An introduction to Redis data types and abstractions

Redis is not a *plain* key-value store, it is actually a *data structures server*, supporting different kinds of values. What this means is that, while in traditional key-value stores you associated string keys to string values, in Redis the value is not limited to a simple string, but can also hold more complex data structures. The following is the list of all the data structures supported by Redis, which will be covered separately in this tutorial:

- Binary-safe strings.
- Lists: collections of string elements sorted according to the order of insertion. They are basically *linked lists*.
- Sets: collections of unique, unsorted string elements.
- Sorted sets, similar to Sets but where every string element is associated to a floating number value, called *score*. The elements are always taken sorted by their score, so unlike Sets it is possible to retrieve a range of elements (for example you may ask: give me the top 10, or the bottom 10).
- Hashes, which are maps composed of fields associated with values. Both the field and the value are strings. This is very similar to Ruby or Python hashes.
- Bit arrays (or simply bitmaps): it is possible, using special commands, to handle String values like an array of bits: you can set and clear individual bits, count all the bits set to 1, find the first set or unset bit, and so forth.
- HyperLogLogs: this is a probabilistic data structure which is used in order to estimate the cardinality of a set. Don't be scared, it is simpler than it seems... See later in the HyperLogLog section of this tutorial.

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 common patterns.

For all the examples we'll use the `redis-cli` utility, 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:

- Very long keys are not a good idea. For instance a key of 1024 bytes is a bad idea not only memory-wise, but also because the lookup of the key in the dataset may require several costly key-comparisons. Even when the task at hand is to match the existence of a large value, hashing it (for example with SHA1) is a better idea, especially from the perspective of memory and bandwidth.
- Very short keys are often not a good idea. There is little point in writing "u1000flw" as a key if you can instead write "user:1000:followers". The latter is more readable and the added space is minor compared to the space used by the key object itself and the value object. While short keys will obviously consume a bit less memory, your job is to find the right balance.
- Try to stick with a schema. For instance "object-type:id" is a good idea, as in "user:1000". Dots or dashes are often used for multi-word fields, as in "comment:1234:reply.to" or "comment:1234:reply-to".
- The maximum allowed key size is 512 MB.

Redis Strings

The Redis String type is the simplest type of value you can associate with a Redis key. It is the only data type in Memcached, so it is also very natural for newcomers to use it in Redis.

Since Redis keys are strings, when we use the string type as a value too, we are mapping a string to another string. The string data type is useful for a number of use cases, like caching HTML fragments or pages.

Let's play a bit with the string type, using `redis-cli` (all the examples will be performed via `redis-cli` in this tutorial).

```
> set mykey somevalue
OK
> get mykey
"somevalue"
```

As you can see using the SET and the GET commands are the way we set and retrieve a string value. Note that SET will replace any existing value already stored into the key, in the case that the key already exists, even if the key is associated with a non-string value. So SET performs an assignment.

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

The SET command has interesting options, that are provided as additional arguments. For example, I may ask SET to fail if the key already exists, or the opposite, that it only succeed if the key already exists:

```
> set mykey newval nx
(nil)
> set mykey newval xx
OK
```

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

```
> set counter 100
OK
> incr counter
(integer) 101
> incr counter
(integer) 102
> incrby counter 50
(integer) 152
```

The INCR command parses the string value as an integer, increments it by one, and finally sets the obtained value as the new 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 enter into a race condition. For instance, it will never happen that client 1 reads "10", client 2 reads "10" at the same time, both increment to 11, and set the new value to 11. The final value will always be 12 and the read-increment-set operation is performed while all the other clients are not executing a command at the same time.

There are a number of commands for operating on strings. For example the GETSET command sets a key to a new value, returning the old value as the result. You can use this command, for example, if you have a system that increments a Redis key using INCR every time your web site receives a new visitor. You may want to collect this information once every hour, without losing a single increment. You can GETSET the key, assigning it the new value of "0" and reading the old value back.

The ability to set or retrieve the value of multiple keys in a single command is also useful for reduced latency. For this reason there are the MSET and MGET commands:

```
> mset a 10 b 20 c 30
OK
> mget a b c
1) "10"
2) "20"
3) "30"
```

When MGET is used, Redis returns an array of values.

