Redis

Introduction

Redis is opensource in-memory database that's mostly designed to be a key-value store.

- In-memory means that data is primarily stored in RAM. This makes the speed of accessing your data generally faster / more consistent when compared to other databases that store your data onto disk.
- Key-value store means that the data is accessed based on unique keys that you give it. This means that you don't have the normal flexibility of a SQL-like solution, so your data likely has to be denormalized and you may need to be clever in how you structure/access it.

What else makes Redis unique? Although Redis is a key-value store, the values can be arbitrary values or collections: lists, sets, sorted sets, etc.. It also provides some advanced features: master-slave replication, persisting to disk, clustering, scripting support, different strategies for automatically deleting data, etc..

> **NOTE**: The most important thing to know when using Redis is that your application needs to be able to handle data loss / inconsistent data. You'll see why when you read the section on persistence and replication. Transactions won't help -- Redis transactions aren't like normal database transactions.

Setup

The following setup instructions are for Ubuntu/Mint variant of Linux. They were tested on Linux Mint 18.2.

Server

You can use the package manager on your system to install redis...

```
$ sudo apt-get install redis-server
```

Once installed, you can run the following to figure out what version of Redis you're running...

```
$ redis-server -v
Redis server v=3.0.6 sha=00000000:0 malloc=jemalloc-3.6.0 bits=64
build=687a2a319020fa42
```

Start Redis by running…

```
$ redis-server
```

```
3468:C 27 Nov 07:13:14.745 # Warning: no config file specified, using the
default config. In order to specify a config file use redis-server
/path/to/redis.conf
3468:M 27 Nov 07:13:14.745 * Increased maximum number of open files to
10032 (it was originally set to 1024).
                _.-``__ ''-._
           _.-``    `.  `_.  ''-._           Redis 3.0.6 (00000000/0) 64 bit
       .-`` .-```.  ```\/    _.,_ ''-._
      (    '      ,       .-`  | `,    )     Running in standalone mode
      |`-._`-...-` __...-.``-._|'` _.-'|     Port: 6379
      |    `-._   `._    /     _.-'    |     PID: 3468
       `-._    `-._  `-./  _.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |           http://redis.io
       `-._    `-._`-.__.-'_.-'    _.-'
      |`-._`-._    `-.__.-'    _.-'_.-'|
      |    `-._`-._        _.-'_.-'    |
       `-._    `-._`-.__.-'_.-'    _.-'
           `-._    `-.__.-'    _.-'
               `-._        _.-'
                   `-.__.-'
```

CLI Client

Connect to Redis using the provided commandline client…

```
$ redis-cli
> set test1 "hi!"
OK
> get test1
"hi!"
> del test1
(integer) 1
> get test1
(nil)
>
```

For a full list of commands that you can use with the commandline client, see https://redis.io/commands.

GUI Client

If you want a GUI client, install redis-commander. On a mint/ubuntu box perform the following commands...

```
$ sudo apt-get install npm nodejs nodejs-legacy
$ npm install -g redis-commander
$ redis-commander
```

Once the process has started, open a browser and go to http://localhost:8081/.

Data Types

Remember that Redis is a key-value store. For some unique key, it can hold a value. You can think of it like a giant Java HashMap.

The value in a key-value entry can be one of the following 5 data types:

- Strings -- arbitrary values
  (similar to Java String or byte[])
- Lists -- a collection that can have items appended to the front or back
  (similar to Java List, but you can't insert data in the middle)
- Sets -- a collection that will ignore duplicate items, also no control of sort order
  (similar to Java HashSet)
- Sorted Sets -- a collection that will ignore duplicate items, but also sorted by item
  (similar to Java TreeSet)
- Hashes -- a collection of key-value pairs
  (similar to Java HashMap)

String

You can think of a string in Redis as an arbitrary byte array rather than text -- that is, strings are just raw data and don't have properties to common to textual strings like encoding (e.g. UTF-8, US-ASCII, etc..) or locale (e.g. US_ENGLISH, TURKISH, etc..).

Common usecases include:

- storing serialized objects
- storing blobs such as images

  NOTE: The maximum length is 512 megabytes. Anything larger than 512 megabytes won't work with Redis.

Get/set (GET/SET)

Get/set a string.

> **NOTE**: In addition to this, there's also a GETSET (sets a value and return the old value) and SETNX (sets if doesn't exist).

```
> set user:1 "firstname:steve lastname:stevenson"
OK
> get user:1
"firstname:steve lastname:stevenson"
```

Get/set many at once (MGET/MSET)

Get/set multiple strings.

> **NOTE**: In addition to this, there's MSETNX (sets if doesn't exist).

> **NOTE**: The documentation claims that MSET is an atomic operation, but I don't think this will be atomic with sharding/clustering with clustering (depending on how it works).

```
> mset key1 "a" key2 "b"
OK
> mget key1 key2
1) "a"
2) "b"
```

Get substring (GETRANGE)

Get part of a string (similar to substring).

Arguments for this are not obvious. They are...

1. key to get
2. substring start position (if negative, starts from the end)
3. substring end position (if negative, starts from the end)

> **NOTE**: If your substring goes out of bounds, they automatically get capped to the length of the string. See last getrange call in the example above.

```
> set user:1 "firstname:steve lastname:stevenson"
OK
```

```
> getrange user:1 0 5
"firstn"
> getrange user:1 5 -5
"name:steve lastname:steve"
> getrange user:1 5 5000
"name:steve lastname:stevenson"
```

Get length (STRLEN)

Get length of string.

```
> set user:1 "firstname:steve lastname:stevenson"
OK
> get user:1
"firstname:steve lastname:stevenson"
> strlen user:1
(integer) 34
```

Append to an existing string (APPEND)

Append data onto an existing string.

```
> set user:1 "firstname:steve lastname:stevenson"
OK
> get user:1
"firstname:steve lastname:stevenson"
> append user:1 " address:123 fake st"
(integer) 54
> get user:1
"firstname:steve lastname:stevenson address:123 fake st"
```

Increment/decrement by some integer (INCR/DECR)

These operations will increment/decrement a string, provided that it's an integer.

