CS 145: NoSQL Activity

Stanford University, Fall 2015

A Quick Introdution to Redis

For this assignment, compile your answers on a separate pdf to submit and verify that they work using Redis.

Installing Redis

Windows

1. Visit https://github.com/rgl/redis/downloads in your web browser and download the installer for Redis. At the time of this writing, the filename of the installer you should look for is `redis-2.4.6-setup-32-bit.exe`.
2. In the File Explorer, go to C:\Program Files\Redis (or whatever directory you ended up installing Redis into) and double-click on `redis-server.exe` to start the server.
3. In the same folder, double-click on `redis-cli.exe` to start a client terminal window. This is where you'll be executing your commands during the activity.

Mac/Linux

1. Open a terminal window, and download Redis by running
   `curl http://download.redis.io/redis-stable.tar.gz`
2. Extract Redis into a folder by running
   `tar xzf redis-stable.tar.gz`
3. Go into the folder you just extracted by running
   `cd redis-stable`
4. Build Redis by running
   `make`
5. Start the Redis server by running
   `src/redis-server`
6. In a new terminal window, go to the same folder and start a Redis client by running
   `src/redis-cli`
   This is where you'll be executing your commands during the activity.

Check your installation

Finally, make sure Redis is working properly. In the client interface for Redis (which you should have opened in the last step of installation), type the following and make sure you get the output shown below.

```
SET foo bar
=> OK
GET foo
=> "bar"
```

Activity Data

Load the Sample Database

1. Download the text file from http://web.stanford.edu/class/cs145/activities/RedisActivityData.txt. Save it anywhere, and open it in a text editor.
2. In your Redis client window, simply copy and paste all the commands in the file into the client terminal. (Rather than doing CTRL+V or Cmd-V, you may need to right-click and select "Paste".)

What the Sample Database Contains

- Several **hashes** identified by the key *user:$userID* with *username*, *age* and *city* fields
- A **hash** identified by the key *users* with fields *username* and the corresponding userID.
- A key *next_user_id* that keeps track of the user ID that should be assigned to the next new user. Right now, it is set to 10.

Try doing a couple of `GET` (for single values) and `HGETALL` (for hashes) commands to make sure you can see these items in your local copy of the database. (Try `HGETALL users`, for example. You should get 10 usernames and 10 ids.)

Problem 1: Inserting and Deleting

First, let's try inserting a new user into the database. In our Redis database, like in SQL, each user has a unique numeric identifier, which we'll call $userID. However, unlike SQL, Redis does not have a way of automatically incrementing this ID for new users that get created. Instead, we create a key to keep track of user IDs for us, and increment that whenever we create a new user.

Let's check what ID was assigned to the last user that was created using the `GET` command:

```
GET next_user_id
=> 10
```

To increment this value, we can simply use the `INCR` command:

```
INCR next_user_id
=> 11
```

which increases the value of next_user_id by 1 and tells us what user ID we should assign our new user. Note that this has to be done manually — Redis will not do this for you as your add new users.

Now that we know this, we can create the hash representing our user using the `HMSET` command, which allows us to create or update a hash we have in our database:

```
HMSET user:11 username "johnsmith" age 24 city "Stanford"
=> OK
```

This creates a hash identified by the key *user:11* with the fields *username*, *age* and *city* and the values "johnsmith", "24" and "Stanford", respectively. You can think of each hash as almost like a tuple or a row in a regular SQL table. It's just that in Redis, there isn't a larger construct like a table that links all these rows together. Instead, we make it

clear through the way we name our keys — *user:11* represents a user because of the naming convention we're using to identify hashes that represent users. All hashes representing users, in our design, will be of the form *user:$userID*.

Wait, there's one problem! There's a *users* hash that keeps track of all usernames and the corresponding user ID for that user. To see what it contains, type:

```
HGETALL users
=>  1) "jill"
    2) "1"
    3) "kirito"
    4) "2"
    5) "sara"
    6) "3"
    7) "ann"
    8) "4"
    9) "joy"
    10) "5"
    11) "firas"
    12) "6"
    13) "matt"
    14) "7"
    15) "bob"
    16) "8"
    17) "grace"
    18) "9"
    19) "phil"
    20) "10"
```

(Despite the way the output looks, this is not just a really big array – the data is actually stored in pairs. "jill" holds the value "1", "kirito" holds the value "2", and so on.)

If you know there's a user named "bob", you can type `HGET users bob`, which will return 8, the userID for bob. Then, once you've retrieved this information from the global *users* hash, you can type `HGETALL user:8` to find out all the information for bob.

