**Redis Hands-on Practices** (Prepared by: Baburam Chaudhary)

1. **Setup and Installation: Two options: Linux or Docker**
2. **Installing redis-server in Linux System**

Open Terminal and type the following commands.

$ sudo apt update

$ sudo apt install redis-server

$ redis-server // start redis server

$ redis-server & // start redis server in background

$ ps –ef | grep redis // to check as if redis server is started or not

$ redis-cli // to start the redis client cli

// redis is running in local host and port number is 6379

1. **Using redis-server in Docker**

// download/pull redis image

Open command prompt or terminal and type following commands.

$ docker pull redis

$ docker images // to confirm if redis image is installe or not.

$ docker run –d --name <container\_name\_or\_id> <container\_name\_alias>

$ docker run –d --name redis redis

// confirm if docker container has redis image running or not.

$ docker ps

// Run redis-cli

$ docker exec –it <redis\_container\_name> redis-cli

$ docker exec –it redis redis-cli

// if all things work you should see following in terminal/cmd prompt.

127.0.0.1:6379>

// check if connection is properly established between redis client and server.

127.0.0.1:6379> PING // if connected properly, returns with PONG

1. **Data Types**
2. **String:** Redis strings store sequences of bytes, including text, serialized objects, and binary arrays. Strings are the simplest type of value you can associate with a Redis key.

$ **SET <key> <value>**

$ SET email ‘[email@domain.com](mailto:email@domain.com)’

$ GET email

$ GETRANGE email 0 4

$ DEL email

$ SET key1 “Hello”

$ APPEND key1 “ World”

$ RENAME key1 greeting // Rename a key

$ GET key1 // returns: (nil)

$ GET greeting // returns: “Hello World”

$ SET server:name someserver // using name spaces to define key

$ GET server:name

$ SET server:port 8000

$ EXISTS server:port // check if key exists or not, returns (integer) 1 if key exists.

$ SET marks 90

$ INCR marks // this increment key ‘marks’ by 1.

$ INCRBY marks 2 // this increment key by 2.

$ DECR marks // decrement marks by 1.

$ DECRBY marks 5 // decrement marks by 5.

$ SET pi 3.14

$ INCRBYFLOAT pi 0.0001

// setting multiple key/value pairs

$ MSET lang English technology Redis

$ MGET lang technology

$ STRLEN lang

$ STRLEN technology

// if you want everything to clear out entirely.

$ FLUSHALL

// set key to expire at certain points.

$ SET a 5

$ EXPIRE a 10 // after 10 seconds the key ‘a’ will be expired.

$ TTL a // to get remaining time left to survive.

// alternative way: setting key/value along with expiration

$ SETEX b 10 1 // here 10 seconds to expire and 1 is value of key b

// if you want to persist the key, after setting expiration.

$ PERSIST b // this will cancel out the expiration.

$ SET c 10 EX 50 // expire key c with value 10 after 50 seconds.

$ SET usage:63 ‘{“balance”: 600.16, “currency”: ”Rupees”, “lastLogin”: 123423535, “maxUsers”: 10, region”: “Lumbini”, “startDate”: 235325353235, “tier”: “premium”, “user”: “ramKaram”}’ EX 7200

// to get all the keys set so far

$ KEYS \*

1. **Lists:** Redis lists are linked lists of string values. Redis lists are frequently used to:
2. Implement stacks and queues.
3. Build queue management for background worker systems.

$ KEYS <pattern>

$ KEYS \*

$ LPUSH country Nepal

$ LPUSH country USA

$ LRANGE country 0 -1 // start 0 and stop -1.

$ LPUSH country Korea Japan // lpushing list of countries

$ RPUSH country Pakistan

$ LLEN country

$ LPOP country

$ RPOP country

$ LPOP country 3 // pop 3 items of list from the left.

$ RPOP country 3 //pop 3 items of list from the right.

$ LSET country 0 Germany // edit/reset the list value, here 0th item is changed to “Germany”.

// inserting the item before/after an element in a list.

$ LINSERT country before Germany “New Zealand” // here, “New Zealand” is inserted before “Germany”

$ LINSERT country after Germany “United Arabs”

$ LINSERT country after “New Zealand” Pakistan

1. **Set:** Redis Sets are unordered collection of Strings. Redis Sets have the desirable property of not allowing repeated members.

$ SADD cars “Ford”

$ SADD cars “Honda”

$SADD cars “BMW”

// check if a member is a member of the set stored at key.

$ SISMEMBER cars “Ford” // returns: (integer) 1

$ SISMEMBER cars “Mercedes” // returns: (integer) 0

// SMEMBERS: Returns all the members of the set value stored at key.

$ SMEMBERS cars

// SCARD: Returns the set cardinality (number of elements) of the set stored at key.

$ SCARD cars

// SMOVE: Move member from the set at source to the set at destination.

$ SMOVE cars mycars “Ford”

// SREM: Remove the specified members from the set stored at key.

$ SREM cars ‘Honda’

1. **Sorted Sort:** A Redis sorted set is a collection of unique strings (members) ordered by an associated score. When more than one string has the same score, the strings are ordered lexicographically.

$ ZADD users 1980 “Harihar”

$ ZADD users 1982 “Meeru”

$ ZADD users 1975 “John”

$ ZADD users 1988 “Aman”

$ ZADD users 1990 “Katti”

$ ZADD users 1990 “Taylor”

// ZRANK

$ ZRANK users “John”

$ ZRANK users “Katti”

// ZRANGE

$ ZRANGE users 0 -1

// ZINCRBY

$ ZINCRBY users 5 “John” // now John’s score is 1980.

