Getting Started With Redis

BY **LORNA MITCHELL**
DEVELOPER ADVOCATE AT IBM AND DZONE MVB

CONTENTS

WHAT IS REDIS?

Redis is an open-source tool that describes itself as a "data structure server." It allows us to store various data types and access them very quickly because the values are held in memory. If you've had experience with memcached, Redis is a pretty similar tool but has support for more interesting data types. Redis can also be used as a message queue, a Pub/Sub system, geospatial data processing, and more.

WHEN TO USE REDIS

Redis is usually used as an auxiliary data store, meaning there will be a main database (e.g. PostgreSQL or another RDBMS) as well as Redis. Redis is used for transient data, for caching values for fast access, and for session data, which stays safe in Redis thanks to automatic failover if a node fails. Redis isn't persistent storage by default, although it can be configured to persist — so never store anything here that you can't afford to lose. Also, since this is an in-memory data store, the size of the data that can be stored will be related to the size of the RAM available.

REDIS DATA TYPES

Redis can work with the following data types:

- **string**: This is your basic key/value pair.
- **hashes**: The value of this type is itself pairs of keys and values; it's useful for storing objects.
- **lists**: Allows multiple values in a particular order; performs very well if you only add or remove items from either end of the list (called "head" and "tail").
- **sets**: Allows multiple unique values in any order. You can add, remove, and check for any value in the set without performance penalties but cannot add duplicate values.
- **sorted sets**: Sets where each value also has a "score." The data is stored sorted by the score, making it very quick to retrieve data using these values.
- **HyperLogLogs**: A structure used to count unique things using a constant amount of memory, rather than an amount of memory proportional to the items counted.

INSTALLING AND USING REDIS

The instructions on redis.io/download should be your starting point for all platforms, and these are actively maintained. You can also ask your usual package manager (brew for OS X; yum or apt on Linux) for a package called "redis" and go from there. This tool has few dependencies, and while it isn't *officially* supported for Windows, there are ports available that will let you at least try it out if that's your platform.

To use Redis in your own software applications, use a client written for your chosen technology stack. There's an incredibly comprehensive list available at redis.io/clients, but here are some links for some of today's popular web programming languages:

Ruby
PHP
Python
NodeJS

REDIS COMMANDS

You can use these commands either from your language-specific library (in which case they will be named the same as the raw commands) or from `redis-cli` itself. The commands aren't case-sensitive, but are usually displayed as uppercase by convention.

Some of the commands we'll use to check on the data or data store itself without any type-specific prefixes. Try these:

`MONITOR` shows every action taking place on the server. This is very useful for debugging, but gets noisy on a busy Redis server or with monitoring running.

Amazon ElastiCache

µs is the new ms

Fully Managed Redis by AWS

Amazon ElastiCache is a web service that makes it easy to deploy, operate, and scale an in-memory data store and cache in the cloud

- High Availability with automatic monitoring and recovery
- Scalability with Managed Redis Cluster
- Hardened by Amazon
- AWS Integration and Support

Try it today for free:
aws.amazon.com/elasticache

Learn More »

amazon web services

`INFO` shows the current Redis config.

`KEYS [pattern]` finds all the keys that match the pattern. You can supply wildcards, too, such as ? to match one character or * to match none/one/many.

```
> MSET hat blue bag red
OK
> KEYS *
1) “hat”
2) “bag”
> KEYS h*
1) “hat”
> KEYS ?a?
1) “hat”
2) “bag”
```

`SCAN cursor [match PATTERN] [COUNT count]` iterates over all keys and returns the matching ones, using a cursor and returning the values in installments. This avoids performance problems related to a `KEYS` query returning a huge number of results. With `SCAN`, the first return value is the next value of the cursor, which you use to get the next batch of matching results. For example, on a (very generically named) set of 21 keys:

```
> scan 0
1) “9”
2)  1) “key:18”
    2) “key:13”
    3) “key:7”
    4) “key:15”
    5) “key:5”
    6) “key:1”
    7) “key:14”
    8) “key:8”
    9) “key:12”
   10) “key:4”
> scan 9
1) “31”
2)  1) “key:20”
    2) “key:3”
    3) “key:21”
    4) “key:19”
    5) “key:2”
    6) “key:6”
    7) “key:9”
    8) “key:16”
    9) “key:17”
   10) “key:11”
> scan 31
1) “0”
2) 1) “key:10”
```

`SCAN` also has sister commands `HSCAN`, `SSCAN`, and `ZSCAN` that will be covered under the hash, set, and sorted set sections respectively.

