Redis lab

Download Redis: <https://github.com/rgl/redis/downloads>

Now run redis-server.exe from the Redis home directory

Next run the command- line tool, which should connect to the default port 6379 automatically.

redis-cli.exe

After you connect, let’s try to ping the server.

redis 127.0.0.1:6379> PING

PONG

1. Set some values

SET 7wks <http://www.sevenweeks.org/>

GET 7wks

MSET gog <http://www.google.com> yah <http://www.yahoo.com>

MGET gog yah

SET count 2

INCR count

GET count

1. Transactions are allowed in Redis

MULTI

SET prag <http://pragprog.com>

INCR count

EXEC

1. Hash (tables of name-value pairs)

MSET user:eric:name "Eric Redmond" user:eric:password s3cret

MGET user:eric:name user:eric:password

Instead of separate keys, we can create a hash that contains its own key- value pairs.

HMSET user:eric name "Eric Redmond" password s3cret

We need only keep track of the single Redis key to retrieve all values of the hash

HVALS user:eric

Or we can retrieve all hash keys.

HKEYS user:eric  
Or we can get a single value, by passing in the Redis key, followed by the hash key.

Here we get just the password.

HGET user:eric password

1. Lists

To create a list of short-coded websites we’d like to visit, we set the key to USERNAME:wishlist and push any number of values to the right (end) of the list.

RPUSH eric:wishlist 7wks

RPUSH eric:wishlist gog

RPUSH eric:wishlist prag

All list operations in Redis use a zero- based index.

A negative position means the number of steps from the end (-1 is the last element)

LRANGE eric:wishlist 0 -1

LREM removes from the given key some matching values

It also requires a number to know how many matches to remove.

Setting the count to 0 as we do here just removes them all:

LREM eric:wishlist 0 gog

To remove and retrieve each value in the order we added them (like a queue), we can pop them off from the left (head) of the list.

LPOP eric:wishlist  
To act as a stack, after you RPUSH the values, you would RPOP from the end of the list.

Redis provides a single command for popping values from the tail of one list and pushing to the head of another.

It’s called RPOPLPUSH (right pop, left push).

RPOPLPUSH eric:wishlist eric:visited

1. Blocking Lists

The command to block until a value exists to pop is BRPOP.

It requires the key to pop a value from and a timeout in seconds, which we’ll set to five minutes.

First open another terminal and start another redis-cli client.

This will be our digester

BRPOP comments 300

Then switch back to the first console and push a message to comments.

LPUSH comments "Prag is great! I buy all my books there."

If you switch back to the digester console, two lines will be returned: the key and the popped value.

There’s also a blocking version of left pop (BLPOP) and right pop, left push (BRPOPLPUSH).

1. Sets

If we wanted to categorize sets of URLs with a common key, we can add multiple values with SADD.

SADD news nytimes.com pragprog.com

SMEMBERS news

SADD tech pragprog.com apple.com

To find the intersection of websites that both provide news and are technology focused, we use the SINTER command.

SINTER news tech

To find all news sites that are not tech sites, use SDIFF:

SDIFF news tech

We can also build a union of websites that are either news or tech.

Since it’s a set, any duplicates are dropped.

SUNION news tech

That set of values can also be stored directly into a new set (SUNIONSTORE destination key [key ...]).

SUNIONSTORE websites news tech

This also provides a useful trick for cloning a single key’s values to another key, such as SUNIONSTORE news\_copy news.

Similar commands exist for storing intersections (SINTERSTORE) and diffs (SDIFFSTORE).

Just like RPOPLPUSH moved values from one list to another, SMOVE does the same for sets; it’s just easier to remember.

And like LLEN finds the length of a list, SCARD (set cardinality) counts the set; it’s just harder to remember.

Since sets are not ordered, there are no left, right, or other positional commands.

Popping a random value from a set just requires SPOP key, and removing values is SREM key value [value ...].

Unlike lists, there are no blocking commands for sets.

1. **Sorted Sets**

They are ordered like lists and are unique like sets.

