A Brief Introduction to Redis

Dirk Eddelbuettel[1]

[1]Department of Statistics, University of Illinois, Urbana-Champaign, IL, USA

This version was compiled on February 13, 2022

This note provides a brief introduction to Redis highlighting its usefulness in multi-lingual statistical computing.

Overview and Introduction

Redis (Sanfilippo, 2009) is a very popular, very powerful, and very widely-used 'in-memory database-structure store'. It runs as a background process (a "daemon" in Unix lingo) and can be accessed locally or across a network making it very popular choice for 'data caches'. There is more to say about Redis than we possibly could in a *short* vignette introducing it, and other places already do so. The Little Redis Book (Seguin, 2012), while a decade old (!!) is a fabulous *short* start. The official site is very good as well (but by now a little overwhelming as so many features have been added).

This vignette aims highlight two aspects: how easy it is to use Redis on simple examples, and also to stress how Redis enables easy *multi-lingual* computing as it can act as a 'middle-man' between any set of languages capable of speaking the Redis protocol – which may cover most if not all common languages one may want to use!

Data Structure Example One: Key-Value Setter and Getter

We will show several simple examples for the

- `redis-cli` command used directly or via shell scripts
- Python via the standard Python package for Redis
- C / C++ via the hiredis library
- R via **RcppRedis** (Eddelbuettel and Lewis, 2022) utilising the hiredis library

to demonstrate how different languages all can write to and read from Redis. Our first example will use the simplest possibly data structure, a simple `SET` and `GET` of a key-value pair.

Command-Line. `redis-cli` is command-line client. Use is straightforward as shown an simply key-value getter and setter. We show use in 'shell script' form here, but the same commands also work interactively.

```
## note that document processing will show all
## three results at once as opposed to one at time
redis-cli SET ice-cream chocolate
redis-cli GET ice-cream
redis-cli GET ice-cream
#  OK
#  chocolate
#  chocolate
```

Here, as in general, we will omit hostname and authentication arguments: on the same machine, `redis-cli` and the background `redis` process should work 'as is'. For access across a (local or remote) network, the configuration will have to be altered to permit access at given network interfaces and IP address ranges.

We show the redis commands used in uppercase notation, this is in line with the documentation. Note, however, that Redis itself is case-insensitive here so `set` is equivalent to `SET`.

Python. Redis does have bindings for most, if not all, languages to computing with data. Here is a simple Python example.

```
import redis

redishost = "localhost"
redisserver = redis.StrictRedis(redishost)

key = "ice-cream"
val = "strawberry"
res = redisserver.set(key, val)
print("Set", val, "under", key, "with result", res)
#  Set strawberry under ice-cream with result True
key = "ice-cream"
val = redisserver.get(key)
print("Got", val, "from", key)
#  Got b'strawberry' from ice-cream
```

For Python, the redis commands are generally mapped to (lowercase named) member functions of the instantiated redis connection object, here `redisserver`.

C / C++. Compiled languages work similarly. For C and C++, the hiredis 'minimalistic' library from the Redis project can be used—as it is by **RcppRedis**. Here we only show the code without executing it. This example is included in the package are as the preceding ones. C and C++ work similarly to the interactive or Python commands. A simplified (and incomplete, see the `examples/` directory of the package for more) example of writing to Redis would be

```
redisContext *prc;    // pointer to redis context

std::string host = "127.0.0.1";
int port = 6379;

prc = redisConnect(host.c_str(), port);
// should check error here
redisReply *reply = (redisReply*)
    redisCommand(prc, "SET ice-cream %s", value);
// should check reply here
```

Reading is done by submitting for example a `GET` command for a key after which the `redisReply` contains the reply string.