> **NOTE**: In addition to this, there's INCRBY/DECRBY (moves the values by more than 1), and INCRBYFLOAT (floating point number support).

```
> set key1 "0"
OK
> incr key1
(integer) 1
```

```
> incr key1
(integer) 2
> decr key1
(integer) 1
```

List

You can think of a list in Redis as a Java LinkedList of strings. Although you have complete control over where you add/remove/get/insert data the list, it may be slow to set/get items that aren't near the front or back of the list (assuming that your list is very large?).

Common usecases include:

- queues and stacks
- cache of a larger list (e.g. top 30 replies on some reddit post)

> **NOTE**: The maximum length of a list seems to be dependent on architecture? It seems that all architectures support at least 2 billion elements. The maximum size of a single string within a list is probably still 512 megabytes.

Add/remove to either end (LPUSH/RPUSH/LPOP/RPOP)

Add an item to or remove an item from either end of the list. LPUSH/LPOP with add/remove on the left side, while RPUSH/RPOP will add/remove on the right side.

> **NOTE**: In addition to this, there's LPUSHX/RPUSHX (only adds if the list exists), BLPOP/BRPOP (removes but blocks if nothing is available), RPOPLPUSH (removes from right and adds to left), and BRPOPLPUSH (removes from right and adds to left but blocks if nothing is available).

```
> lpush mylist "bbb"
(integer) 1
> rpush mylist "ccc"
(integer) 2
> lpush mylist "aaa"
(integer) 3
> lrange mylist 0 3
1) "aaa"
2) "bbb"
3) "ccc"
> lpop mylist
"aaa"
> rpop mylist
```

```
"ccc"
> lrange mylist 0 3
1) "bbb"
>
```

Insert before/after value (LINSERT)

Inserts an item before or after some value (not index). The value must exist in the list.

```
> lpush mylist2 "bbb"
(integer) 1
> linsert mylist2 after "bbb" "xxx"
(integer) 2
> linsert mylist2 after "bbb" "yyy"
(integer) 3
> linsert mylist2 after "bbb" "zzz"
(integer) 4
> lrange mylist2 0 4
1) "bbb"
2) "zzz"
3) "yyy"
4) "xxx"
```

Replace at index (LSET)

Sets (replaces) an item at some index.

> **NOTE**: If the index is negative, it starts moving from the end of the list.

```
> lrange mylist3 0 4
1) "bbb"
2) "ccc"
3) "bbb"
4) "aaa"
> lset mylist3 1 "zzz"
OK
> lset mylist3 -1 "yyy"
OK
> lrange mylist3 0 4
1) "bbb"
2) "zzz"
3) "bbb"
```

```
4) "yyy"
>
```

Remove by value (LREM)

Removes the occurrences of some item from the list.

Arguments for this are not obvious. They are...

1. key to operate on (name of list)
2. maximum number of occurrences to remove
   0 means remove all
   > 0 means remove a max of this many from the start of the list (starting from left-side)
   < 0 means remove a max of this many from the end of the list (starting from right-side)
3. value to remove

```
> lrange mylist4 0 6
1) "ddd"
2) "bbb"
3) "ddd"
4) "ccc"
5) "bbb"
6) "aaa"
> lrem mylist4 0 "bbb"
(integer) 2
> lrange mylist4 0 6
1) "ddd"
2) "ddd"
3) "ccc"
4) "aaa"
> lrem mylist4 -1 "ddd"
(integer) 1
> lrange mylist4 0 6
1) "ddd"
2) "ccc"
3) "aaa"
```

Get sublist (LRANGE)

Get a sub-list specified by start and end offsets. If the offsets are negative, it means that they start from the end of the list.

```
> lrange mylist6 0 100
```

```
1) "h"
2) "g"
3) "e"
4) "d"
5) "c"
6) "b"
7) "a"
> lrange mylist6 0 2
1) "h"
2) "g"
> lrange mylist6 -2 -1
1) "b"
2) "a"
```

Get length (LLEN)

Get the length of a list.

```
> lpush mylist7 "a"
(integer) 1
> lpush mylist7 "b"
(integer) 2
> lpush mylist7 "c"
(integer) 3
> llen mylist7
(integer) 3
```

Trim to sublist (LTRIM)

Trims the list to some sub-list specified by start and end offsets. If the offsets are negative, it means that they start from the end of the list.

```
> lrange mylist6 0 100
1) "h"
2) "g"
3) "e"
4) "d"
5) "c"
6) "b"
7) "a"
> ltrim mylist6 2 4
OK
```

```
> lrange mylist6 0 100
1) "e"
2) "d"
3) "c"
> ltrim mylist6 0 -2
OK
> lrange mylist6 0 100
1) "e"
2) "d"
```

Set

You can think of a set in Redis as a Java HashSet<String>. That means that it will automatically filter out duplicate strings and no order is maintained. Just like a HashSet…

- adding, removing, and checking to see if a string exists are fast operations.
- intersections/unions/differences can be done much quicker than with a list.

Common usecases include:

- any kind of caches / lookups (e.g. a cache of users logged in)

  NOTE: The maximum length of a set is unknown, but it's probably the same as that with list: seems to be dependent on architecture but will support at least 2 billion elements. The maximum size of a single string within a set is probably still 512 megabytes.