Altering and querying the key space

There are commands that are not defined on particular types, but are useful in order to interact with the space of keys, and thus, can be used with keys of any type.

For example the EXISTS command returns 1 or 0 to signal if a given key exists or not in the database, while the DEL command deletes a key and associated value, whatever the value is.

```
> set mykey hello
OK
> exists mykey
(integer) 1
> del mykey
(integer) 1
> exists mykey
(integer) 0
```

From the examples you can also see how DEL itself returns 1 or 0 depending on whether the key was removed (it existed) or not (there was no such key with that name).

There are many key space related commands, but the above two are the essential ones together with the TYPE command, which returns the kind of value stored at the specified key:

```
> set mykey x
OK
> type mykey
string
> del mykey
(integer) 1
> type mykey
none
```

Redis expires: keys with limited time to live

Before continuing with more complex data structures, we need to discuss another feature which works regardless of the value type, and is called **Redis expires**. Basically you can set a timeout for a key, which is a limited time to

live. When the time to live elapses, the key is automatically destroyed, exactly as if the user called the DEL command with the key.

A few quick info about Redis expires:

- They can be set both using seconds or milliseconds precision.
- However the expire time resolution is always 1 millisecond.
- Information about expires are replicated and persisted on disk, the time virtually passes when your Redis server remains stopped (this means that Redis saves the date at which a key will expire).

Setting an expire is trivial:

```
> set key some-value
OK
> expire key 5
(integer) 1
> get key (immediately)
"some-value"
> get key (after some time)
(nil)
```

The key vanished between the two GET calls, since the second call was delayed more than 5 seconds. In the example above we used EXPIRE in order to set the expire (it can also be used in order to set a different expire to a key already having one, like PERSIST can be used in order to remove the expire and make the key persistent forever). However we can also create keys with expires using other Redis commands. For example using SET options:

```
> set key 100 ex 10
OK
> ttl key
(integer) 9
```

The example above sets a key with the string value `100`, having an expire of ten seconds. Later the TTL command is called in order to check the remaining time to live for the key.

In order to set and check expires in milliseconds, check the PEXPIRE and the PTTL commands, and the full list of SET options.

Redis Lists

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), but rather 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*. The speed of adding a new element with the LPUSH command to the head of a list with ten elements is the same as adding an element to the head of list with 10 million elements.

What's the downside? Accessing an element *by index* is very fast in lists implemented with an Array (constant time indexed access) and not so fast in lists implemented by linked lists (where the operation requires an amount of work proportional to the index of the accessed element).

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

When fast access to the middle of a large collection of elements is important, there is a different data structure that can be used, called sorted sets. Sorted sets will be covered later in this tutorial.

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:

```
> rpush mylist A
(integer) 1
> rpush mylist B
(integer) 2
> lpush mylist first
(integer) 3
> lrange mylist 0 -1
1) "first"
2) "A"
3) "B"
```

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

As you can see RPUSH appended the elements on the right of the list, while the final LPUSH appended the element on the left.

Both commands are *variadic commands*, meaning that you are free to push multiple elements into a list in a single call:

```
> rpush mylist 1 2 3 4 5 "foo bar"
(integer) 9
> lrange mylist 0 -1
1) "first"
2) "A"
3) "B"
4) "1"
5) "2"
6) "3"
7) "4"
8) "5"
9) "foo bar"
```

An important operation defined on Redis lists is the ability to *pop elements*. Popping elements is the operation of both retrieving the element from the list, and eliminating it from the list, at the same time. You can pop elements from left and right, similarly to how you can push elements in both sides of the list:

```
> rpush mylist a b c
(integer) 3
> rpop mylist
"c"
> rpop mylist
"b"
> rpop mylist
"a"
```

We added three elements and popped three elements, so at the end of this sequence of commands the list is empty and there are no more elements to pop. If we try to pop yet another element, this is the result we get:

```
> rpop mylist
(nil)
```

Redis returned a NULL value to signal that there are no elements in the list.

Common use cases for lists

Lists are useful for a number of tasks, two very representative use cases are the following:

- Remember the latest updates posted by users into a social network.
- Communication between processes, using a consumer-producer pattern where the producer pushes items into a list, and a consumer (usually a *worker*) consumes those items and executed actions. Redis has special list commands to make this use case both more reliable and efficient.