R. The **RcppRedis** packages uses **Rcpp** Modules along with **Rcpp** (Eddelbuettel *et al.*, 2022; Eddelbuettel and Balamuta, 2018) to connect the hiredis library to R. A simple R example follows.

```
library(RcppRedis)
redis <- new(Redis, "localhost")
redis$set("ice-cream", "hazelnut")
```

```
#  [1] "OK"
redis$get("ice-cream")
#  [1] "hazelnut"
```

Data Structure Example Two: Hashes

Redis has support for a number of standard data structures including hashes. The official documentation list sixteen commands in the corresponding group covering writing (`hset`), reading (`hget`), checking for key (`hexists`), deleting a key (`hdel`) and more.

```
redis-cli HSET myhash abc 42
redis-cli HSET myhash def "some text"
#  1
#  1
```

We can illustrate reading and checking from Python:

```
print(redisserver.hget("myhash", "abc"))
#  b'42'
print(redisserver.hget("myhash", "def"))
#  b'some text'
print(redisserver.hexists("myhash", "xyz"))
#  False
```

For historical reasons, **RcppRedis** converts to/from R internal serialization on hash operations so it cannot *directly* interoperate with these examples as they not deploy R-internal representation. The package has however a 'catch-all' command `exec` (which executes a given Redis command string) which can be used here:

```
redis$exec("HGET myhash abc")
#  [1] "42"
redis$exec("HGET myhash def")
#  [1] "some text"
redis$exec("HEXISTS myhash xyz")
#  [1] 0
```

Data Structure Example Three: Sets

Sets are another basic data structure that is well-understood. In sets, elements can be added (once), removed (if present), and sets can be joined, intersected and differenced.

```
redis-cli SADD myset puppy
redis-cli SADD myset kitten
redis-cli SADD otherset birdie
redis-cli SADD otherset kitten
redis-cli SINTER myset otherset
#  1
#  1
#  1
#  1
#  kitten
```

We can do the same in Python. Here we showi only the final intersect command—the set-addition commands are equivalent across implementations as are most other Redis command.

```
print(redisserver.sinter("myset", "otherset"))
#  {b'kitten'}
```

And similarly in R where `exec` returns a list:

```
redis$exec("SINTER myset otherset")
#  [[1]]
#  [1] "kitten"
```

Data Structure Example Four: Lists

So far we have looked at setters and getters for values, hashes, and sets. All of these covered only one value per key. But Redis has support for many more types.

```
redis-cli LPUSH mylist chocolate
redis-cli LPUSH mylist strawberry vanilla
redis-cli LLEN mylist
#  1
#  3
#  3
```

We can access the list in Python. Here we show access by index. Note that the index is zero-based, so 'one' accesses the middle element in a list of length three.

```
print(redisserver.lindex("mylist", 1))
#  b'strawberry'
```

In R, using the 'list range' command for elements 0 to 1:

```
redis$exec("LRANGE mylist 0 1")
#  [[1]]
#  [1] "vanilla"
#
#  [[2]]
#  [1] "strawberry"
```

The **RcppRedis** list commands (under the 'standard' names) work on serialized R objects so we once again utilize the `exec` command to execute this using the 'standard' name. As access to unserialized data is useful, the package also two alternates for numeric and string return data:

```
redis$listRangeAsStrings("mylist", 0, 1)
#  [1] "vanilla"    "strawberry"
```

Data Structure Example Five: Sorted Sets

Redis offers another data structure that can be of interest to us for use in time series. Recall how packages **zoo** (Zeileis *et al.*, 2021) and **xts** (Ryan and Ulrich, 2020) are, essentially, indexed containers around (numeric) matrices with a sorting index. This is commonly the `Date` type in R for daily data, or a `POSIXct` datimetime type for intra-daily data at approximately a microsecond resolution. One can then index by day or datetime, subset, merge, ... We can store such data in Redis using sorted sets using the index as the first column. A quick R example illustrates.

```
m1 <- matrix(c(100, 1, 2, 3), 1)
redis$zadd("myz", m1) # add m1 indexed at 100
#  [1] 1
m2 <- matrix(c(105, 2, 2, 4), 1)
redis$zadd("myz", m2) # add m1 indexed at 105
#  [1] 1
```