Add/remove (SADD/SREM)

Adds/removes members from the set.

```
> sadd myset1 "aaa"
(integer) 1
> sadd myset1 "aaa"
(integer) 0
> sadd myset1 "bbb"
(integer) 1
> sadd myset1 "ccc"
(integer) 1
> smembers myset1
1) "aaa"
2) "bbb"
3) "ccc"
> srem myset1 "ccc"
(integer) 1
```

```
> smembers myset1
1) "aaa"
2) "bbb"
```

> **NOTE**: The example above may look like it's respecting insertion order but it's probably a fluke.

Test if contained (SISMEMBER)

Tests to see if some value is a member of a set.

```
> smembers myset1
1) "aaa"
2) "bbb"
> sismember myset1 "ccc"
(integer) 0
> sismember myset1 "bbb"
(integer) 1
> sismember myset1 "aaa"
(integer) 1
```

Get all (SMEMBERS)

Get all members of a set.

> **NOTE**: Do not use this if the set is large. Use SSCAN instead.

```
> smembers myset1
1) "aaa"
2) "bbb"
3) "ccc"
```

Get size (SCARD)

Gets the size of a set (number of members/values in a set).

```
> smembers myseta
1) "aaa"
2) "ccc"
3) "000"
> scard myseta
(integer) 3
```

Difference/union/intersection between sets (SDIFF/SINTER/SUNION)

Performs a difference/union/intersection between 2 or more sets and returns the result.

> **NOTE**: There's also SDIFFSTORE/SINTERSTORE/SUNIONSTORE -- these do the same thing but the results are stored in a new set rather than being returned to you.

```
> smembers myseta
1) "aaa"
2) "ccc"
3) "000"
> smembers mysetb
1) "zzz"
2) "xxx"
3) "000"
> sunion myseta mysetb
1) "000"
2) "ccc"
3) "aaa"
4) "xxx"
5) "zzz"
> sinter myseta mysetb
1) "000"
> sdiff myseta mysetb
1) "aaa"
2) "ccc"
> sdiff mysetb myseta
1) "xxx"
2) "zzz"
```

Move item to another set (SMOVE)

Move a member of one set to another set.

> **NOTE**: The documentation claims that this is an atomic operation, but depending on how clustering/sharding works this may not be atomic across multiple nodes.

```
> smembers myseta
1) "aaa"
2) "ccc"
3) "000"
> smembers mysetb
```

```
1) "zzz"
2) "xxx"
3) "000"
> smove myseta mysetb "000"
(integer) 1
> smembers myseta
1) "aaa"
2) "ccc"
> smembers mysetb
1) "zzz"
2) "xxx"
3) "000"
> smove myseta mysetb "fake"
(integer) 0
> smembers myseta
1) "aaa"
2) "ccc"
> smembers mysetb
1) "zzz"
2) "xxx"
3) "000
```

Remove/get a random item (SPOP/SRANDMEMBER)

Remove/get a member from a set (at random).

```
> smembers mysetb
1) "zzz"
2) "xxx"
3) "000"
> spop mysetb
"zzz"
> srandmember mysetb
"000"
> smembers mysetb
1) "xxx"
2) "000"
```

Iterate over set (SSCAN)

SSCAN returns you entries from the set in an iterative manner -- meaning you get things in chunks rather than getting everything at once. This is useful if you want to implement something like pagination.

The usage pattern for SSCAN is SSCAN key cursor [MATCH pattern] [COUNT count]

- key is the name of your set.
- cursor is a value returned from a previous scan that you can pass in to continue scanning.
  (initial value is 0)
- [MATCH pattern] is an optional glob-style argument that will filter your results.
  (for example MATCH *Alabama*)
- [COUNT count] is an optional argument that will try to keep the results of a single scan to the size you specify.
  (for example COUNT 10)

In the example below, we're scanning over the elements of a set 5 at a time. Notice that…

- the first value being returned by sscan is a number, followed by a list of items. The number can be thought of as a "cursor" -- you start at 0 and pass the updated value to subsequent sscans until you get back 0. When you get back 0, that's how you know it's the end.
- certain sscan results may return more/less than the desired count. In the first sscan, we ask for 5 but get back 6.

```
> smembers mysetc
 1) "n"
 2) "j"
 3) "m"
 4) "h"
 5) "k"
 6) "c"
 7) "l"
 8) "o"
 9) "a"
10) "g"
11) "d"
12) "i"
13) "b"
14) "f"
15) "e"
16) "p"
> sscan mysetc 0 COUNT 5
1) "10"
2) 1) "n"
   2) "l"
   3) "c"
```

```
   4) "b"
   5) "m"
   6) "h"
> sscan mysetc 10 COUNT 5
1) "5"
2) 1) "e"
   2) "j"
   3) "o"
   4) "a"
   5) "g"
> sscan mysetc 5 COUNT 5
1) "0"
2) 1) "f"
   2) "k"
   3) "d"
   4) "i"
   5) "p"
```

SSCAN has the following gotchas (from the documentation)…

1. A given element may be returned multiple times. It is up to the application to handle the case of duplicated elements, for example only using the returned elements in order to perform operations that are safe when re-applied multiple times.
2. Elements that were not constantly present in the collection during a full scan, may be returned or not: it is undefined.

Hash (Map)

You can think of a hash in Redis as a Java HashMap<String,String>. That means that…

- it maps keys to values.
- duplicate keys aren't allowed.
- no order is maintained.
- getting / setting values when you know the key is a fast operation.

You'll notice that may of the operations for string and set have hash equivalents.

Common usecases include:

- storing objects, but when you need to look up individual members of that object

  NOTE: Instead of calling it maps and keys, Redis calls it hashes and fields. I use the terms interchangeably in this doc.

NOTE: Here's what the Redis documentation says about hashes… "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)."

NOTE: The maximum size of a field/value within a hash is probably still 512 megabytes.