1. **Hashes :** Redis hashes are record types structured as collections of field-value pairs. You can use hashes to represent basic objects.

// HSET: Sets field in the hash stored at key to value. If key does not exist, a new key holding a hash is created.

$ HSET user:ram name “Ram Krishna”

$ HSET user:ram email “ram@gmail.com”

$ HSET user:ram address “Banepa, Budol”

$ HGET user:ram name

$ HGET user:ram email

$ HGETALL user:ram

// setting up multiple values

$ HMSET user:john name “John Doe” email “john@domain.com” age 55

$ HGETALL user:john

// if we want just a values.

$ HVALS user:john

// HMGET: returns the values at one or more given fields.

$ HMGET user:john name email

$ HKEYS user:john // returns all fields in a hash.

$ HLEN user:john // returns the number of fields in a hash.

$ HDEL user:john age // delete the field age of hash ‘user:john’

// HINCRBY: increments the number stored at field in the hash stored at key by ‘increment’.

$ HINCRBY user:john age 5 // this increment the age field value by 5.

1. **Transactions:** executing the multiple commands at once.

$ MULTI

$ SET name “Manjil”

$ SET address “Banepa, Chardobato”

$ SET education “Bachelors in Electronics and Information Engineering”

$ SETet balance 10000

$ EXEC

// if you want to discard/abort the transactions use DISCARD, for example

$ MULTI

$ GET a

$ GET c

$ SET name “Harijan Thapaliya”

$ SET address “Panchkhel-07, Shikharapur”

$ DISCARD

// **WATCH:** We are asking Redis to perform the transaction only if none of the WATCHed keys were modified. This includes modification made by the client, like write commands, and by Redis itself, like expiration or eviction. If keys were modified between when they were WATCHed and when the EXEC was received, the entire transaction will be aborted instead.

// For this open terminal

$ WATCH b

$ SET b 10

$ multi

$ set b 2

$ get b

$ exec // exec does not run successfully.

1. **lua scripting**

$ eval “redis.call(‘set’, KEYS[1], ARGV[1])” 1 name Harihar

$ get name // returns Harihar

$ eval “redis.call(‘mset’, KEYS[1], ARGV[1], KEYS[2], ARGV[2])” 2 name last\_name Rama Raj

$ get name // returns ‘Rama’

$ get last\_name // returns ‘Raj’

$ eval “local order = redis.call(‘zrange’, KEYS[1], 0, -1); return redis.call(‘hmget’, KEYS[2], unpack(order));” 2 country country\_capital

$ script load "local order = redis.call('zrange', KEYS[1], 0, -1); return redis.call('hmget', KEYS[2], unpack(order));"

$ evalsha `hashvalue of loaded script` 2 country country\_capital

// For example.

$ evalsha 1807412636f2f95da7f3cdf6cb3bb0249e2587c7 2 country country\_capital

// check if script is available for that hash or not.

$ script exists 1807412636f2f95da7f3cdf6cb3bb0249e2587c7

// delete all scripts

$ script flush

1. **check if connection is properly established between redis client and server**.

$ PING

// selecting the databases.

$ SELECT 0

$ SELECT 1

// setting key value in database 0

$ SELECT 0

$ SET a 5

$ SET b “Nepal”

$ GET a

$ SELECT 1

$ SET a “Hello”

$ SET b “Canny Edge Detector”

$ GET a // returns ‘Hello’ because database 1 is selected.

$ GET b

$ SELECT 0

$ GET a // returns 5

$ SELECT 1

$ GET a // returns ‘Helo’

$ CLIENT LIST // checking the number of clients.

// setting the name of the client

$ CLIENT SETNAME “ThisTerminal”

$ CLIENT GETNAME

// to kill the client (by id, or addr)

$ CLIENT KILL id id\_no

$ CLIENT KILL addr addr\_no

1. **Security**

// setting password to redis client

$ config requirepass <password>

$ config requirepass 123

// lets set something

$ set a 10

// open another redis client

$ get a // you got Message “(error) NOAUTH Authentication required.”

// so use password

$ auth 123

$ get a // now returns 10

// For now we are working with redis client in a local redis server.

// what if we want to connect redis client cli with a remote server

$ redis-cli –h 127.0.0.1 –p 6379 // replace with remote server ip and port number

// **benchmarking**

$ redis-benchmark // by default 50 clients and 3bytes of data

$ redis-benchmark –n 1000 // run on 1000 commands

1. **Redis Python**

To install redis python, open Terminal or command prompt and enter following command.

$ pip install redis

1. **Pub/Sub**

SUBSCRIBE, UNSUBSCRIBE and PUBLISH implement the Publish/Subscribe messaging paradigm.

For instance, to subscribe to channels “channel1” and “ch:00” the client issures a SUBSCRIBE providing the names of the channels.

$ SUBSCRIBE channel1 ch:00

Types of Pub/Sub:

1. One-to-One
2. One-to-Many
3. Many-to-Many
4. Many-to-One

To publish message

$ PUBLISH channel1 “Hello World” // from one client.

$ PUBLISH ch:00 “This is message from channel ‘ch:00’ // from another client.

The above is an example of Many-to-One pub/sub messaging. Others can be simple applied.