They have field-value pairs like hashes, but rather than string fields, they are

instead numeric scores that denote the order of the values.

zadd hackers 1940 "Alan Kay"

zadd hackers 1953 "Richard Stallman"

zadd hackers 1965 "Yukihiro Matsumoto"

zadd hackers 1916 "Claude Shannon"

zadd hackers 1969 "Linus Torvalds"

zadd hackers 1912 "Alan Turing"

For sorted sets it’s a joke to return these hackers sorted by their birth year because actually **they are already sorted**.

zrange hackers 0 -1

You can think of sorted sets as like a random access priority queue.

This power has a trade- off, however.

Internally, sorted sets keep values in order, so inserts can take log(N) time to insert (where N is the size of the set), rather than the constant time complexity of hashes or lists.

Next we want to keep track of the popularity of specific shortcodes.

Every time someone visits a URL, the score gets increased.

Like a hash, adding a value to a sorted set requires two values after the Redis key name: the score and the member.

ZADD visits 500 7wks 9 gog 9999 prag

To increment a score, we can either re-add it with the new score, which just updates the score but does not add a new value, or increment by some number, which will return the new value.

ZINCRBY visits 1 prag

To get values from our visits set, we can issue a range command, ZRANGE, which returns by position, just like the list datatype’s LRANGE command.

Except in the case of a sorted set, the position is ordered by score from lowest to highest.

So, to get the top two scoring visited sites (zero-based), use this:

ZRANGE visits 0 1

To get the scores of each element as well, append WITHSCORES to the previous code.

To get them in reverse, insert the word REV, as in ZREVRANGE.

ZREVRANGE visits 0 -1 WITHSCORES

But if we’re using a sorted set, it’s more likely we want to range by score, rather than by position.

ZRANGEBYSCORE set min max has a slightly different syntax from ZRANGE.

Since the low and high range numbers are *inclusive* by default, we can make a score number *exclusive* by prefixing it with an opening paren: (. So, this will return all scores where 9 <= score <= 9

ZRANGEBYSCORE visits 9 10000

But the following will return 9 < score <= 10,000:

ZRANGEBYSCORE visits (9 10000

We can also range by both positive and negative values, including infinities.

This returns the entire set.

ZRANGEBYSCORE visits -inf inf

1. **Unions**

The union operation syntax looks like this:

ZUNIONSTORE destination numkeys key [key ...]  
 [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]

destination is the key to store into, and key is one or more keys to union.

destination is the key to store into, and key is one or more keys to union.

numkeys is simply the number of keys you’re about to join, while weight is the optional number to multiply each score of the relative key by (if you have two keys, you can have two weights, and so on).

Finally, aggregate is the optional rule for resolving each weighted score and summing by default, but you can also choose the min or max between many scores.

Let’s use this command to measure the importance of a sorted set of short- codes.

First we’ll create another key that scores our short codes by votes.

Each visitor to a site can vote if they like the site or not, and each vote adds a point.

ZADD votes 2 7wks 0 gog 9001 prag

We want to figure out the most important websites in our system, as some combination of votes and visits.

Votes are important, but to a lesser extent, website visits also carry some weight (perhaps people are so enchanted by the website, they simply forget to vote).

We want to add the two types of scores together to compute a new importance score, while giving votes a weight of double importance—multiplied by two.

ZUNIONSTORE importance 2 visits votes WEIGHTS 1 2 AGGREGATE SUM  
(integer) 3

ZRANGEBYSCORE importance -inf inf WITHSCORES

This command is powerful in other ways too.

For example, if we need to double all scores of a set, we can union a single key with a weight of 2 and store it back into itself.

ZUNIONSTORE votes 1 votes WEIGHTS 2

ZRANGE votes 0 -1 WITHSCORES

10 Expiry

A common use case for a key-value system like Redis is as a fast-access cache for data that’s more expensive to retrieve or compute.

Expiration helps keep the total key set from growing unbounded, by tasking Redis to delete a key- value after a certain time has passed.