Get/set (HGET/HSET)

Get/set a value of a key.

```
> hset mymap2 key1 val1
(integer) 1
> hset mymap2 key2 val2
(integer) 1
> hset mymap2 key3 val3
(integer) 1
> hget mymap2 key1
"val1"
> hget mymap2 key2
"val2"
> hget mymap2 key3
"val3"
```

Get/set many at once (HMGET/HMSET)

Get/set the values of multiple keys fields.

```
> hmset mymap3 k1 v1 k2 v2 k3 v3
OK
> hmget mymap3 k1 k2 k3
1) "v1"
2) "v2"
3) "v3"
```

Get keys (HKEYS)

Get all keys.

NOTE: Do not use this if the map is large. Use HSCAN instead.

```
> hkeys mymap3
1) "k1"
2) "k2"
3) "k3"
```

Get values (HVALS)

Get all values.

> **NOTE**: Do not use this if the map is large. Use HSCAN instead.

```
> hvals mymap3
1) "v1"
2) "v2"
3) "v3"
```

Get all (HGETALL)

Get all key-value pairs.

> **NOTE**: Do not use this if the map is large. Use HSCAN instead.

```
> hgetall mymap3
1) "k1"
2) "v1"
3) "k2"
4) "v2"
5) "k3"
6) "v3"
```

Get size (HLEN)

Get the number of key-value pairs in the map.

```
> hlen mymap3
(integer) 3
```

Delete by key (HDEL)

Delete a an entry in the map by key.

```
> hgetall mymap3
```

```
1) "k1"
2) "v1"
3) "k2"
4) "v2"
5) "k3"
6) "v3"
> hdel mymap3 k1
(integer) 1
> hgetall mymap3
1) "k2"
2) "v2"
3) "k3"
4) "v3"
```

Check if key exists (HEXISTS)

Test to see if a key exists in the map.

```
> hgetall mymap4
1) "key1"
2) "val1"
3) "key2"
4) "val2"
5) "key3"
6) "val3"
> hexists mymap4 key1
(integer) 1
> hexists mymap4 key11111
(integer) 0
```

Get size of value (HSTRLEN)

Get the length of a value in the map.

```
> hgetall mymap4
1) "key1"
2) "val1"
3) "key2"
4) "val2"
5) "key3"
6) "val3"
> hstrlen mymap4 key1
```

```
(integer) 4
```

Increase/decrease value (HINCRBY)

Increase/decrease the value in a map by some integer.

> **NOTE**: There's also HINCRBYFLOAT if you're dealing with floating point instead of integers.

```
> hgetall mymap6
1) "k1"
2) "1"
3) "k2"
4) "2"
5) "k3"
6) "3"
> hincrby mymap6 k1 10
(integer) 11
> hget mymap6 k1
"11"
```

Iterate over hash (HSCAN)

HSCAN returns you entries from the map in an iterative manner -- meaning you get things in chunks rather than getting everything at once. This is useful if you want to implement something like pagination.

The usage pattern for HSCAN is HSCAN key cursor [MATCH pattern] [COUNT count]

- key is the name of your map.
- cursor is a value returned from a previous scan that you can pass in to continue scanning.
  (initial value is 0)
- [MATCH pattern] is an optional glob-style argument that will filter your results.
  (for example MATCH *Alabama*)
- [COUNT count] is an optional argument that will try to keep the results of a single scan to the size you specify.
  (for example COUNT 10)

HSCAN is just like SSCAN, so look up SSCAN to see a usage example.

> **NOTE**: One thing to know is that the count value is just a hint to the implementation and can be ignored by the implementation. On a set with 100 elements, my count of 5 was

ignored and the entire set was returned. This is known behaviour according to this: https://github.com/antirez/redis/issues/1723#issuecomment-42385837.

Sorted Set

You can think of a sorted set in Redis as a Java TreeSet<String>. The main difference is that rather than passing in comparator, for Redis we directly supply a "score" in which the set members are ordered by (keyed sort vs comparison sort). Just like a TreeSet…

- most normal set operations apply (see the section on Set)
- iterating in order is possible and reasonably fast

The terminology here is super important…

- SCORE → floating point number that decides the sort order
- RANK → position/index within the sorted set (e.g. lowest item is rank 0, second lowest is rank 1,etc..)

Common usecases include:

- prioritizing some set of items (e.g. jobs)
- caching of a high-score list for a game

NOTE: The maximum length of a sorted set is unknown, but it's probably the same as that with list: seems to be dependent on architecture but will support at least 2 billion elements. The maximum size of a single string within a set is probably still 512 megabytes.

Add/remove (ZADD/ZREM)

Add or remove items from the sorted set.

ZADD has some optional args: ZADD key [NX|XX] [CH] [INCR] score member [score member...]

- NX = only update, don't add new elements
- XX = only add new elements, don't update if already exists
- CH = return value of ZADD becomes number of elements changed, so rather than getting just the number of new elements added you'll get the number of elements new elements PLUS the number of updated elements.

```
> ZADD sset2 102 c 100 a 101 b
(integer) 3
> ZADD sset2 CH 101.4444 c 100 a
(integer) 1
> ZRANGE sset2 0 1000 WITHSCORES
1) "a"
```

```
2) "100"
3) "b"
4) "101"
5) "c"
6) "101.4444"
> ZREM sset2 c
(integer) 1
> ZRANGE sset2 0 1000 WITHSCORES
1) "a"
2) "100"
3) "b"
4) "101"
```

Remove range (ZREMRANGEBYRANK/ZREMRANGEBYSCORE)

Remove the items in some range of your set.