In this first example we insert two matrices (with three values each) index at 100 and 105, respectively, to the sorted set under key `myz`. We will then ask for a range of data over the range from 90 to 120 which will include both sets of observations.

```
res <- redis$zrangebyscore("myz", 90, 120)
res
#       [,1] [,2] [,3] [,4]
#  [1,]  100    1    2    3
#  [2,]  105    2    2    4
```

Communication Example: Publish/Subscribe

We have seen above that writen a *value* to a particular *key* into a list, set, or sorted set is straightforward. So is publishing into a *channel*. Redis keeps track of the current subscribers to a channel and dispatches the published content.

Subscribers can join, and leave, anytime. Data is accessible via the publish/subscribe (or "pub/sub") mechanism while being subscribe. There is no mechanism *within pub/sub* to obtain 'older' values, or to re-request values. Such services can however be provided by Redis is its capacity of a data store.

As subscription is typically blocking, we cannnot show a simple example in the vignette. But an illustration (without running code) follows.

```
ch1 <- function(x) { cat("Got", x, "in ch1\n") }
redis$subscribe("ch1")
```

Here we declare a callback function which by our convention uses the same name as the channel. So in the next when the subscription is activated, the callback function is registered with the current Redis object. Once another process or entity publishes to the channel, the callback function will be invoked along with the value published on the channel.

Application Example: Hash R Objects

The ability to serialize R object makes it particularly to store R objects directly. This can be useful for data sets, and well as generated data

```
fit <- lm(Volume ~ . - 1, data=trees)
redis$hset("myhash", "data", trees)
#  [1] 1
redis$hset("myhash", "fit", fit)
#  [1] 1
fit2 <- redis$hget("myhash", "fit")
all.equal(fit, fit2)
#  [1] TRUE
```

The retrieved model fit is equal to the one we stored in Redis. We can also re-fit on the retrieved data and obtain the same coefficient. (The fit object stores information about the data set which differs here for technical reason internal to R; the values are the same.)

```
data2 <- redis$hget("myhash", "data")
fit2 <- redis$hget("myhash", "fit")
fit3 <- lm(Volume ~ . - 1, data=data2)
all.equal(coef(fit2), coef(fit3))
#  [1] TRUE
```

Summary

This vignettet introduces the Redis data structure engine, and demonstrates how reading and writing different data types from different programming languages including R, Python and shell is concise and effective. A final example of storing an R dataset and model fit further illustrates the versatility of Redis.

References

Eddelbuettel D, Balamuta JJ (2018). "Extending R with C++: A Brief Introduction to Rcpp." *The American Statistician*, **72**(1). doi:10.1080/00031305.2017.1375990.

Eddelbuettel D, François R, Allaire J, Ushey K, Kou Q, Russel N, Chambers J, Bates D (2022). *Rcpp: Seamless R and C++ Integration*. R package version 1.0.8, URL https://CRAN.R-Project.org/package=Rcpp.

Eddelbuettel D, Lewis BW (2022). *RcppRedis: 'Rcpp' Bindings for 'Redis' using the 'hiredis' Library*. R package version 0.2.0, URL https://CRAN.R-Project.org/package=RcppRedis.

Ryan JA, Ulrich JM (2020). *xts: eXtensible Time Series*. R package version 0.12.1, URL https://CRAN.R-project.org/package=xts.

Sanfilippo S (2009). "Redis In-memory Data Structure Server." https://redis.io.

Seguin K (2012). "The Little Redis Book." https://www.openmymind.net/redis.pdf.

Zeileis A, Grothendieck G, Ryan JA (2021). *zoo: S3 Infrastructure for Regular and Irregular Time Series (Z's Ordered Observations)*. R package version 1.8-9, URL https://CRAN.R-project.org/package=zoo.