The reason this hash exists is because it allows you to search for users based on usernames – unlike SQL, Redis does not have a way of searching for tuples based on values of fields. The design of a Redis database must allow you to find things based on one field only. In the case of the *users* hash, that one column are the usernames of each user. (In a sense, these usernames are acting like primary keys you might see in a SQL database.) In other words, there is no command in Redis that will let you do things like "Find all users where age $>$ 24" by looping over all the *user:$userID* hashes. You have to build your own structure that will keep track of users ages so you can quickly look them up yourself.

Question 1

If you haven't already, create a new user with the username "johnsmith" with userID 11 using `HMSET` (see the example earlier in the worksheet). You've probably noticed by now that "johnsmith" isn't part of the *users* hash yet. Update the *users* hash with your new user.

To delete any key in the database, you can use the `DEL` command. So, if you wanted to delete a user, youd run:

```
DEL user:$userID
```

where you replace $userID with the user ID of the user you want to delete.

Question 2

Delete user 8, bob. Make sure to update the *users* hash to reflect the fact that bob is gone.

Now, how do we update existing information about users? For example, say that our boss has come to us saying that we want our product to be more education-focused, so our database should now start keep track of the college that each user attends.

To update our "johnsmith" user with the school they attend, run:

```
HSET user:11 school "Stanford University"
```

then run `HGETALL user:11` to have it print out the *user:11* hash and verify that it's been added successfully.

Question 3

Does adding a school field to one of our user hashes add a school field to any of the other user hashes? Run `HGETALL` on one of the other *user:$userID* hashes to check. How does this differ from how you'd add a new field to information about users in SQL?

Question 4

Say you want to add a field to some of the user hashes named *warning* that holds a string about some bad behavior the user has received a warning for. Users that are good don't have the warning field, because they've never had bad behavior (at least, they haven't been caught). Let's add the *warning* field to users 3, 5, and 10. For the value, type the warning that those users received when they were caught for bad behavior (i.e. "Swearing the kid-friendly areas of the forums"). You can make up the warnings you want to give them.

Problem 2: Filtering and Counting

Doing a count of all the users in the database is pretty easy. Just count all the fields (the usernames) of users in the *users* hash:

```
HLEN users
=> 10
```

(Depending on how many users you deleted or created, the number you get back might be different.)

Counting users based on other attributes is a bit more tricky. Again, there's no simple way in Redis to search through all the user:$userID hashes and return a list of users where city == "Stanford" (for example). As mentioned above, a Redis database must be designed so that everything is accessible by some primary key. In Redis, this means we have to maintain groups of cities and corresponding user IDs ourselves, if we want to look up users by city.

So how do we find all users that live in a certain city? The set of users that live in a certain city is just that – a set. No ordering, just a group of users. Remember that Redis has several data structures, and a set is one of them. So, we would create a set for each city, with the key corresponding to the city, and the value corresponding to a bunch of users (a set of user IDs).

For example, to create a set indicating that the users with IDs 4, 8, and 9 live in Chicago, we might run something like the following (where `SADD` is just the command for adding items to a set):

```
SADD cities:chicago:users 4
SADD cities:chicago:users 8
SADD cities:chicago:users 9
```

Then, to find the count of users living in Chicago, all we have to do is run:

```
SCARD cities:chicago:users
```

(CARD here refers to the "CARDinality" of a set.)

Like our global *users* hash, these sets are something that we'd have to update every time we update, delete, or add a new user to make sure all the data is consistent with each other.

Question 5

Take a look at all the *user:$userID* hashes to see which cities users are from. What are the names of the sets you'd have to create for each city? The names should be in the form of *cities:$cityName:users*.

Question 6

Create these sets using a bunch of `SADD cities:$cityName:users $userID` commands. You can use the `SMEMBER` command to see what a set contains and check your work.

Question 7

Find the number of users from Seattle, Boston, and Tokyo.

Question 8

These keys that you're creating should remind you of a SQL construct that acts very much in the same way – indexes. Compare and contrast SQL indexes with these keys you're creating by hand. Do they provide the same functionality? How easy or hard are they to set up compared to each other? Is there a performance difference when doing lookups? How about when modifying data?

Question 9

Do a Google search (or refer to your Redis commands cheatsheet) about Redis data structures. What new keys would you create if you wanted to query based on users's ages? How would you name them? Would you use a Set? A Hash? A Sorted Set? Why?

Problem 3: Aggregation

Now, let's calculate some statistics about the ages of our users. Again, we need to create a data structure representing a view where we can actually access users by age. In Question 9, you might have suggested that we create keys representing sets for each unique age so we can query by ages.

