1.	Start Redis
1.1	make sure vagrant is installed: http://www.vagrantup.com/downloads
1.2	install virtualbox: https://www.virtualbox.org/wiki/Downloads
1.3	install git: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
1.4	download the project - go to a folder of your choice via command line window - execute the command: **git clone https://github.com/tomvdbulck/training-no-sql.git**
1.5	start up the vagrant box - navigate to the redis/vagrant folder: ***cd redis/vagrant*** - enter the command: ***vagrant up***
This will now download and install the vagrant box, this might take a while	
1.6	start up the redis server via a **command prompt** - enter the command: **vagrant ssh** - you will now go the the vagrant box - start the server via entering: **redis-server**
1.7	Lets keep that command line window open and open **new command prompt** - navigate to the **redis/vagrant** folder - enter the command **vagrant ssh** You should see vagrant@precise32:~$
1.8	Verify if the redis server is running, enter the command: **redis-cli ping** the server will reply with PONG
2.	Basic operations (set, get, mset, mget, delete, exists, incr, expire and TTL)
	go into command line and enter: **redis-cli** you will now see a new prompt: 127.0.0.1:6379>
2.1	SET will allow you to enter a key/value pair, it will either insert or update an entry.

	Enter **set mykey myvalue** That will return OK Via **get mykey** you should see the value: myvalue
2.2	You can delete values by **del mykey** This returns 1 When you will now execute **get mykey** you will get Nil as value Running **del mykey** again will return 0 (as there is no more element to delete)
2.3	Set by default will automatically update the value if the key already exists. **set mykey myvalue** returns OK **set mykey newvalue** returns OK via **get mykey** you can see that the value is now newvalue
2.4	Some additional arguments can be passed with the SET command **set mykey myvalue** => OK **set mykey value2 nx** will return Nil - this because nx will only allow a SET if the key does not exist **set mykey2 value2 nx** will return OK By passing xx as argument SET will only work if the key already exists. **set myNonExistingKey myvalue xx** will return Nil set mykey value2 xx will return OK
2.5	INCR allows you to increase a value. For example: **set counter 100** is just an integer value stored as a string By entering **incr counter**, redis will automatically convert the String to an Integer and increase it by 1. **get counter** will show you that the value is now 101 You can also increase by more then one via **incrby counter 50**. Via **get counter** you will see that the value is now 151 If a value is not numeric, like **set mykey value**, then **incr mykey** will throw an error: (error) ERR value is not an integer or out of range

2.6	via **type mykey** you can see that it is a string while the **type counter** will return an integer
2.7	via EXISTS you can see if a key exists or not. **exists akey** returns (integer) 0. But after the command **set akey is_really_a_key** **exists akey** will now return (integer) 1
2.8	MSET allows you to set multiple keys at once **mset a 10 b 20 c 30** will return OK and enter 3 key/value pairs via **mget a b c** you can get the values of multiple keys at once: 1) "10" 2) "20" 3) "30"
2.9	Redis allows you to define a time to live on a record, meaning: after the time you defined the record will be deleted automatically. **set key toExpire someValue** At this moment it is just a normal record. But when you run **expire toExpire 5** you still have 5 seconds time to run **get toExpire,** returning someValue. After 5 seconds **get toExpire** will no longer result in a value (Nil instead) **ttl toExpire** allows you to see the current time to live - which can be 5, 4, 3, …
3.	Redis Lists
	Redis lists are implemented as linked lists This means that adding a new element to the back/front of the list takes little to no time (even if you have milions of entries) However - accessing elements via the key in a linked list is not that fast.

	Redis prioritizes adding data to a list. Redis lists can also be defined with a maximum length - which can also be an advantage as you will see later on. For fast access to the middle of a list a different data structure can be used, called sorted sets - this will also be handled later on
3.1	The RPUSH command adds a new element into a list on the right, at the tail. > **rpush mylist A** (integer) 1 > **rpush mylist B** (integer) 2 The LPUSH command adds a new element on the left, at the head. > **lpush mylist first** (integer) 3 Via LRANGE you can extract ranges of elements from a list > **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.
3.2	You can also push multiple elements in a single call. (variadic commands) > 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"
3.3	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”
3.4	**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. Enter the following:

	> **rpush mylist 1 2 3 4 5** (integer) 5 > **ltrim mylist 0 2** OK > **lrange mylist 0 -1** 1) "1" 2) "2" 3) "3"
4.	Redis Hashes
4.1	Redis hashes allow you to represent field/value pairs in an “object” > **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"
4.2	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 HGETbut returns an array of values: > **hmget user:1000 username birthyear no-such-field** 1) "antirez" 2) "1977" 3) (nil)
5.	Redis Sets
	Redis Sets are unordered collections of strings.
5.1	With the SADD command you can add elements to a set > **sadd myset 1 2 3** (integer) 3 > **smembers myset** 1. 3 2. 1 3. 2
5.2	Redis has commands to test for membership. Does a given element exist? > **sismember myset 3** (integer) 1

	> **sismember myset 30** (integer) 0
5.3	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. Imagine we want to tag news. If our news ID 1000 is tagged with tags 1, 2, 5 and 77, we can have one set associating our tag IDs with the news item: > **sadd news:1000:tags 1 2 5 77** (integer) 4
5.4	However sometimes I may 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
5.5	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 ….
6.	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.
6.1	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 1969 "Linus Torvalds"** (integer) 1 > **zadd hackers 1912 "Alan Turing"** (integer) 1 As you can see zadd is similar to sadd, but takes an extra element: the score
6.2	With sorted sets it is trivial to return a list of hackers sorted by their birth year because actually *they are already sorted*. > **zrange hackers 0 -1** 1) "Alan Turing" 2) "Hedy Lamarr" 3) "Claude Shannon" 4) "Alan Kay" Note: 0 -1 means from the first element (0) to the last (-1)
6.3	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"
*	Extra: poker game
A.1	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

A.2	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 thegame:1:deck key. This is accomplished usingSUNIONSTORE, 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
A.3	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"
A.4	Now it's 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.