**Redis server**

**Install in lunix:**

1. **cp -r /home/sc849m/redis-3.2.9.tar.gz .**
2. **tar xzf redis-3.2.9.tar.gz**
3. **cd redis-3.2.9**
4. **make**
5. **cd src/**
6. **yum install tcl (if tcl is not istalled)**
7. **make test**
8. **make install**
9. **cd ../utils/**
10. **./install\_server.sh &**

**[root@spectredev redis-3.2.9]# netstat | grep 6379**

**tcp 0 0 localhost:60082 localhost:6379 TIME\_WAIT**

**[root@spectredev redis-3.2.9]# ps -ef|grep redis**

**root 15493 1 0 15:50 ? 00:00:01 /usr/local/bin/redis-server 127.0.0.1:6379**

**root 15519 9117 0 16:01 pts/0 00:00:00 grep redis**

**Link:**

[**https://discuss.pivotal.io/hc/en-us/articles/205308418-How-to-install-and-use-Redis-on-Linux**](https://discuss.pivotal.io/hc/en-us/articles/205308418-How-to-install-and-use-Redis-on-Linux)

[**https://redis.io/topics/quickstart**](https://redis.io/topics/quickstart)

[**https://redis.io/topics/data-types-intro**](https://redis.io/topics/data-types-intro)

**Checking for redis is working:**

**[root@spectredev redis-3.2.9]# redis-cli**

**127.0.0.1:6379> ping**

**PONG**

**127.0.0.1:6379>**

**127.0.0.1:6379>**

**127.0.0.1:6379> set sachin chippalkatti**

**OK**

**127.0.0.1:6379> get sachin**

**"chippalkatti"**

**14/11/2019**

**INSTALLATION:**

$ tar xzf redis-5.0.6.tar.gz

$ cd redis-5.0.6

$ make

$ make test

The binaries that are now compiled are available in the src directory. Run Redis with:

$ src/redis-server

You can interact with Redis using the built-in client:

$ src/redis-cli

Redis is not a plain key-value store, it is actually a data structures server, supporting different kinds of values.

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
* 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.
* Streams: append-only collections of map-like entries that provide an abstract log data type. They are covered in depth in the [Introduction to Redis Streams](https://redis.io/topics/streams-intro).

**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.

## Redis Strings

The Redis String type is the simplest type of value you can associate with a Redis key

SET key value [expiration EX seconds|PX milliseconds] [NX|XX]

> set mykey somevalue

OK

> get mykey

"somevalue"

**SET foo1 "sachin" EX 10**

The [SET](https://redis.io/commands/set) command has interesting options, that are provided as additional arguments. For example, I may ask [SET](https://redis.io/commands/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

**NX –** I may ask [SET](https://redis.io/commands/set) to fail if the key already exists.

XX - I may ask [SET](https://redis.io/commands/set) to success if the key already exists.

**127.0.0.1:6379> GET foo1**

**"200"**

**127.0.0.1:6379> SET foo1 100 NX**

**(nil)**

**127.0.0.1:6379> GET foo1**

**"200"**

**127.0.0.1:6379> SET foo1 100 XX**

**OK**

**127.0.0.1:6379> GET foo1**

**"100"**

> set counter 100

OK

> incr counter

(integer) 101

> incr counter

(integer) 102

> incrby counter 50

(integer) 152

**DECR foo1 100**

**DECRBY foo1 100**

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](https://redis.io/commands/mset) and [MGET](https://redis.io/commands/mget) commands:

> mset a 10 b 20 c 30

OK

> mget a b c

1) "10"

2) "20"

3) "30"

When [MGET](https://redis.io/commands/mget) is used, Redis returns an array of values.

the [EXISTS](https://redis.io/commands/exists) command returns 1 or 0 to signal if a given key exists or not in the database, while the [DEL](https://redis.io/commands/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

> type mykey

string

## Redis expires: keys with limited time to live

**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](https://redis.io/commands/del) command with the key.

> expire key 5

(integer) 1

> get key (immediately)

"some-value"

> get key (after some time)

(nil)

> set key 100 ex 10

OK

> ttl key

(integer) 9

**127.0.0.1:6379> persist b**

**(integer) 1**

**127.0.0.1:6379> get b**

**"20"**

## Redis Lists

The [LPUSH](https://redis.io/commands/lpush) command adds a new element into a list, on the left (at the head), while the [RPUSH](https://redis.io/commands/rpush) command adds a new element into a list, on the right (at the tail). Finally the [LRANGE](https://redis.io/commands/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"

