 1
```

redis> ZSCORE myzset "one"

```
"1"
```

redis>

ZUNIONSTORE destination numkeys key [key ...] [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]

Related commands

- ZADD
- ZCARD
- ZCOUNT
- ZINCRBY
- ZINTERSTORE
- ZRANGE
- ZRANGEBYSCORE
- ZRANK
- ZREM
- ZREMRANGEBYRANK
- ZREMRANGEBYSCORE
- ZREVRANGE
- ZREVRANGEBYSCORE
- ZREVRANK
- ZSCORE
- **ZUNIONSTORE**

Available since 2.0.0.

Time complexity: O(N)+O(M log(M)) with N being the sum of the sizes of the input sorted sets, and M being the number of elements in the resulting sorted set.

Computes the union of `numkeys` sorted sets given by the specified keys, and stores the result in `destination`. It is mandatory to provide the number of input keys (`numkeys`) before passing the input keys and the other (optional) arguments.

By default, the resulting score of an element is the sum of its scores in the sorted sets where it exists.

Using the `WEIGHTS` option, it is possible to specify a multiplication factor for each input sorted set. This means that the score of every element in every input sorted set is multiplied by this factor before being passed to the aggregation function. When `WEIGHTS` is not given, the multiplication factors default to `1`.

With the `AGGREGATE` option, it is possible to specify how the results of the union are aggregated. This option defaults to `SUM`, where the score of an element is summed across the inputs where it exists. When this option is set to either `MIN` or `MAX`, the resulting set will contain the minimum or maximum score of an element across the inputs where it exists.

If `destination` already exists, it is overwritten.

Return value

Integer reply: the number of elements in the resulting sorted set at `destination`.

Examples

redis> ZADD zset1 1 "one"

```
(integer) 1
```

redis> ZADD zset1 2 "two"

```
(integer) 1
```

redis> ZADD zset2 1 "one"

```
(integer) 1
```

redis> ZADD zset2 2 "two"

```
(integer) 1
```

redis> ZADD zset2 3 "three"

```
(integer) 1
```

redis> ZUNIONSTORE out 2 zset1 zset2 WEIGHTS 2 3

```
(integer) 3
```

redis> ZRANGE out 0 -1 WITHSCORES

```
1) "one"
```

redis>