For example both the popular Ruby libraries resque and sidekiq use Redis lists under the hood in order to implement background jobs.

The popular Twitter social network takes the latest tweets posted by users into Redis lists.

To describe a common use case step by step, imagine your home page shows the latest photos published in a photo sharing social network and you want to speedup access.

- Every time a user posts a new photo, we add its ID into a list with LPUSH.
- When users visit the home page, we use `LRANGE 0 9` in order to get the latest 10 posted items.

Capped lists

In many use cases we just want to use lists to store the *latest items*, whatever they are: social network updates, logs, or anything else.

Redis allows us to use lists as a capped collection, only remembering the latest N items and discarding all the oldest items using the LTRIM command.

The LTRIM command is similar to LRANGE, but **instead of displaying the specified range of elements** it sets this range as the new list value. All the elements outside the given range are removed.

An example will make it more clear:

```
> rpush mylist 1 2 3 4 5
(integer) 5
> ltrim mylist 0 2
OK
> lrange mylist 0 -1
1) "1"
2) "2"
3) "3"
```

The above LTRIM command tells Redis to take just list elements from index 0 to 2, everything else will be discarded. This allows for a very simple but useful pattern: doing a List push operation + a List trim operation together in order to add a new element and discard elements exceeding a limit:

```
LPUSH mylist <some element>
LTRIM mylist 0 999
```

The above combination adds a new element and takes only the 1000 newest elements into the list. With LRANGE you can access the top items without any need to remember very old data.

Note: while LRANGE is technically an O(N) command, accessing small ranges towards the head or the tail of the list is a constant time operation.

Blocking operations on lists

Lists have a special feature that make them suitable to implement queues, and in general as a building block for inter process communication systems: blocking operations.

Imagine you want to push items into a list with one process, and use a different process in order to actually do some kind of work with those items. This is the usual producer / consumer setup, and can be implemented in the following simple way:

- To push items into the list, producers call LPUSH.
- To extract / process items from the list, consumers call RPOP.

However it is possible that sometimes the list is empty and there is nothing to process, so RPOP just returns NULL. In this case a consumer is forced to wait some time and retry again with RPOP. This is called *polling*, and is not a good idea in this context because it has several drawbacks:

1. Forces Redis and clients to process useless commands (all the requests when the list is empty will get no actual work done, they'll just return NULL).
2. Adds a delay to the processing of items, since after a worker receives a NULL, it waits some time. To make the delay smaller, we could wait less between calls to RPOP, with the effect of amplifying problem number 1, i.e. more useless calls to Redis.

So Redis implements commands called BRPOP and BLPOP which are versions of RPOP and LPOP able to block if the list is empty: they'll return to the caller only when a new element is added to the list, or when a user-specified timeout is reached.

This is an example of a BRPOP call we could use in the worker:

```
> brpop tasks 5
1) "tasks"
2) "do_something"
```

It means: "wait for elements in the list `tasks`, but return if after 5 seconds no element is available".

Note that you can use 0 as timeout to wait for elements forever, and you can also specify multiple lists and not just one, in order to wait on multiple lists at the same time, and get notified when the first list receives an element.

A few things to note about BRPOP:

1. Clients are served in an ordered way: the first client that blocked waiting for a list, is served first when an element is pushed by some other client, and so forth.
2. The return value is different compared to RPOP: it is a two-element array since it also includes the name of the key, because BRPOP and BLPOP are able to block waiting for elements from multiple lists.
3. If the timeout is reached, NULL is returned.

There are more things you should know about lists and blocking ops. We suggest that you read more on the following:

- It is possible to build safer queues or rotating queues using RPOPLPUSH.
- There is also a blocking variant of the command, called BRPOPLPUSH.

Automatic creation and removal of keys

So far in our examples we never had to create empty lists before pushing elements, or removing empty lists when they no longer have elements inside. It is Redis' responsibility to delete keys when lists are left empty, or to create an empty list if the key does not exist and we are trying to add elements to it, for example, with LPUSH.

This is not specific to lists, it applies to all the Redis data types composed of multiple elements -- Sets, Sorted Sets and Hashes.