- ZREMRANGEBYRANK removes all items in a index/rank range
- ZREMRANGEBYSCORE remove all items in a score range

  NOTE: There's also ZREMRANGEBYLEX (unsure about this? Look up ZRANGEBYLEX).

```
> ZRANGE sset10 0 10000
1) "a"
2) "b"
3) "c"
4) "d"
5) "e"
6) "f"
> ZREMRANGEBYRANK sset10 1 3
(integer) 3
> ZRANGE sset10 0 10000
1) "a"
2) "e"
3) "f"
> ZRANGE sset11 0 10000 WITHSCORES
 1) "a"
 2) "100"
 3) "b"
 4) "101"
 5) "c"
 6) "102"
```

```
 7) "d"
 8) "103"
 9) "e"
10) "104"
11) "f"
12) "105"
> ZREMRANGEBYSCORE sset11 101 104
(integer) 4
> ZRANGE sset11 0 10000 WITHSCORES
1) "a"
2) "100"
3) "f"
4) "105"
```

Increment/decrement score by some amount (ZINCRBY)

Add to or remove from the score of an existing item in the set.

```
> ZRANGE sset2 0 1000 WITHSCORES
1) "a"
2) "100"
3) "b"
4) "101"
> ZINCRBY sset2 b 100000
(error) ERR value is not a valid float
> ZINCRBY sset2 10000 b
"10101"
> ZRANGE sset2 0 10000000 WITHSCORES
1) "a"
2) "100"
3) "b"
4) "10101"
```

Get items within some range (ZRANGE/ZRANGEBYSCORE)

Get the items within some rank range (between a min index and max index, where index refers to the the position in the set). If you add WITHSCORES to the end, it'll return the score for each item as well.

> **NOTE**: Want to get get the items within some SCORE range instead? Use ZRANGEBYSCORE.
>
> **NOTE**: Want to get get the reverse? There's ZREVRANGE/ZREVRANGEBYSCORE.

> **NOTE**: There's also ZLEXRANGE/ZREVRANGEBYLEX, for getting which is the same thing but based on keys as the sort order instead of score? This is too confusing to write out, just read the documentation.

```
> ZRANGE sset2 0 10000000
1) "a"
2) "b"
> ZRANGE sset2 0 10000000 WITHSCORES
1) "a"
2) "100"
3) "b"
4) "10101"
```

Get number of items in a index/rank range (ZCOUNT)

Get the number of items in the set within some rank (between a min index and max index, where index refers to the the position in the set).

> **NOTE**: There's also ZLEXCOUNT? This is too confusing to write out, just read the documentation for ZLEXRANGE and then for ZLEXCOUNT.

```
> ZRANGE sset2 0 10000000
1) "a"
2) "b"
> ZCOUNT sset2 0 10000000
(integer) 2
```

Get score of item within set (ZSCORE)

Get score of an item within the set.

```
> ZRANGE sset2 0 10000000 WITHSCORES
1) "a"
2) "100"
3) "b"
4) "10101"
> ZSCORE sset2 a
"100"
> ZSCORE sset2 b
"10101"
```

Get index/rank of item within set (ZRANK)

Get the index of an item in the set (the rank of the item).

```
> ZRANGE sset2 0 10000000 WITHSCORES
1) "a"
2) "100"
3) "b"
4) "10101"
> ZRANK sset2 a
(integer) 0
> ZRANK sset2 b
(integer) 1
```

Get total size (ZCARD)

Get the total number of items in the set.

```
> ZRANGE sset2 0 10000000 WITHSCORES
1) "a"
2) "100"
3) "b"
4) "10101"
> ZCARD sset2
(integer) 2
```

Union/intersection between sorted sets (ZUNIONSTORE/ZINTERSTORE)

Performs a difference/union/intersection between 2 or more sorted sets and places the results in a new set.

Arguments for this are slightly confusing, it's…

1. destination set (where to store the results)
2. number of sets to perform the operation on (e.g. 2)
3. the names of the sets we're performing the operation on

So for example if we were unioning 4 sets together it would be: ZUNIONSTORE dst_zset 4 src_zset1 src_zset2 src_zset3 src_zset4.

```
> ZRANGE sset3 0 10000000
1) "a"
2) "b"
```

```
3) "c"
> ZRANGE sset4 0 10000000
1) "d"
2) "b"
> ZUNIONSTORE sset3_and_4 2 sset3 sset4
(integer) 4
> ZRANGE sset3_and_4 0 10000000
1) "a"
2) "d"
3) "b"
4) "c"
> ZINTERSTORE sset3_int_4 2 sset3 sset4
(integer) 2
> ZRANGE sset3_int_4 0 10000000
1) "b"
2) "c"
```

Iterate over sorted set (ZSCAN)

ZSCAN returns you entries from the sorted set in an iterative manner -- meaning you get things in chunks rather than getting everything at once. This is useful if you want to implement something like pagination.

> **NOTE**: Items returned by ZSCAN are not ordered.

The usage pattern for ZSCAN is ZSCAN key cursor [MATCH pattern] [COUNT count]

- key is the name of your sorted set.
- cursor is a value returned from a previous scan that you can pass in to continue scanning.
  (initial value is 0)
- [MATCH pattern] is an optional glob-style argument that will filter your results.
  (for example MATCH *Alabama*)
- [COUNT count] is an optional argument that will try to keep the results of a single scan to the size you specify.
  (for example COUNT 10)

ZSCAN is just like SSCAN, so look up SSCAN to see a usage example.

> **NOTE**: One thing to know is that the count value is just a hint to the implementation and can be ignored by the implementation. On a set with 100 elements, my count of 5 was ignored and the entire set was returned. This is known behaviour according to this: https://github.com/antirez/redis/issues/1723#issuecomment-42385837.

Time-to-live (TTL) / Expiry

In Redis, you can set your keys to automatically be removed after some amount of time has elapsed.

To set a key to expire, you can use any of the following commands:

- EXPIRE → set the key to expire after some number of seconds.
- PEXPIRE → set the key to expire after some number of milliseconds.
- EXPIREAT → set the key to expire at some Unix timestamp (in seconds).
- PEXPIREAT → set the key to expire at some Unix timestamp (in milliseconds).