SET ice "I'm melting..."

EXPIRE ice 10

EXISTS ice

EXISTS ice

…

Setting and expiring keys is so common that Redis provides a shortcut command called SETEX.

SETEX ice 10 "I'm melting..."

You can query the time a key has to live with TTL.

Setting ice to expire as shown earlier and checking its TTL will return the number of seconds left.

TTL ice

At any moment before the key expires, you can remove the timeout by running PERSIST key.

PERSIST ice

For marking a countdown to a specific time, EXPIREAT accepts a Unix timestamp (as seconds since January 1, 1970) rather than a number of seconds to count up to. <http://www.unixtimestamp.com/index.php>

e.g. 8/10/2014 17:00 = **1412787600)**

In other words, EXPIREAT is for absolute timeouts, and EXPIRE is for relative timeouts.

A common trick for keeping only recently used keys is to update the expire time whenever you retrieve a value.

This is the most recently used (MRU) caching algorithm to ensure your most recently used keys will remain in Redis, while the neglected keys will just expire as normal.

**11 Namespaces**

So far, we’ve interacted only with a single namespace.

For example, if you wrote an internationalized key-value store, you could store different translated responses in different namespaces.

The key greeting could be set to “guten tag” in a German namespace and “bonjour” in French.

When a user selects their language, the application just pulls all values from the namespace assigned.

In Redis nomenclature, a namespace is called a *database* and is keyed by number.

So far, we’ve always interacted with the default namespace 0 (also known as database 0).

Here we set greeting to the English hello.

SET greeting hello  
GET greeting

But if we switch to another database via the SELECT command, that key is unavailable.

SELECT 1

GET greeting

SET greeting "guten tag"  
SELECT 0

GET greeting

Since all databases are running in the same server instance, Redis lets us shuffle keys around with the MOVE command.

Here we move greeting to database 2:

MOVE greeting 2  
SELECT 2

GET greeting

Redis has plenty of other commands for actions such as renaming keys (RENAME), determining the type of a key’s value (TYPE), and deleting a key-value (DEL).

There’s also the painfully dangerous FLUSHDB, which removes all keys from this Redis database, and its apocalyptic cousin, FLUSHALL, which removes all keys from all Redis databases.

Check out the online documentation for the full list of Redis commands. <http://redis.io/commands>

**12 Publish Subscribe**

Earlier we saw blockin queues

Under many circumstances we want a slightly inverted behavior, where several subscribers want to read the announcements of a single publisher

Redis provides some specialized publish-subscribe (or pub-sub) commands.

Let’s improve on the commenting mechanism we made using blocking lists, by allowing a user to post a comment to multiple subscribers (as opposed to just one).

We start with some subscribers that connect to a key, known as a *channel* in pub-sub nomenclature.

Let’s start two more clients and subscribe to the comments channel.

Subscribing will cause the CLI to block.

SUBSCRIBE comments

With two subscribers, we can publish any string we want as a message to the comments channel.

The PUBLISH command will return the integer 2, meaning two subscribers received it.

PUBLISH comments "Check out this shortcoded site! 7wks"

Both of the subscribers will receive a *multibulk reply* (a list) of three items: the string “message,” the channel name, and the published message value.

When your clients want to no longer receive correspondence, they can execute the UNSUBSCRIBE comments command to disconnect from the comments channel or simply UNSUBSCRIBE alone to disconnect from all channels.

13 Durability and server configuration

Redis has a few persistence options.

First is no persistence at all, which will simply keep all values in main memory.

If you’re running a basic caching server, this is a reasonable choice since durability always increases latency.

One of the things that sets Redis apart from other fast-access caches like memcached is its built-in support for storing values to disk.

By default, key- value pairs are only occasionally saved.

You can run the LASTSAVE command to get a Unix timestamp of the last time a Redis disk write succeeded, or you can read the last\_save\_time field from the server INFO output.

You can force durability by executing the SAVE command (or BGSAVE, to asynchronously save in the background).