Basically we can summarize the behavior with three rules:

1. When we add an element to an aggregate data type, if the target key does not exist, an empty aggregate data type is created before adding the element.
2. When we remove elements from an aggregate data type, if the value remains empty, the key is automatically destroyed.
3. Calling a read-only command such as LLEN (which returns the length of the list), or a write command removing elements, with an empty key, always produces the same result as if the key is holding an empty aggregate type of the type the command expects to find.

Examples of rule 1:

```
> del mylist
(integer) 1
> lpush mylist 1 2 3
(integer) 3
```

However we can't perform operations against the wrong type if the key exists:

```
> set foo bar
OK
> lpush foo 1 2 3
(error) WRONGTYPE Operation against a key holding the wrong kind of value
> type foo
string
```

Example of rule 2:

```
> lpush mylist 1 2 3
(integer) 3
> exists mylist
(integer) 1
> lpop mylist
"3"
> lpop mylist
"2"
> lpop mylist
"1"
> exists mylist
(integer) 0
```

The key no longer exists after all the elements are popped.

Example of rule 3:

```
> del mylist
(integer) 0
> llen mylist
(integer) 0
> lpop mylist
(nil)
```

Redis Hashes

Redis hashes look exactly how one might expect a "hash" to look, with field-value pairs:

```
> hmset user:1000 username antirez birthyear 1977 verified 1
OK
> hget user:1000 username
"antirez"
> hget user:1000 birthyear
"1977"
> hgetall user:1000
1) "username"
2) "antirez"
3) "birthyear"
4) "1977"
5) "verified"
6) "1"
```

While hashes are handy to represent *objects*, actually the number of fields you can put inside a hash has no practical limits (other than available memory), so you can use hashes in many different ways inside your application.

The command HMSET sets multiple fields of the hash, while HGET retrieves a single field. HMGET is similar to HGET but returns an array of values:

```
> hmget user:1000 username birthyear no-such-field
1) "antirez"
2) "1977"
3) (nil)
```

There are commands that are able to perform operations on individual fields as well, like HINCRBY:

```
> hincrby user:1000 birthyear 10
(integer) 1987
> hincrby user:1000 birthyear 10
(integer) 1997
```

You can find the full list of hash commands in the documentation.

It is worth noting that small hashes (i.e., a few elements with small values) are encoded in special way in memory that make them very memory efficient.

Redis Sets

Redis Sets are unordered collections of strings. The SADD command adds new elements 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.

```
> sadd myset 1 2 3
(integer) 3
> smembers myset
1. 3
2. 1
3. 2
```

Here I've added three elements to my set and told Redis to return all the elements. As you can see they are not sorted -- Redis is free to return the elements in any order at every call, since there is no contract with the user about element ordering.

Redis has commands to test for membership. For example, checking if an element exists:

```
> sismember myset 3
(integer) 1
> sismember myset 30
(integer) 0
```

"3" is a member of the set, while "30" is not.

Sets are good for expressing relations between objects. For instance we can easily use sets in order to implement tags.

A simple way to model this problem is to have a set for every object we want to tag. The set contains the IDs of the tags associated with the object.

One illustration is tagging news articles. If article ID 1000 is tagged with tags 1, 2, 5 and 77, a set can associate these tag IDs with the news item:

```
> sadd news:1000:tags 1 2 5 77
(integer) 4
```

We may also want to have the inverse relation as well: the list of all the news tagged with a given tag:

```
> sadd tag:1:news 1000
(integer) 1
> sadd tag:2:news 1000
(integer) 1
> sadd tag:5:news 1000
(integer) 1
> sadd tag:77:news 1000
(integer) 1
```

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

```
> smembers news:1000:tags
1. 5
2. 1
3. 77
4. 2
```

Note: in the example we assume you have another data structure, for example a Redis hash, which maps tag IDs to tag names.

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 command, which performs the intersection between different sets. We can use:

```
> sinter tag:1:news tag:2:news tag:10:news tag:27:news
... results here ...
```

In addition to intersection you can also perform unions, difference, extract a random element, and so forth.

