Partitioning: how to split data among multiple Redis instances.

Partitioning is the process of splitting your data into multiple Redis instances, so that every instance will only contain a subset of your keys. The first part of this document will introduce you to the concept of partitioning, the second part will show you the alternatives for Redis partitioning.

Why partitioning is useful

Partitioning in Redis serves two main goals:

- It allows for much larger databases, using the sum of the memory of many computers. Without partitioning you are limited to the amount of memory a single computer can support.
- It allows scaling the computational power to multiple cores and multiple computers, and the network bandwidth to multiple computers and network adapters.

Partitioning basics

There are different partitioning criteria. Imagine we have four Redis instances **R0**, **R1**, **R2**, **R3**, and many keys representing users like `user:1`, `user:2`, ... and so forth, we can find different ways to select in which instance we store a given key. In other words there are *different systems to map* a given key to a given Redis server.

One of the simplest ways to perform partitioning is with **range partitioning**, and is accomplished by mapping ranges of objects into specific Redis instances. For example, I could say users from ID 0 to ID 10000 will go into instance **R0**, while users form ID 10001 to ID 20000 will go into instance **R1** and so forth.

This system works and is actually used in practice, however, it has the disadvantage of requiring a table that maps ranges to instances. This table needs to be managed and a table is needed for every kind of object, so therefore range partitioning in Redis is often undesirable because it is much more inefficient than other alternative partitioning approaches.

An alternative to range partitioning is **hash partitioning**. This scheme works with any key, without requiring a key in the form `object_name:<id>`, and is as simple as:

- Take the key name and use a hash function (e.g., the `crc32` hash function) to turn it into a number. For example, if the key is `foobar`, `crc32(foobar)` will output something like `93024922`.

- Use a modulo operation with this number in order to turn it into a number between 0 and 3, so that this number can be mapped to one of my four Redis instances. `93024922 modulo 4` equals 2, so I know my key `foobar` should be stored into the **R2** instance. *Note: the modulo operation returns the remainder from a division operation, and is implemented with the `%` operator in many programming languages.*

There are many other ways to perform partitioning, but with these two examples you should get the idea. One advanced form of hash partitioning is called **consistent hashing** and is implemented by a few Redis clients and proxies.

Different implementations of partitioning

Partitioning can be the responsibility of different parts of a software stack.

- **Client side partitioning** means that the clients directly select the right node where to write or read a given key. Many Redis clients implement client side partitioning.
- **Proxy assisted partitioning** means that our clients send requests to a proxy that is able to speak the Redis protocol, instead of sending requests directly to the right Redis instance. The proxy will make sure to forward our request to the right Redis instance according to the configured partitioning schema, and will send the replies back to the client. The Redis and Memcached proxy Twemproxy implements proxy assisted partitioning.
- **Query routing** means that you can send your query to a random instance, and the instance will make sure to forward your query to the right node. Redis Cluster implements an hybrid form of query routing, with the help of the client (the request is not directly forwarded from a Redis instance to another, but the client gets *redirected* to the right node).

Disadvantages of partitioning

Some features of Redis don't play very well with partitioning:

- Operations involving multiple keys are usually not supported. For instance you can't perform the intersection between two sets if they are stored in keys that are mapped to different Redis instances (actually there are ways to do this, but not directly).
- Redis transactions involving multiple keys can not be used.
- The partitioning granularity is the key, so it is not possible to shard a dataset with a single huge key like a very big sorted set.
- When partitioning is used, data handling is more complex, for instance you have to handle multiple RDB / AOF files, and to make a backup of your data you need to aggregate the persistence files from multiple instances and hosts.
- Adding and removing capacity can be complex. For instance Redis Cluster supports mostly transparent rebalancing of data with the ability to add and remove nodes at runtime, but other systems like client side partitioning and proxies don't support this feature. However a technique called *Pre-sharding* helps in this regard.

Data store or cache?

Although partitioning in Redis is conceptually the same whether using Redis as a data store or as a cache, there is a significant limitation when using it as a data store. When Redis is used as a data store, a given key must always map to the same Redis instance. When Redis is used as a cache, if a given node is unavailable it is not a big problem if a different node is used, altering the key-instance map as we wish to improve the *availability* of the system (that is, the ability of the system to reply to our queries).

Consistent hashing implementations are often able to switch to other nodes if the preferred node for a given key is not available. Similarly if you add a new node, part of the new keys will start to be stored on the new node.

The main concept here is the following:

- If Redis is used as a cache **scaling up and down** using consistent hashing is easy.
- If Redis is used as a store, **a fixed keys-to-nodes map is used, so the number of nodes must be fixed and cannot vary**. Otherwise, a system is needed that is able to rebalance keys between nodes when nodes are added or removed, and currently only Redis Cluster is able to do this - Redis Cluster is generally available and production-ready as of April 1st, 2015.

Presharding

We learned that a problem with partitioning is that, unless we are using Redis as a cache, to add and remove nodes can be tricky, and it is much simpler to use a fixed keys-instances map.

However the data storage needs may vary over the time. Today I can live with 10 Redis nodes (instances), but tomorrow I may need 50 nodes.

Since Redis has an extremely small footprint and is lightweight (a spare instance uses 1 MB of memory), a simple approach to this problem is to start with a lot of instances from the start. Even if you start with just one server, you can decide to live in a distributed world from day one, and run multiple Redis instances in your single server, using partitioning.

And you can select this number of instances to be quite big from the start. For example, 32 or 64 instances could do the trick for most users, and will provide enough room for growth.

In this way as your data storage needs increase and you need more Redis servers, what you do is simply move instances from one server to another. Once you add the first additional server, you will need to move half of the Redis instances from the first server to the second, and so forth.

Using Redis replication you will likely be able to do the move with minimal or no downtime for your users:

- Start empty instances in your new server.
- Move data configuring these new instances as slaves for your source instances.
- Stop your clients.

- Update the configuration of the moved instances with the new server IP address.
- Send the `SLAVEOF NO ONE` command to the slaves in the new server.
- Restart your clients with the new updated configuration.
- Finally shut down the no longer used instances in the old server.

Implementations of Redis partitioning

So far we covered Redis partitioning in theory, but what about practice? What system should you use?

Redis Cluster

Redis Cluster is the preferred way to get automatic sharding and high availability. It is generally available and production-ready as of April 1st, 2015. You can get more information about Redis Cluster in the Cluster tutorial.

Once Redis Cluster is available, and if a Redis Cluster compliant client is available for your language, Redis Cluster will be the de facto standard for Redis partitioning.

Redis Cluster is a mix between *query routing* and *client side partitioning*.

Twemproxy

Twemproxy is a proxy developed at Twitter for the Memcached ASCII and the Redis protocol. It is single threaded, it is written in C, and is extremely fast. It is open source software released under the terms of the Apache 2.0 license.

Twemproxy supports automatic partitioning among multiple Redis instances, with optional node ejection if a node is not available (this will change the keys-instances map, so you should use this feature only if you are using Redis as a cache).

It is *not* a single point of failure since you can start multiple proxies and instruct your clients to connect to the first that accepts the connection.

Basically Twemproxy is an intermediate layer between clients and Redis instances, that will reliably handle partitioning for us with minimal additional complexities.

You can read more about Twemproxy in this antirez blog post.

Clients supporting consistent hashing

An alternative to Twemproxy is to use a client that implements client side partitioning via consistent hashing or other similar algorithms. There are multiple Redis clients with support for consistent hashing, notably Redis-rb, Predis and Jedis.

Please check the full list of Redis clients to check if there is a mature client with consistent hashing implementation for your language.