`TYPE [key]` can return information about the datatype stored in a particular key. This is useful for finding out which of the prefixed commands to use to work with a particular key.

COMMAND PREFIXES

Redis commands sometimes behave differently depending which datatype is being handled. To indicate this, the commands are often prefixed with a character to indicate which datatype they work with. Here's the main ones to look out for:

H: for hashes

S: for sets

Z: for sorted sets (because we already used S).

L: for lists, operations on the left-hand end of the list.

R: for lists (obviously!), operations on the right-hand end of the list.

There are some other conventions such as using M for operations that handle multiple values and B for blocking operations.

NAMESPACING KEYS

Keys in Redis are simply strings; there isn't a built-in way to keep related keys together. By convention, though, we often use the colon character as a separator. This is very handy when used with the `KEYS` command to find all keys matching a particular pattern. Some examples:

```
product:98632
user:janebloggs
article:42:views
article:42:categories
```

By keeping the keys descriptive and well-organized, it becomes easier to find data. It also becomes easier to identify data that is no longer needed. Since Redis is an in-memory datastore, being conscientious about tidying up is very important! There's more information about expiring keys in the “Persisting and Expiring Data” section.

KEY/VALUE COMMANDS

Let's start simple!

`SET`: Sets a value

`GET`: Retrieves it

`GETSET`: Sets the value and fetches the previous one:

```
> SET name Alice
OK
> GET name
“Alice”
> GETSET name Bob
“Alice”
> GET name
“Bob”
```

Working with one key at a time might seem slow, so Redis offers commands for setting and reading multiple entries at one time — `MGET` gets multiple key/values and `MSET` sets multiple key/values:

```
> MSET fruit apple cookie choc-chip
OK
> MGET cookie fruit
1) “choc-chip”
2) “apple”
```

If you're counting items or otherwise working with numbers, there are some shortcut commands to make life easier.

`INCR` and `DECR`: Increment/decrement values (and create them if the key didn't already exist)

`INCRBY` and `DECRYBY`: Add/subtract values from the current value.

```
> GET counter
(nil)
> INCR counter
(integer) 1
> GET counter
“1”
> INCRBY counter 3
(integer) 4
> DECR counter
(integer) 3
> GET counter
“3”
```

Redis uses the string type to support other features, such as bitmaps. This is where values are written to individual bits in a string and used as a very compact way of storing, for example, a bunch of boolean fields for a particular item. The string is considered to be a string of 2^32 bits, all set to zero.

`GETBIT`: Sets a particular bit in the bitmap

`SETBIT` : Gets a particular bit in the bitmap.

```
> SETBIT prefs 5 1
(integer) 0
> SETBIT prefs 3 1
(integer) 0
> GETBIT prefs 100
(integer) 0
> GETBIT prefs 3
(integer) 1
```

HASH COMMANDS

Hashes store multiple fields as the value for a particular key. They are a neater way of keeping related values together than using many similarly-named keys, for example. It's common to use a hash to store an object with properties, making use of the ability to set multiple fields.

Reading and writing individual fields is fairly easy using the `HSET` and `HGET` commands, and there is also support for operations handling multiple fields in one command.

`HSET`: Sets a field in a hash

`HMSET`: Sets multiple fields in a hash

`HGET`: Gets a field from a hash

`HMGET`: Gets multiple fields from a hash:

```
> HSET user:alice name alice
(integer) 1
> HMSET user:alice dress blue food mushrooms
OK
> HGET user:alice food
“mushrooms”
```

We can inspect the hash in a few different ways:

`HLEN:` returns the number of fields in the hash.

`HKEYS:` The fields in the hash.

`HVALS:` The values of the fields in the hash.

`HGETALL:` returns keys interlaced with field values.

`HEXISTS:` checks if this field exists in the hash.

```
> HMSET user:belle dress yellow name belle food cake
OK
> HLEN user:belle
(integer) 3
> HKEYS user:belle
1) “dress”
2) “name”
3) “food”
> HVALS user:belle
1) “yellow”
2) “belle”
3) “cake”
> HGETALL user:belle
1) “dress”
2) “yellow”
3) “name”
4) “belle”
5) “food”
6) “cake”
> HEXISTS user:belle shoes
(integer) 0
```

We can also inspect within a hash using the `HSCAN` command, which is especially useful on large hashes or when looking for a specific group of fields within a hash. This works like the `SCAN` command mentioned above, using cursors to paginate the results as appropriate.