To remove the expiry on a key, you can use the PERSIST command.

To check when a key expires, you can use any of the following commands:

- TTL → returns the number of seconds left before before the key expires.
- PTTL → returns the number of milliseconds left before the key expires.

The following example…

1. sets the key “test1”
2. marks the key “test1” to expire after 2 seconds
3. waits 1 second and gets the key
4. waits 1 more second and then gets the key again (should be expired at this point)

```
> set test1 "hihihihihi"
OK
> expire test1 2
(integer) 1
> get test1
"hihihihihi"
> get test1
(nil)
```

> **NOTE**: There is a bug/issue with key expiries and optimistic locking. See the section on transactions and optimistic locking for more information.

Publish/Subscribe

Redis can be used as a message bus between endpoints of your application. Unlike a normal redis list/queue, pubsub messages are …

- transient → aren’t stored or cached, and may not be delivered in certain cases

- broadcasted → every subscriber will receive a published message

The transient property seems like a negative, but on the upside it's a large part of what makes it fast.

> **NOTE**: Redis's clustering mechanism doesn't seem to handle pubsub very well. For each message that gets published, the current clustering architecture will broadcast it to every other node in the cluster, even if no clients are subscribed to that particular channel on a node.

Subscribing for Messages

To subscribe to receive messages, use the subscribe command. Here's an example that subscribes to the channel "channel1" and receives the messages "hello world" and "test test test test".

```
> subscribe channel1
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "channel1"
3) (integer) 1
1) "message"
2) "channel1"
3) "hello world"
1) "message"
2) "channel1"
3) "test test test test"
```

Publishing Messages

To publish a message, use the publish command. Here's an example that publishes to the channel "channel1" and sends 2 messages: "hello world" and "test test test test".

```
> publish channel1 "hello world"
(integer) 1
> publish channel1 "test test test test"
(integer) 1
```

Concurrency

You can group multiple commands that you want to run such that they run atomically. This is called a transaction. The difference between Redis transactions and transactions in a relational database are that …

- there are no transaction consistency levels in Redis. In Redis, you're just queueing up commands rather than performing database operations.
- there is no rolling back if a command in the transaction fails. In Redis, if you run a transaction and it fails on a queued command, any modifications made by previous commands in the transaction will still be applied.

  NOTE: Keep the 2 above points in mind, as they have serious downstream implications if you're using persistence. See the persistence section for more info.

Transactions

Here's how to use transactions in Redis

- start a transaction, you use the MULTI command.
- execute the transaction that you started, you use the EXEC command.
- discard the transaction that you started, you use the DISCARD command.

Here's an example of executing a simple transaction (from the docs)…

```
> MULTI
OK
> INCR foo
QUEUED
> INCR bar
QUEUED
> EXEC
1) (integer) 1
2) (integer) 1
```

Here's an example of discards a transaction (from the docs)...

```
> SET foo 1
OK
> MULTI
OK
> INCR foo
QUEUED
> DISCARD
```

```
OK
> GET foo
"1"
```

> **NOTE**: Transactions don't work at all if you're using clustering, unless you use what are called "clustering hashtags", which seem to be a way of forcing your keys to all be on the same node?

Optimistic Locking

There are no explicit locks in Redis. You can, however, use a WATCH command to make sure that the value for a key hasn't changed between the time you issue your watch / start your transaction and the time you execute your transaction.

So for example, I want to make sure that the key "test1" doesn't change from its initial value of while my transaction is being queued/executed.

I begin by getting the value for "test1"...

```
> get test1
"aaaa"
```

Then, I create the watch and get the value again to make sure that it didn't change...

```
> watch test1
OK
> get test1
"aaaa"
```

> **NOTE**: If the GET after the WATCH isn't watch you expect, you can UNWATCH the key and retry.

Then, I setup and perform my transaction as normal would.

```
> multi
OK
> set test2 ddddd
QUEUED
> exec
(nil)
```

Notice the last line says "(nil)". That means that the transaction wasn't executed because the WATCH we issued detected that the value for "test1" was changed by another client.

NOTE: In Redis, keys can have a TTL value. That means that you can set a key to be automatically removed after some amount of time has elapsed. If you WATCH a key but it expires before you EXEC, it will still execute (this is according to the docs --I haven't had a chance to test this yet).

NOTE: When EXEC is called, all keys are UNWATCHed, regardless of whether the transaction was aborted or not. Also when a client connection is closed, everything gets UNWATCHed. (from docs)

Scripting

Redis supports running scripts. A user can pass in a LUA script using the EVAL command and have it execute as a single atomic unit (just like transactions).

The upside is with this is that it gives you a good amount of flexibility in what you can do. For example, it's really easy to write new functionality such as conditional updates or removes. The downside is that you can write code that ends up being slow (e.g. heavy loops), causing the server to perform poorly. Generally, so long as your code is simple, LUA scripts execute fairly quickly.

```
> get test1
"hi"
> eval "return redis.call('get', KEYS[1])" 1 test1
"hi"
> eval "if (redis.call('get', KEYS[1]) == 'hi') then redis.call('set',
KEYS[1], 'bye') end" 1 test1
(nil)
> get test1
"bye"
```

The args for the EVAL function are confusing: EVAL script numkeys key [key ...] arg [arg ...]

1. script → This is the actual LUA script.
2. numkeys key [key …] → These are the keys that your script will be touching. It's important for Redis to know this because if you're using clustering, all the keys being touched need to end up being on the same node. Keys that you pass into your script can be read via the KEYS array (starting from array index 1, not 0). (See the section on clustering for more information on clustering).
3. arg [arg ...] → These are arbitrary arguments you can pass into your script. You can access these in your script via the ARGV array (starting from array index 1, not 0).