> rpush mylist a b c

(integer) 3

> rpop mylist

"c"

> rpop mylist

"b"

> rpop mylist

"a"

> rpush mylist 1 2 3 4 5

(integer) 5

> ltrim mylist 0 2

OK

> lrange mylist 0 -1

1) "1"

2) "2"

3) "3"

**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"

hmget user:1000 username birthyear no-such-field

1) "antirez"

2) "1977"

3) (nil)

> hincrby user:1000 birthyear 10

(integer) 1987

> hincrby user:1000 birthyear 10

(integer) 1997

## Redis Sets

Redis Sets are unordered collections of strings. The [SADD](https://redis.io/commands/sadd) command adds new elements to a set.

> sadd myset 1 2 3

(integer) 3

> smembers myset

1. 3

2. 1

3. 2

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

> sismember myset 3

(integer) 1

> sismember myset 30

(integer) 0

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

> spop game:1:deck

"C6"

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](https://redis.io/commands/scard).

> scard game:1:deck

(integer) 47

## Redis Sorted sets

**zadd key [NX|XX] [CH] [INCR] score member [score member ...]**

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*

> zadd hackers 1940 "Alan Kay"

(integer) 1

> zadd hackers 1957 "Sophie Wilson"

(integer) 1

> zrange hackers 0 -1

1) "Alan Turing"

2) "Hedy Lamarr"

> zrevrange hackers 0 -1

1) "Linus Torvalds"

2) "Yukihiro Matsumoto"

> zrange hackers 0 -1 withscores

1) "Alan Turing"

2) "1912"

> zrank hackers "Anita Borg"

(integer) 4

## Lexicographical scores

> zrangebylex hackers [B [P

To Clear all the data:

127.0.0.1:6379> flushall

OK

# Redis Streams

Redis streams are primarily an append only data structure. At least conceptually, because being Redis Streams an abstract data type represented in memory, they implement more powerful operations, to overcome the limits of the log file itself.

**XADD mystream ID field string [field string ...]**

**127.0.0.1:6379> XADD mystream \* name sachin address chippalkatti**

**"1573734656522-0"**

**127.0.0.1:6379> multi**

**OK**

**127.0.0.1:6379> XADD mystream \* name anand address chippalkatti**

**QUEUED**

**127.0.0.1:6379> XADD mystream \* name swati address chippalkatti**

**QUEUED**

**127.0.0.1:6379> exec**

**1) "1573734781508-0"**

**2) "1573734781508-1"**

**127.0.0.1:6379> xrange mystream - +**

**1) 1) "1573734656522-0"**

**2) 1) "name"**

**2) "sachin"**

**3) "address"**

**4) "chippalkatti"**

**2) 1) "1573734781508-0"**

**2) 1) "name"**

**2) "anand"**

**3) "address"**

**4) "chippalkatti"**

**3) 1) "1573734781508-1"**

**2) 1) "name"**

**2) "swati"**

**3) "address"**

**4) "chippalkatti"**

**Can be used as Consumer:**

**127.0.0.1:6379> xread STREAMS mystream 0**

**1) 1) "mystream"**

**2) 1) 1) "1573734656522-0"**

**2) 1) "name"**

**2) "sachin"**

**3) "address"**

**4) "chippalkatti"**

**2) 1) "1573734781508-0"**

**2) 1) "name"**

**2) "anand"**

**3) "address"**

**4) "chippalkatti"**

**3) 1) "1573734781508-1"**

**2) 1) "name"**

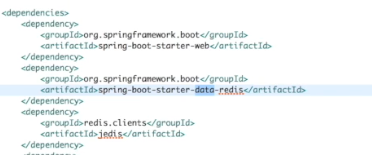
**2) "swati"**

**3) "address"**

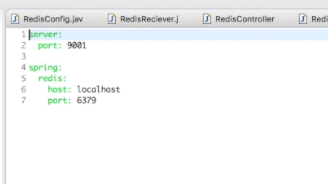
**4) "chippalkatti"**

**Redis can be used as**

1. **In memory Database**
2. **Cache**
3. **Message broker**

****

****

****