In Redis, there is a data structure called Sorted Sets that act kind of like Sets (in that they're just a collection of values) but attaches a score to each item in the Sorted Set. The score can be anything numeric – a ranking, a timestamp, a count, or even age! Sorted Sets come with a couple of extra commands that allow you to do simple calculations on items based on their score.

To create a Sorted Set, we use the `ZADD` command to add pairs of scores and items (in our case, ages and usernames) to a Sorted Set named *users:ages*:

```
ZADD users:ages 32 "bob"
ZADD users:ages 29 "jill"
```

and so on.

Question 10

Finish creating the *users:ages* Sorted Set using `ZADD`.

Question 11

Print out all users in order of age (increasing) using `ZRANGE users:ages 0 -1`. `ZRANGE`'s last two arguments are the starting index (here it is 0, meaning we want to start printing from the first user in the set) and the ending index (here it is -1, meaning we want to end printing from the last user in the set). You can do `ZRANGE users:ages 0 -1 WITHSCORES` to have Redis print out the ages for each user as well.

Question 12

Find the username of the youngest user. Remember, *users:ages* is sorted by age, and you can use `ZRANGE` to access users by index in the sorted set.

Question 13

Find the username of the oldest user.

Question 14

Find the user with the median age. Hint: `ZCARD users:ages` will give you the number of users in the sorted set. If you find the user in the middle of the list...

Redis Commands Cheatsheet

This is not the full set of commands available in Redis. You can see more commands at redis.io/commands.

Generic Commands

Command	Description
`exists` *key*	Test if specified key exists. Return: 1 if exists, 0 if not
`del` *key1 key2 ... keyN*	Remove the specified keys. Return: integer > 0 if keys removed, 0 if none of the keys existed
`type` *key*	Return the type of the value stored at *key*, as a string. Return: "none", "string", "list", "set"
`keys` *pattern*	Return all keys matching pattern. Ex: `keys h*llo, keys h?llo, heys h[aeo]llo` Return: bulk reply string with keys separated by spaces
`rename` *oldkey newkey*	Atomically renames key. Return 1 if OK, 0 if *oldkey* doesn't exist or if it equals *newkey*

Strings

Command	Description
`set` *key value*	Sets the value of *key* to the string *value*. Return: 1 if OK, 0 if error.
`get` *key*	Gets the value of *key*. Return: string value if OK, "nil" if *key* does not exist
`incr` *key*, `decr` *key*	Increments/decrements value of *key* by 1. Return: New value after increment/decrement operation

Sets

Command	Description
`sadd` *key member*	Adds *member* to the set stored at *key*. Return: 1 if OK, 0 if element was already a set member; error if *key* isn't a set
`srem` *key member*	Removes *member* from set key. Return: 1 if OK, 0 element not a set member; error if *key* isn't a set
`scard` *key*	Returns the number of elements in set *key*. Return: integer number of elements; 0 if empty or *key* doesn't exist
`sismember` *key member*	Return whether *member* is in set *key*. Return: 1 if element is a member, 0 if not or if *key* doesn't exist
`smembers` *key*	Returns all of the members of the set *key*. Return: the members

Sorted Sets

Command	Description
zadd *key score member*	Adds *member* to zset *key*, with specified score. Return: 1 if added, 0 if element was already a member and score was updated
zrem *key member*	Removes *member* from zset key. Return: 1 if removed, 0 element not a member
zrange *key start end [withscores]*	Returns elements in zset *key* within the specified index range, sorted in order. Option: "withscores" will also return scores.
zcard *key*	Returns the number of elements in zset *key*. Return: integer number of elements; 0 if empty or *key* doesn't exist
zscore *key element*	Returns the score of the specified element in zset *key*. Return: the score, as a string; or "nil" if *key* or *element* don't exist

Hashes

Command	Description
hset *key field value*	Sets hash *field* to the *value*. Will create a new hash if *key* does not exist. Return: 1 if new field was created, 0 if existing field was updated or already exists
hget *key field*	Returns value of *field* stored in hash *key*. Return: value of *field*, or nil if *field* or *key* do not exist
hmset *key field1 value1 .. fieldN valueN*	Sets fields to the values provided, replacing existing values if any. Creates new hash if non exists at *key*.
hexists *key field*	Returns 1 if *field* exists in hash at *key*. Returns 0 if *key* or *field* don't exist.
hdel *key field*	Removes *field* from hash stored at *key*. Return: 1 if field removed, 0 if field was not present
hlen *key*	Returns the number of fields in hash stored at *key*, or 0 if *key* does not exist
hkeys *key*, hvals *key*	Returns the keys or values of the hash stored a *key*.
hgetall *key*	Returns the keys and values of the hash stored at *key*, as a multi bulk reply in the form field1, value1, ..., fieldN, valueN.