Persistence

Redis provides 2 types of persistence:

- RDB snapshots → snapshot of Redis at some point in time
- AOF → log of commands appended to the end of a file

If you need persistence, it's typical to enable both. RDB snapshots give you archivable backups while AOF can be used for more immediate recovery from a crash.

Neither of these persistence models are foolproof. In the event of a Redis crash, the persisted data will likely not be up-to-date with whatever your last executed Redis command was. This applies even if you're using Redis transaction.

> **NOTE**: Remember how Redis transactions work. The "transaction" isn't really a database-style transaction at all. There is no rolling back or explicit committing -- the commands are executed one at a time and any modifications made by a command are applied as soon as it runs. All a Redis "transaction" does is guarantee that the commands you queued up are all executed together. Because Redis does everything in a single threaded event-loop, that means no other commands can execute while your "transaction" is executing.

Redis Database (RDB) Snapshot

RDB snapshots are snapshots of the Redis's state that get persisted out to disk. That means that all your key/value pairs at some point in time will get written to disk. If Redis goes down and comes back up again, it reloads the RDB snapshot back into memory.

Redis continues operating normally while the snapshot is being written out to disk. You don't have to worry about race conditions where some key/value is modified prior to being written -- what gets written out is the data at the point in time which the RDB backup was triggered.

The way this works internally is that the Redis server process forks when the RDB snapshot is triggered. The fork is what writes out the RDB snapshot, while the main server process keeps servicing requests. Although the main process and the fork are pointing to the same data in memory, modifications made to key-value pairs by the main process won't affect the fork -- modifications are done in a copy-on-write fashion.

You can turn on RDB snapshotting in your configuration file via the save option...

```
# dump the dataset to disk every 900 seconds if at least 1 key changed
# you can have multiple save statements in the conf file
```

```
dbfilename dump.rdb
save 900 1
```

If you want to manually trigger RDB snapshotting, use the SAVE/BGSAVE command.

Append-only File (AOF)

This is typical of what most other databases use when persisting data. A “log” of commands are appended to a file as they come in. If the server crashes, the log can be replayed to restore state.

The problem with AOF is that writing to a file doesn’t necessarily mean persisting to disk. Data being written may be cached at various levels before actually being written out. You can configure Redis to fsync() the AOF at certain points (even after every key modification), but according to some sources even fsync() calls may not be respected…

> (from H2 DB docs)
> By default, MySQL calls fsync for each commit. When using one of those methods, only around 60 write operations per second can be achieved, which is consistent with the RPM rate of the hard drive used. Unfortunately, even when calling FileDescriptor.sync() or FileChannel.force(), data is not always persisted to the hard drive, because most hard drives do not obey fsync(): see Your Hard Drive Lies to You. In Mac OS X, fsync does not flush hard drive buffers. See Bad fsync?. So the situation is confusing, and tests prove there is a problem.

In any event, Redis provides the following options when configuring fsync for AOF:

- Always → call fsync() for any change (very slow)
- Per second → call fsync() every second
- Never → leave it up to the OS/drivers/etc.. to flush out the cached writes to disk

You can turn on AOF in your configuration file via the append* options...

```
appendonly yes
appendfilename "appendonly.aof"
appendfsync everysec # can also be set to always or no
```

Replication

Redis uses a master-slave replication model. A Redis server can have some number of slaves that it copies data to. By default, these slaves are read-only -- you can write data to only the master, but you can read data from either the slaves or the master.

Depending on what you're doing, reading data from the slaves may cause problems. Just like with persistence, there are consistency issues with replication. Replication between master and slaves is asynchronous. That means that when the user writes something to the master, the master responds with an OK before the data is sent out to the slaves. The slaves will be eventually consistent with the master, but you should not expect consistency between the data on your master server vs the data on your slave servers.

Doing things in transactions will not help with this. See the section on persistence for why this is.

To setup a Redis server as a slave of another server, use the slaveof configuration property…

```
# slaveof <masterip> <masterport>
slaveof masternode.mycompany.com 7000
```

> **NOTE**: I think I remember this being dependent on persistence?

Clustering

This part of Redis is vague, but here's how I think it works.

Clustering works by splitting the database into 16,384 different slots. Each node in the cluster is going to be responsible for some portion of that 16,384.

Unlike Cassandra, consistent hashing isn't used to determine which node a key sits on. Instead, the user manually specifies what slots a node is responsible for. Keys are hashed to determine the slot number, and then routed to the correct node. If a node is …

- added to the cluster, slots need to be manually moved to the new node.
- removed from the cluster, slots assigned to it need to be manually moved to other nodes BEFORE shutting down.

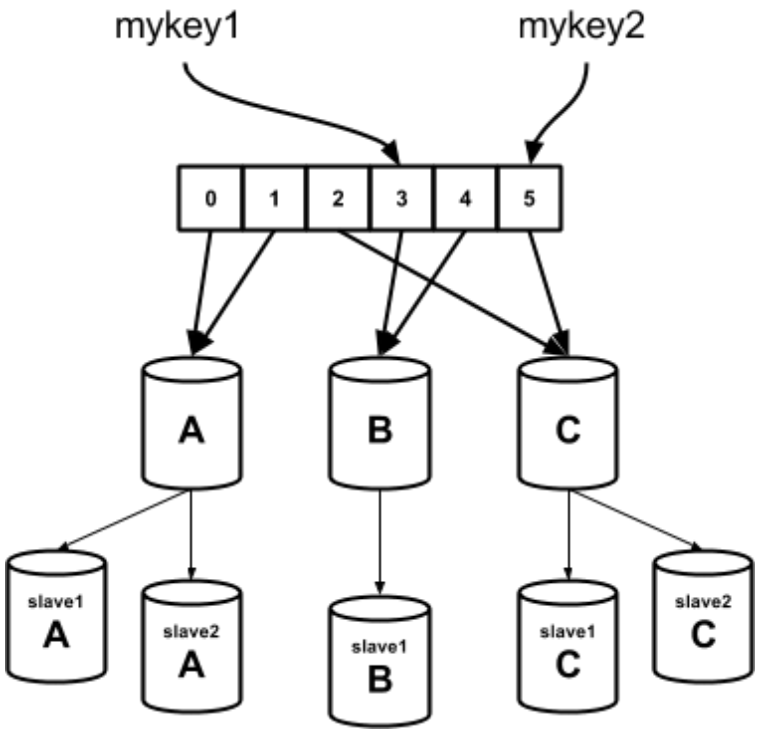
Unlike Cassandra, there is no quorum-style replication. Each node in the cluster can have slaves. If the node crashes, one of the slaves takes its place.

