Clustering Redis Databases

For very large databases, Redis Enterprise Cloud distributes database data to different cloud instances. For example:

- When data grows beyond the the RAM resources of a single server.
  
  Multiple shards should be used when data grows to 25 GB (50 GB for Redis on Flash) to create multiple shards.

- The operations performed against the database are CPU intensive enough to degrade performance.
  
  Clustering distributes operational load, whether to instances on the same server or across multiple servers.

This distribution is called "clustering" because it manages the way data is distributed throughout the cluster of nodes that support the database.

How data is distributed

A Redis Cloud cluster is a set of managed Redis processes and cloud instances, with each process managing a subset of the database keyspace. Clustering uses multiple cores and resources of multiple instances to overcome scaling challenges.

In a Redis Cloud cluster, the keyspace is partitioned into hash slots. At any given time a slot resides on and is managed by a single Redis server.

An instance that belongs to a cluster can manage multiple slots. This division of the key space, known as "sharding", is achieved by hashing the key names, or parts of these (key hash tags), in order to obtain the slot in which a key should reside.

Even when using multiple Redis processes, the use of a Redis Enterprise Cloud cluster is nearly transparent to the application that uses it. The cluster is accessible via a single endpoint that automatically routes all operations to the relevant shards, without the complexity of a cluster-aware Redis client. This allows applications to benefit from using the cluster without performing any code changes, even if they were not designed to use it beforehand.

When creating or editing a Redis database on Redis Enterprise Cloud, the system automatically calculates the number of shards needed based on the database memory limit and required throughput.

Multi-key operations

Operations on multiple keys in a sharded Redis Cloud cluster are supported with the following limitations:

1. **Multi-key commands**: Redis offers several commands that accept multiple keys as arguments. In a sharded setup, multi-key commands can only be used when all affected keys reside in the same slot. These commands are: BITOP, BLPOP, BRPOP, BRPOPLPUSH, MSETNX, RPOPLPUSH, SDIFF, SDIFFSTORE, SINTER, SINTERSTORE, SMOVE, SORT, SUNION, ZINTER, ZINTERSTORE, ZUNION, ZUNIONSTORE, ZDIFF, ZDIFFSTORE

2. **Geo commands**: In GEORADIUS/GEORADIUSBYMEMBER/GEOSEARCHSTORE commands, the STORE and STOREDIST options can only be used when all affected keys reside in the same slot.

3. **Transactions**: All operations within a WATCH/MULTI/EXEC block should be performed on keys that are in the same slot.

4. **Lua scripts**: All keys that are used by the script must reside in the same slot and need to be provided as arguments to the EVAL/EVALSHA commands (as per the Redis specification).

5. **Renaming/Copy keys**: The use of the RENAME/RENAME NX/COPY commands is allowed only when both the key’s original name and its new name are mapped to the same hash slot.
6. **Variadic commands**: The use of (MGET, MSET, HMGET, HMSET, etc..) and pipelining are supported with Redis Cloud cluster like if it were a non-cluster DB.

**Changing the hashing policy**

The clustering configuration of a Redis Cloud instance can be changed. However, hashing policy changes delete existing data (FLUSHDB) before they’re applied. These changes include:

1. Changing the hashing policy, either from standard to custom or vice versa.
2. Changing the order of custom hashing policy rules.
3. Adding rules before existing ones in the custom hashing policy.
4. Deleting rules from the custom hashing policy.
5. Disabling clustering for the database.

**Standard hashing policy**

When using the standard hashing policy, a Redis Cloud cluster behaves like the standard, open-source Redis cluster, and hashing is performed as follows:

1. Keys with a hashtag: a key’s hashtag is any substring between ‘{’ and ‘}’ in the key’s name. That means that when a key’s name includes the pattern ‘{…}’, the hashtag is used as input for the hashing function. For example, the following key names have the same hashtag and are mapped to the same slot: foo{bar}, {bar}baz & foo{bar}baz.
2. Keys without a hashtag: when a key doesn’t contain the ‘{…}’ pattern, the entire key’s name is used for hashing.

You can use the ‘{…}’ pattern to direct related keys to the same hash slot, so that multi-key operations are supported on them. On the other hand, not using a hashtag in the key’s name results in a (statistically) even distribution of keys across the keyspace’s shards. If your application does not perform multi-key operations, you don’t need to construct key names with hashtags.

**Custom hashing policy**

A Redis Cloud cluster can be configured to use a custom hashing policy. A custom hashing policy is required when different keys need to be kept together on the same shard to allow multi-key operations. Redis Cloud’s custom hashing policy is provided via a set of Perl Compatible Regular Expressions (PCRE) rules that describe the dataset’s key name patterns.

To configure a custom hashing policy, enter regular expression (RegEx) rules that identify the substring in the key’s name - hashtag - on which hashing will be done. The hashing tag is denoted in the RegEx by the use of the ‘tag’ named subpattern. Different keys that have the same hashtag will be stored and managed in the same slot.

Once you enable the custom hashing policy, the Redis Cloud’s default RegEx rules that implement the standard hashing policy are:

<table>
<thead>
<tr>
<th>RegEx Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.<em>{(?&lt;tag&gt;.</em>).}.*</td>
<td>Hashing is done on the substring between the curly braces.</td>
</tr>
<tr>
<td>(?&lt;tag&gt;.*).</td>
<td>The entire key’s name is used for hashing.</td>
</tr>
</tbody>
</table>

You can modify existing rules, add new ones, delete rules, or change their order to suit your application’s requirements.

**Custom hashing policy notes and limitations**

1. You can define up to 32 RegEx rules, each up to 256 characters.
2. RegEx rules are evaluated by their order.
3. The first rule matched is used; strive to place common key name patterns at the beginning of the rule list.

4. Key names that do not match any of the RegEx rules trigger an error.

5. The ‘.*(?<tag>)’ RegEx rule forces keys into a single slot as the hash key is always empty. When used, this should be the last catch-all rule.

6. The following flag is enabled in our regular expression parser:
   - **PCRE_ANCHORED**: the pattern is constrained to match only at the start of the string which is being searched.