`HSCAN` finds hashes and fields within them:

```
> HSCAN user:ariel 0
1) “0”
2) 1) “name”
   2) “ariel”
   3) “superpower”
   4) “mermaid”
   5) “dress”
   6) “green tail”
   7) “sisters”
   8) “6”
> HSCAN user:ariel 0 MATCH s*
1) “0”
2) 1) “superpower”
   2) “mermaid”
   3) “sisters”
   4) “6”
```

LIST COMMANDS

Lists in Redis are a chain of values implemented by a linked-list data structure. Lists are very performant to work with when only the values near the beginning or end of the list are operated on. In Redis, these are visualized as the “left” and “right” ends of a list and have commands prefixed with L and R respectively. To add things onto either end of the list is a `PUSH` command and to remove and return things from an end is a `POP` command.

Lists are useful for implementing stacks or queues in Redis.

`LPUSH`: Adds a new value onto the left end of the list.

`RPUSH`: Adds a new value onto the right end of the list.

`LRANGE`: Returns some items from the list, specifying how many and where to start.

```
> LPUSH rhyme little
(integer) 1
> LPUSH rhyme twinkle
(integer) 2
> LPUSH rhyme twinkle
(integer) 3
> RPUSH rhyme star
(integer) 4
> LRANGE rhyme 0 -1
1) “twinkle”
2) “twinkle”
3) “little”
4) “star”
> LRANGE rhyme 1 2
1) “twinkle”
2) “little”
```

`LPOP`: Removes the leftmost value from the list and return it.

`RPOP`: Removes the rightmost value from the list and returns it.

`LLEN`: Returns how many items are in the list.

```
> LLEN rhyme
(integer) 4
> RPOP rhyme
“star”
> LPOP rhyme
“twinkle”
> LRANGE rhyme 0 -1
1) “twinkle”
2) “little”
```

LSET: Puts a value at a particular location in the list (must be within the current range of indexes of the list).

LINDEX: Gets a value from a specific position from either the left or right ends of the list.

LINSERT: Places a value before or after a particular other value anywhere in the list (performs better near the left end because there are fewer items to traverse).

```
> LPUSH rainbow yellow
(integer) 1
> LPUSH rainbow orange
(integer) 2
> LPUSH rainbow red
(integer) 3
> LSET rainbow 1 green
OK
> LINDEX rainbow 2
“yellow”
> LINSERT rainbow BEFORE green amber
(integer) 4
> LRANGE rainbow 0 -1
1) “red”
2) “amber”
3) “green”
4) “yellow”
```

LTRIM cuts the list down to size — very handy for a short buffer of things:

```
> LPUSH recent_orders latte
(integer) 1
> LPUSH recent_orders chai
(integer) 2
> LPUSH recent_orders smoothie
(integer) 3
> LRANGE recent_orders 0 -1
1) “smoothie”
2) “chai”
3) “latte”
> LTRIM recent_orders 0 1
OK
> LRANGE recent_orders 0 -1
1) “smoothie”
2) “chai”
> LPUSH recent_orders mocha
(integer) 3
> LTRIM recent_orders 0 1
OK
> LRANGE recent_orders 0 -1
1) “mocha”
2) “smoothie”
```

SET COMMANDS

Redis has both sets and sorted sets, which are pretty similar but do use separate commands. This section covers the non-sorted variety first, with the sorted ones detailed a little later on. Sets are keys with multiple unique values, which can be useful for a use case where values cannot be duplicated. Sets are also great for comparing and combining together to answer questions that would be tricky with, for example, a traditional relational database store.

SADD: Puts a value into a set (if it already exists, it won’t be added again).

SMEMBERS: Shows all members of a set.

SSCAN: Inspects the contents of a set; it works like SCAN and uses a cursor and optionally a pattern to match:

```
> SADD post:1:tags tech
(integer) 1
> SADD post:1:tags javascript
(integer) 1
> SADD post:1:tags tips
(integer) 1
> SADD post:1:tags couchdb
(integer) 1
> SADD post:2:tags couchdb
(integer) 1
> SADD post:2:tags pouchdb
(integer) 1
> SADD post:2:tags pouchdb
(integer) 0
> SMEMBERS post:2:tags
1) “pouchdb”
2) “couchdb”
> SSCAN post:2:tags 0
1) “0”
2) 1) “pouchdb”
   2) “couchdb”
```

SUNION: Returns the combined contents of two or more sets, removing duplicates.

SINTER: Returns values that appear in two or more sets.

SINTERSTORE: Is the same as SINTER, but stores the result in the named key rather than returning it:

```
> SINTER post:1:tags post:2:tags
1) “couchdb”
> SINTERSTORE overlap post:1:tags post:2:tags
(integer) 1
> SMEMBERS overlap
1) “couchdb”
> SUNION post:1:tags post:2:tags
1) “couchdb”
2) “tips”
3) “javascript”
4) “pouchdb”
5) “tech”
```

SPOP: Removes and returns a random element from the set.