> **NOTE**: Just like with replication and persistence, you should have no expectation of consistency with clustering. See the section on replication and persistence to see why. Also, Redis transactions won't work if the keys queued up hit different nodes.

> **NOTE**: Clustering doesn't help if your key has a very large value. For example, if your key points to a set that has 10 billion members, that set will sit on a single server. It doesn't get distributed over the nodes in the cluster.

According to the docs, in the real world a Redis cluster can be scaled up to around 1000 master nodes, but you can technically reach up to 16,384 mast nodes (1 master node for each slot).

The following diagram should give a clear picture of how clustering works. For the sake of simplicity I've reduced the number of slots from 16,384 to 5. Keys get hashed to slots, and each slot is manually assigned to a node in the cluster. That's how you know which key goes to which node.



Transactions and Hashtags

Normal transactions won't work if you're using a cluster. The problem is that your transaction may have commands queued up with keys that direct to multiple different nodes. There's currently no way to lockdown multiple nodes and execute those commands sequentially.

> **NOTE**: It doesn't look like there ever will be away to lock down multiple nodes at once since it will likely kill performance.
>
> **NOTE**: This applies to LUA scripts as well. See the section on scripting for more information.

There is no workaround. You need to re-architect your database such that you use the new hash tags feature. Hash tags are when you only hash a portion of the key to figure out which slot to operate in. You use the { and } keys to specify that portion.…

```
> MULTI
OK
> SET {user5512A}.name "steve"
QUEUED
> SET {user5512A}.score "100"
QUEUED
> EXEC
1) (integer) 1
2) (integer) 1
```

You're technically operating on 2 different keys in this transaction: {user5512A}.name and {user5512A}.score. But, because the portion of the keys wrapped in { and } are the same, they're guaranteed to hash to the same slot. As such, this transaction will work fine.

> **NOTE**: You need to be consistent with your use of the hash tag feature. If you're going to use it, you need to use it everywhere -- even when you aren't using transactions. All your reads/writes need to end up pointing to the same slot.

If we didn't use { and }, the keys would very likely hash to different slots, meaning that there's no guarantee they'll end up on the same node.

Cluster Setup

> **NOTE**: The setup process is very vague. None of this has been tested -- it's just been pieced together from information I found while Googling. I'll tighten this up when if I ever have to use it.

It looks like nothing about setting up a cluster or managing it is automated in any way -- it's up to you as the user to…

- introduce nodes to each other.
- move slots onto new nodes.
- move slots out of nodes before shutting them down.
- redistribute slots if load is uneven across the cluster.

Also, the master-slave replication for clustering seems to be different than the normal master-slave replication talked about in the replication section.

Before you do anything, you need to enable clustering by changing the configuration file…

```
cluster-enabled yes
```

Once you do that, restart the server and you can issue a bunch of CLUSTER commands. But, it's unclear which commands are used for setup: https://redis.io/commands#cluster.

This stackoverflow post documents how to setup a single node cluster with a slave: https://stackoverflow.com/a/27397735/1196226. The python script it links to doesn't seem to work at (fails to install from pip).

Here's what I think you should do…

For the initial node, create it and issue the CLUSTER ADD SLOTS command that the stackoverflow post describes…

```
> CLUSTER ADD SLOTS 0 1 2 3 ... 16384
```

For the next node, create it and issue a CLUSTER MEET command with the initial node's IP and port…

```
> CLUSTER MEET masterA.mycompany.com 1234
```

Once the new node is in, you need to do slot migration (see the next section).

> **NOTE**: It doesn't look like it's enough to just call MEET. If the cluster is large, you need to pass in a few nodes and then wait for the new node to get the info for all the other nodes via Redis's gossiping protocol. I'm not sure at what point a node is expected to be fully a part of the cluster? Maybe the info from the CLUSTER INFO command can help?

Slot Migration

> **NOTE**: Just like the setup steps, this is also vague. None of this has been tested.

To migrate a slot from one node to another, you need to use the CLUSTER SETSLOT command on both nodes. The usage is slightly different from the sending node vs the receiving node.

On the receiving node...

```
> CLUSTER SETSLOT 5 IMPORTING src-node-id
```

On the sending node...

```
> CLUSTER SETSLOT 5 MIGRATING dst-node-id
```

Once you do that, you need to manually start iterating over the items in the slot and moving them over. This isn't something that automatically happens -- you do it yourself.

Read items using CLUSTER GETKEYSINSLOT and migrate them over using CLUSTER MIGRATE…

```
> CLUSTER GETKEYSINSLOT 5 3
"47344|273766|70329104160040|key1"
"47344|273766|70329104160040|key2"
"47344|273766|70329104160040|key3"
> MIGRATE dst-node-host dst-node-port "" 0 5000 KEYS key1 key2 key3
```

Keep doing this until you run out of keys to migrate, then issue a CLUSTER SETSLOT <slot> NODE <destination-node-id> command on either the sender or receiver to finalize…

```
> CLUSTER SETSLOT 5 NODE dst-node-id
```