The command to extract an element is called SPOP, and is handy to model certain problems. For example in order to implement a web-based poker game, you may want to represent your deck with a set. Imagine we use a one-char prefix for (C)lubs, (D)iamonds, (H)earts, (S)pades:

```
>  sadd deck C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 CJ CQ CK
   D1 D2 D3 D4 D5 D6 D7 D8 D9 D10 DJ DQ DK H1 H2 H3
   H4 H5 H6 H7 H8 H9 H10 HJ HQ HK S1 S2 S3 S4 S5 S6
   S7 S8 S9 S10 SJ SQ SK
   (integer) 52
```

Now we want to provide each player with 5 cards. The SPOP command removes a random element, returning it to the client, so it is the perfect operation in this case.

However if we call it against our deck directly, in the next play of the game we'll need to populate the deck of cards again, which may not be ideal. So to start, we can make a copy of the set stored in the `deck` key into the `game:1:deck` key.

This is accomplished using SUNIONSTORE, which normally performs the union between multiple sets, and stores the result into another set. However, since the union of a single set is itself, I can copy my deck with:

```
> sunionstore game:1:deck deck
(integer) 52
```

Now I'm ready to provide the first player with five cards:

```
> spop game:1:deck
"C6"
> spop game:1:deck
"CQ"
> spop game:1:deck
"D1"
> spop game:1:deck
"CJ"
> spop game:1:deck
"SJ"
```

One pair of jacks, not great...

This is a good time to introduce the set command that provides the number of elements inside a set. This is often called the *cardinality of a set* in the context of set theory, so the Redis command is called SCARD.

```
> scard game:1:deck
(integer) 47
```

The math works: 52 - 5 = 47.

When you need to just get random elements without removing them from the set, there is the SRANDMEMBER command suitable for the task. It also features the ability to return both repeating and non-repeating elements.

Redis Sorted sets

Sorted sets are a data type which is similar to a mix between a Set and a Hash. Like sets, sorted sets are composed of unique, non-repeating string elements, so in some sense a sorted set is a set as well.

However while elements inside sets are not ordered, every element in a sorted set is associated with a floating point value, called *the score* (this is why the type is also similar to a hash, since every element is mapped to a value).

Moreover, elements in a sorted sets are *taken in order* (so they are not ordered on request, order is a peculiarity of the data structure used to represent sorted sets). They are ordered according to the following rule:

- If A and B are two elements with a different score, then A > B if A.score is > B.score.
- If A and B have exactly the same score, then A > B if the A string is lexicographically greater than the B string. A and B strings can't be equal since sorted sets only have unique elements.

Let's start with a simple example, adding a few selected hackers names as sorted set elements, with their year of birth as "score".

```
> zadd hackers 1940 "Alan Kay"
(integer) 1
> zadd hackers 1957 "Sophie Wilson"
(integer) 1
> zadd hackers 1953 "Richard Stallman"
(integer) 1
> zadd hackers 1949 "Anita Borg"
(integer) 1
> zadd hackers 1965 "Yukihiro Matsumoto"
(integer) 1
> zadd hackers 1914 "Hedy Lamarr"
(integer) 1
> zadd hackers 1916 "Claude Shannon"
(integer) 1
> zadd hackers 1969 "Linus Torvalds"
(integer) 1
> zadd hackers 1912 "Alan Turing"
(integer) 1
```

As you can see ZADD is similar to SADD, but takes one additional argument (placed before the element to be added) which is the score. ZADD is also variadic, so you are free to specify multiple score-value pairs, even if this is not used in the example above.

With sorted sets it is trivial to return a list of hackers sorted by their birth year because actually *they are already sorted*.

Implementation note: 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:

```
> zrange hackers 0 -1
1) "Alan Turing"
2) "Hedy Lamarr"
3) "Claude Shannon"
4) "Alan Kay"
5) "Anita Borg"
6) "Richard Stallman"
7) "Sophie Wilson"
8) "Yukihiro Matsumoto"
9) "Linus Torvalds"
```

Note: 0 and -1 means from element index 0 to the last element (-1 works here just as it does in the case of the LRANGE command).