SRANDMEMBER: Returns a random value from the set without removing it from the set:

```
> SADD coin heads
(integer) 1
> SADD coin tails
(integer) 1
> SRANDMEMBER coin
“tails”
> SRANDMEMBER coin
“heads”
> SRANDMEMBER coin
“heads”
> SPOP coin
“heads”
> SMEMBERS coin
1) “tails”
```

SORTED SET COMMANDS

Sorted sets *sound* a lot like sets, but in truth, the minor differences make them handy for quite different use cases. For sets, we hold multiple things alongside one another, but the sorted sets are great for counting things, leaderboards, and other short-term counting tasks. Since the sorting for sorted sets goes from low to high by default, many of the commands for working with sorted sets (all prefixed with Z) have matching “sister” commands that reverse the order. The reverse order means that the highest scoring item appears first in the list.

`ZADD`: Puts a value into the sorted set with a score.

`ZINCR`: Adds to the score of a particular value in the sorted set (create the set and value as needed).

`ZRANG:` Gets some or all members of the set in order of score (with smallest first).

`WITHSCORES`: Gets the scores as well as an additional return value after each item.

`ZREVRANGE`: Similar to `ZRANGE` but is sorted with the highest scores first, which is useful for most-viewed/commented/voted-type features:

```
> ZADD product_views 1 table
(integer) 1
> ZINCRBY product_views 1 bench
“1”
> ZINCRBY product_views 1 bench
“2”
> ZINCRBY product_views 1 wheelbarrow
“1”
> ZRANGE product_views 0 -1
1) “table”
2) “wheelbarrow”
3) “bench”
> ZREVRANGE product_views 0 0 WITHSCORES
1) “bench”
2) “2”
```

`ZSCORE`: Gets the current score of a particular value.

`ZRANK`: Finds out what “position” a value has if ranked in order of score, starting from zero.

`ZREVRANK`: Finds out what “position” a value has if ranked from highest score to lowest, starting from zero:

```
> ZINCRBY product_views 1 bench
“3”
> ZSCORE product_views bench
“3”
> ZRANK product_views bench
(integer) 2
> ZREVRANK product_views bench
(integer) 0
```

PERSISTING AND EXPIRING DATA

We’ve touched on the importance of keeping Redis data tidy. One way to achieve this is to set keys to expire after a certain amount of time so that they “decay” out of storage. On the flip side, sometimes you really do want your data to remain intact. In this section, we’ll also look at the configuration settings for Redis that will allow some or all of your data to survive a program restart.

EXPIRING KEYS

Setting keys to expire is a good way to make the most of the storage available to Redis because it avoids old data hanging around and taking up space. Note that expiry is only at the key level — it isn’t possible to expire nested values, so it’s useful to consider expiry policies when designing data structures.

`EXPIRE`: Sets how long (in seconds) this key should live for; after this time, the key will be deleted.

`SETEX`: Combines the `SET` and `EXPIRE` commands since the two are so often used together.

`EXPIREAT`: Defines at what time (in Unix timestamp format) this key should be deleted.

`TTL`: Defines the time until this key will expire, or uses -1 if it isn’t set to expire and -2 if the key doesn’t exist at all.

`PERSIST`: Removes the expiry on a key, so it will not be automatically deleted:

```
> SET temp_value 42
OK
> EXPIRE temp_value 10
(integer) 1
> TTL temp_value
(integer) 3
> TTL temp_value
(integer) -2
> SET temp_value 42
OK
> EXPIRE temp_value 10
(integer) 1
> PERSIST temp_value
(integer) 1
> TTL temp_value
(integer) -1
```

CONFIGURING PERSISTENCE

By default, Redis isn't reliably persistent. The out-of-the-box settings have it snapshot to disk at intervals, but rest assured that this will never have happened immediately before the server fails! This is basically by design; Redis is a blisteringly fast datastore because it stores in-memory and doesn't write to disk. That said, there are definitely some situations where more persistence than "probably none" would be desirable.

Redis gives two options for persistence:

- Snapshotting is used periodically, but this period can be configured to meet the requirements. Beware that frequent snapshotting will affect performance on busy systems. The shapshot feature is also a good way to take a backup of Redis. This is technically referred to as the RDF (Redis Data File) option.
- Changelog writes each Redis command to a file, syncing it to disk in the background at (configurable) intervals. This AOF (Append Only File) approach is a good way to capture changes and also gives a useful log that can be parsed if required.