SAVE

We can alter the rate of storage to disk by adding, removing, or altering one of the save fields in the configuration file

The redis.conf file (or redis.windows.conf) that comes with the distribution—found in /etc/redis on \*nix systems (in msvs/x64/Release/pub in Windows) —is fairly self-explanatory, so we’re going to cover only a portion of the file.

 By default there are three fields, prefixed by the save keyword followed by a time in seconds and a minimum number of keys that must change before a write to disk occurs.

For example, to trigger a save every 5 minutes (300 seconds) if any keys change at all, you would write the following:

save 300 1

The set means if 10,000 keys change, save in 60 seconds; if 10 keys change, save in 300 seconds, and any key changes will be saved in at least 900 seconds (15 minutes).

save 900 1  
save 300 10  
save 60 10000

You can add as many or few save lines as necessary to specify precise thresholds.

Redis is *eventually durable* by default, in that it asynchronously writes values to disk in intervals defined by our save settings, or it is forced to write by client-initiated commands.

This is acceptable for a second-level cache or session server but is insufficient for storing data you need to be durable, like financial data. (**2nd Level Cache – on disk, 1st Level Cache in memory**, which reduces the calls made to the DB server, by conserving data already loaded from the database. Database access is therefore, necessary only when the retrieving data is currently not available in the cache.)

If a Redis server crashes, our users might not appreciate having lost money.

Redis provides an append-only file (appendonly.aof) that keeps a record of all write commands.

If the server crashes before a value is saved, it executes the commands on startup, restoring its state; appendonly must be enabled by setting it to yes in the redis.conf file.

appendonly yes

Then we must decide how often a command is appended to the file.

Setting always is the more durable, since every command is saved.

It’s also slow, which often negates the reason people have for using Redis.

By default everysec is enabled, which saves up and writes commands only once a second.

This is a decent trade-off, since it’s fast enough, and worst case you’ll lose only the last one second of data.

Finally, no is an option, which just lets the OS handle flushing.

It can be fairly infrequent, and you’re often better off skipping the append-only file altogether rather than choosing it.

# appendfsync always  
appendfsync everysec  
# appendfsync no

Append-only has more detailed parameters, which may be worth reading about in the config file when you need to respond to specific production issues.

14 Benchmarking

There are several more advanced settings for speeding up slow query logs, encoding details, making latency tweaks, and importing external config files.

To aid in testing your server configuration, Redis provides an excellent benchmarking tool.

It connects locally to port 6379 by default and issues 10,000 requests using 50 parallel clients.

We can execute 100,000 requests with the -n argument.

**redis-benchmark -n 100000**

**15 Master Slave replication**

Just like other NoSQL databases (such as MongoDB and Neo4j), Redis supports master-slave replication.

One server is the master by default if you don’t set it as a slave of anything. Data will be replicated to any number of slave servers.

Making slave servers is easy.

We first need a copy of our redis.conf file.

**$ copy redis.conf redis-s1.conf**

The file will remain largely the same but with the following changes:

port 6380

slaveof 127.0.0.1 6379

If all went according to plan, you should see something similar to the following in the slave server’s log when you start it:

**$ redis-server ./conf/redis-s1.conf**

[9003] 16 Oct 23:51:52 \* Connecting to MASTER...  
[9003] 16 Oct 23:51:52 \* MASTER <-> SLAVE sync started  
[9003] 16 Oct 23:51:52 \* Non blocking connect for SYNC fired the event.  
[9003] 16 Oct 23:51:52 \* MASTER <-> SLAVE sync: receiving 28 bytes from master  
[9003] 16 Oct 23:51:52 \* MASTER <-> SLAVE sync: Loading DB in memory  
[9003] 16 Oct 23:51:52 \* MASTER <-> SLAVE sync: Finished with success

In the master do this

SADD meetings "StarTrek Pastry Chefs" "LARPers Intl."

If we connect the command line to our slave, we should receive our meeting list.

SMEMBERS meetings