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

```
> zrevrange hackers 0 -1
1) "Linus Torvalds"
2) "Yukihiro Matsumoto"
3) "Sophie Wilson"
4) "Richard Stallman"
5) "Anita Borg"
6) "Alan Kay"
7) "Claude Shannon"
8) "Hedy Lamarr"
9) "Alan Turing"
```

It is possible to return scores as well, using the `WITHSCORES` argument:

```
> zrange hackers 0 -1 withscores
1) "Alan Turing"
2) "1912"
3) "Hedy Lamarr"
4) "1914"
5) "Claude Shannon"
6) "1916"
7) "Alan Kay"
8) "1940"
9) "Anita Borg"
10) "1949"
11) "Richard Stallman"
12) "1953"
13) "Sophie Wilson"
14) "1957"
15) "Yukihiro Matsumoto"
16) "1965"
17) "Linus Torvalds"
18) "1969"
```

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 1950 inclusive. We use the ZRANGEBYSCORE command to do it:

```
> zrangebyscore hackers -inf 1950
1) "Alan Turing"
2) "Hedy Lamarr"
3) "Claude Shannon"
4) "Alan Kay"
5) "Anita Borg"
```

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:

```
> zremrangebyscore hackers 1940 1960
(integer) 4
```

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

Another extremely useful operation defined for sorted set elements is the get-rank operation. It is possible to ask what is the position of an element in the set of the ordered elements.

```
> zrank hackers "Anita Borg"
(integer) 4
```

The ZREVRANK command is also available in order to get the rank, considering the elements sorted a descending way.

Lexicographical scores

With recent versions of Redis 2.8, a new feature was introduced that allows getting ranges lexicographically, assuming elements in a sorted set are all inserted with the same identical score (elements are compared with the C `memcmp` function, so it is guaranteed that there is no collation, and every Redis instance will reply with the same output).

The main commands to operate with lexicographical ranges are ZRANGEBYLEX, ZREVRANGEBYLEX, ZREMRANGEBYLEX and ZLEXCOUNT.

For example, let's add again our list of famous hackers, but this time use a score of zero for all the elements:

```
> zadd hackers 0 "Alan Kay" 0 "Sophie Wilson" 0 "Richard Stallman" 0
  "Anita Borg" 0 "Yukihiro Matsumoto" 0 "Hedy Lamarr" 0 "Claude Shannon"
  0 "Linus Torvalds" 0 "Alan Turing"
```

Because of the sorted sets ordering rules, they are already sorted lexicographically:

```
> zrange hackers 0 -1
1) "Alan Kay"
2) "Alan Turing"
3) "Anita Borg"
4) "Claude Shannon"
5) "Hedy Lamarr"
6) "Linus Torvalds"
7) "Richard Stallman"
8) "Sophie Wilson"
9) "Yukihiro Matsumoto"
```

Using ZRANGEBYLEX we can ask for lexicographical ranges:

```
> zrangebylex hackers [B [P
1) "Claude Shannon"
2) "Hedy Lamarr"
3) "Linus Torvalds"
```

Ranges can be inclusive or exclusive (depending on the first character), also string infinite and minus infinite are specified respectively with the + and − strings. See the documentation for more information.

This feature is important because it allows us to use sorted sets as a generic index. For example, if you want to index elements by a 128-bit unsigned integer argument, all you need to do is to add elements into a sorted set with the same score (for example 0) but with an 16 byte prefix consisting of **the 128 bit number in big endian**. Since numbers in big endian, when ordered lexicographically (in raw bytes order) are actually ordered numerically as well, you can ask for ranges in the 128 bit space, and get the element's value discarding the prefix.

If you want to see the feature in the context of a more serious demo, check the Redis autocomplete demo.

Updating the score: leader boards

Just a final note about sorted sets before switching to the next topic. Sorted sets' scores can be updated at any time. Just calling ZADD against an element already included in the sorted set will update its score (and position) with O(log(N)) time complexity. As such, sorted sets are suitable when there are tons of updates.

Because of this characteristic a common use case is leader boards. The typical application is a Facebook game where you combine the ability to take users sorted by their high score, plus the get-rank operation, in order to show the top-N users, and the user rank in the leader board (e.g., "you are the #4932 best score here").

Bitmaps

Bitmaps are not an actual data type, but a set of bit-oriented operations defined on the String type. Since strings are binary safe blobs and their maximum length is 512 MB, they are suitable to set up to 2^{32} different bits.

Bit operations are divided into two groups: constant-time single bit operations, like setting a bit to 1 or 0, or getting its value, and operations on groups of bits, for example counting the number of set bits in a given range of bits (e.g., population counting).