In practice, most Redis installations will use a combination of these two approaches to give the best possible performance/recovery options. There is excellent documentation available for this that also covers the details of backing up with RDF files and how to restore them.

Redis gives two options for persistence:

1. Snapshotting is used periodically, but this period can be configured to meet the requirements. Beware that frequent snapshotting will affect performance on busy systems. The shapshot feature is also a good way to take a backup of Redis. This is technically referred to as the RDF (Redis Data File) option.
2. Changelog writes each Redis command to a file, syncing it to disk in the background at (configurable) intervals. This AOF (Append Only File) approach is a good way to capture changes and also gives a useful log that can be parsed if required.

In practice, most Redis installations will use a combination of these two approaches to give the best possible performance/recovery options. There is excellent documentation available for this that also covers the details of backing up with RDF files and how to restore.

PUB/SUB WITH REDIS

Redis has built-in Publish/Subscribe support, and it's surprisingly simple to use. Subscribers listen on a channel and publishers broadcast messages to that channel. This requires use of multiple clients, as demonstrated in the examples below. Messages published to a channel while a client is subscribed will arrive on that client with three elements to the message:

1. The type of message, either "message" or "subscribe" or "unsubscribe."
2. The channel this relates to.
3. The message itself.

SUBSCRIBE starts listening to a channel or channels for messages:

```
> SUBSCRIBE whispers chatter
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "whispers"
3) (integer) 1
1) "subscribe"
2) "chatter"
3) (integer) 2
```

PUBLISH sends a message to a channel:

```
> PUBLISH whispers "hello ... world"
(integer) 1
```

And on the client:

```
1) "message"
2) "whispers"
3) "hello ... world"
```

QUEUES IN REDIS

Redis is such a nice, lightweight data storage tool that it makes an easy addition to almost any application. It's also possible to build upon the Redis features to create extra functionality. A good example of this is using a Redis installation as the basis for a simple queue. There are some fairly fully featured queue solutions based on Redis (such as Resque: resque.github.io), but a simple solution can be created using the list datatype.

By pushing items onto the left-hand end of a list and having the worker (the name for a queue processor) take items from the right-hand end, we can create a simple queue. To add items into the queue, we can use LPUSH:

```
> LPUSH todo breakfast
(integer) 1
> LPUSH todo newspaper
(integer) 2
> LPUSH todo email
(integer) 3
```

For the component that will be processing the queue items, we could use RPOP to fetch things from the right-hand end of the list. This approach needs caution, however; if the list is empty and the worker repeatedly tries to read from it, then there will be a lot of wasted resources spent trying to read an empty list! Instead, we

can use the blocking features for our list and take items with BRPOP. Using the blocking commands means that the command will wait a little while to see if an item appears in an empty list before returning.

```
> BRPOP todo 1
1) “todo”
2) “breakfast”
> BRPOP todo 1
1) “todo”
2) “newspaper”
> BRPOP todo 1
1) “todo”
2) “email”
> BRPOP todo 1
(nil)
(1.03s)
```

There are many features that can be implemented by building on top of the data types made available by Redis.

TRANSACTIONS IN REDIS

Sometimes, we want to combine operations in Redis, having them happen “at the same time” or not at all. In support of this, Redis supports transactions. Be aware that the transaction won’t be rolled back if one of the commands fails; transactions in Redis are more like a queue and then they batch operate on all those commands at once.

`MULTI`: Starts collecting commands into a transaction.

`DISCARD`: Throws away the commands (we don’t want to execute them, after all).

`EXEC`: Runs all the commands, and their results will appear in turn:

```
> SET a 1
OK
> MULTI
OK
> INCR a
QUEUED
> DISCARD
OK
> GET a
“1”
> MULTI
OK
> INCR a
QUEUED
> INCR a
QUEUED
> EXEC
1) (integer) 2
2) (integer) 3
>
```

ABOUT THE AUTHOR

LORNA MITCHELL is based in Yorkshire, UK; she is a Developer Advocate with IBM Watson Data Platform, a published author and experienced conference speaker. She brings her technical expertise on a range of topics to audiences all over the world with her writing and speaking engagements, always delivered with a very practical slant. In her spare time, Lorna blogs at lornajane.net.

RESOURCES AND FURTHER READING

Homepage and brilliant documentation

Redis Cookbook by Tiago Macedo, Fred Oliveira

The Little Redis Book by Karl Seguin (free)

7 Databases in 7 Weeks by Eric Redmond and Jim R. Wilson (has an excellent chapter on Redis)

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com