One of the biggest advantages of bitmaps is that they often provide extreme space savings when storing information. For example in a system where different users are represented by incremental user IDs, it is possible to remember a single bit information (for example, knowing whether a user wants to receive a newsletter) of 4 billion of users using just 512 MB of memory.

Bits are set and retrieved using the SETBIT and GETBIT commands:

```
> setbit key 10 1
(integer) 1
> getbit key 10
(integer) 1
> getbit key 11
(integer) 0
```

The SETBIT command takes as its first argument the bit number, and as its second argument the value to set the bit to, which is 1 or 0. The command automatically enlarges the string if the addressed bit is outside the current string length.

GETBIT just returns the value of the bit at the specified index. Out of range bits (addressing a bit that is outside the length of the string stored into the target key) are always considered to be zero.

There are three commands operating on group of bits:

1. BITOP performs bit-wise operations between different strings. The provided operations are AND, OR, XOR and NOT.
2. BITCOUNT performs population counting, reporting the number of bits set to 1.
3. BITPOS finds the first bit having the specified value of 0 or 1.

Both BITPOS and BITCOUNT are able to operate with byte ranges of the string, instead of running for the whole length of the string. The following is a trivial example of BITCOUNT call:

```
> setbit key 0 1
(integer) 0
> setbit key 100 1
(integer) 0
> bitcount key
(integer) 2
```

Common use cases for bitmaps are:

- Real time analytics of all kinds.
- Storing space efficient but high performance boolean information associated with object IDs.

For example imagine you want to know the longest streak of daily visits of your web site users. You start counting days starting from zero, that is the day you made your web site public, and set a bit with SETBIT every time the user visits the web site. As a bit index you simply take the current unix time, subtract the initial offset, and divide by 3600*24.

This way for each user you have a small string containing the visit information for each day. With BITCOUNT it is possible to easily get the number of days a given user visited the web site, while with a few BITPOS calls, or simply fetching and analyzing the bitmap client-side, it is possible to easily compute the longest streak.

Bitmaps are trivial to split into multiple keys, for example for the sake of sharding the data set and because in general it is better to avoid working with huge keys. To split a bitmap across different keys instead of setting all the bits into a key, a trivial strategy is just to store M bits per key and obtain the key name with `bit-number/M` and the Nth bit to address inside the key with `bit-number MOD M`.

HyperLogLogs

A HyperLogLog is a probabilistic data structure used in order to count unique things (technically this is referred to estimating the cardinality of a set). Usually counting unique items requires using an amount of memory proportional to the number of items you want to count, because you need to remember the elements you have already seen in the past in order to avoid counting them multiple times. However there is a set of algorithms that trade memory for precision: you end with an estimated measure with a standard error, which in the case of the Redis implementation is less than 1%. The magic of this algorithm is that you no longer need to use an amount of memory proportional to the number of items counted, and instead can use a constant amount of memory! 12k bytes in the worst case, or a lot less if your HyperLogLog (We'll just call them HLL from now) has seen very few elements.

HLLs in Redis, while technically a different data structure, are encoded as a Redis string, so you can call GET to serialize a HLL, and SET to deserialize it back to the server.

Conceptually the HLL API is like using Sets to do the same task. You would SADD every observed element into a set, and would use SCARD to check the number of elements inside the set, which are unique since SADD will not re-add an existing element.

While you don't really *add items* into an HLL, because the data structure only contains a state that does not include actual elements, the API is the same:

- Every time you see a new element, you add it to the count with PFADD.
- Every time you want to retrieve the current approximation of the unique elements *added* with PFADD so far, you use the PFCOUNT.

```
> pfadd hll a b c d
(integer) 1
> pfcount hll
(integer) 4
```

An example of use case for this data structure is counting unique queries performed by users in a search form every day.

Redis is also able to perform the union of HLLs, please check the full documentation for more information.

Other notable features

There are other important things in the Redis API that can't be explored in the context of this document, but are worth your attention:

- It is possible to iterate the key space of a large collection incrementally.
- It is possible to run Lua scripts server side to improve latency and bandwidth.
- Redis is also a Pub-Sub server.

Learn more

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

Thanks for reading, and have fun hacking with Redis!