Database clustering

Open source Redis is a single-threaded process to provide speed and simplicity. A single Redis process is bound by the CPU core that it is running on and available memory on the server.

Redis Enterprise Software (RS) supports database clustering to allow customers to spread the load of a Redis process over multiple cores and the RAM of multiple servers. A database cluster is a set of Redis processes where each process manages a subset of the database keyspace.

In an RS cluster, the keyspace is partitioned into database shards. At any moment a shard resides on a single node and is managed by that node. Each node in a Redis database cluster can manage multiple shards. The key space in the shards is divided into hash slots. The slot of a key is determined by a hash of the key name or part of the key name.

Database clustering is transparent to the Redis client that connects to the database. The Redis client accesses the database through a single endpoint that automatically routes all operations to the relevant shards. You can connect an application to a single Redis process or a clustered database without any difference in the application logic.

Terminology

In clustering, these terms are commonly used:

- Tag or Hash Tag - A part of the key that is used in the hash calculation.
- Slot or Hash Slot - The result of the hash calculation.
- Shard - Redis process that is part of the Redis clustered database.

When to use clustering (sharding)

Clustering is an efficient way of scaling Redis that should be used when:

- The dataset is large enough to benefit from using the RAM resources of more than one node. When a dataset is more than 25 GB (50 GB for RoF), we recommend that you enable clustering to create multiple shards of the database and spread the data requests across nodes.

- The operations performed against the database are CPU-intensive, resulting in performance degradation. By having multiple CPU cores manage the database’s shards, the load of operations is distributed among them.

Number of shards

When enabling database clustering you can set the number of database shards. The minimum number of shards per database is 2 and the maximum depends on the subscription you purchased.

Once database clustering has been enabled and the number of shards has been set, you cannot disable database clustering or reduce the number of shards. You can only increase the number of shards by a multiple of the current number of shards. For example, if the current number of shards was 3, you can increase to 6, 9, 12 and so on.

Supported hashing policies

Standard hashing policy

When using the standard hashing policy, a clustered database behaves just like a standard, open-source, Redis cluster:

- Keys with a hash tag: a key’s hash tag is any substring between ‘{’ and ‘}’ in the key’s name. That means that when a key’s name includes the pattern ‘{…}’, the hash tag is used as input for the hashing function. For example, the following key names
Keys with a hash tag: when a key does not 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 can be executed on these keys. On the other hand, not using a hash tag 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 do not need to construct key names with hash tags.

Custom hashing policy

A clustered database can be configured to 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. The custom hashing policy is provided through a set of Perl Compatible Regular Expressions (PCRE) rules that describe the dataset’s key name patterns.

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

Once you enable the custom hashing policy, the following default RegEx rules are implemented. Update these rules to fit your specific logic:

<table>
<thead>
<tr>
<th>RegEx Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.*{(?&lt;tag&gt;.*)}.*</code></td>
<td>Hashing is done on the substring between the curly braces.</td>
</tr>
<tr>
<td><code>(?&lt;tag&gt;.*)</code></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 in their order and the first rule matched is used. Therefore, strive to place common key name patterns at the beginning of the rule list.
3. Key names that do not match any of the RegEx rules trigger an error.
4. The ‘.*(?<tag>.)’ RegEx rule forces keys into a single slot because the hash key are always empty. Therefore, when used, this should be the last, catch-all rule.
5. The following flag is enabled in the regular expression parser: PCRE_ANCHORED: the pattern is constrained to match only at the start of the string being searched.

Changing the hashing policy

The hashing policy of a clustered database can be changed. However, most hashing policy changes trigger the deletion (i.e., FLUSHDB) of the data before they can be applied.

Examples of such changes include:

- Changing the hashing policy from standard to custom or conversely, custom to standard.
- Changing the order of custom hashing policy rules.
- Adding new rules in the custom hashing policy.
- Deleting rules from the custom hashing policy.

Note: The recommended workaround for updates that are not enabled, or require flushing the database, is to back up the database and import the data to a newly configured database.
Multi-key operations

Operations on multiple keys in a clustered database are supported with the following limitations:

- **Multi-key commands**: Redis offers several commands that accept multiple keys as arguments. In a clustered database, most multi-key commands are not allowed across slots. The following multi-key commands are allowed across slots: `DEL`, `MSET`, `MGET`, `EXISTS`, `UNLINK`, `TOUCH`.

  In CRBDs, multi-key commands can only be run on keys that are in the same slot.

  Commands that affect all keys or keys that match a specified pattern are allowed in a clustered database, for example: `FLUSHDB`, `FLUSHALL`, `KEYS`.

  | Note: When using these commands in a sharded setup, the command is distributed across multiple shards and the responses from all shards are combined into a single response.

- **Geo commands**: For the `GEORADIUS` and `GEORADIUSBYMEMBER` commands, the `STORE` and `STOREDIST` options can only be used when all affected keys reside in the same slot.

- **Transactions**: All operations within a `WATCH / MULTI / EXEC` block should be performed on keys that are mapped to the same slot.

- **Lua scripts**: All keys used by a Lua script must be mapped to the same slot and must be provided as arguments to the `EVAL / EVALSHA` commands (as per the Redis specification). Using keys in a Lua script that were not provided as arguments might violate the sharding concept but do not result in the proper violation error being returned.

- **Renaming/Copy keys**: The use of the `RENAME / RENAMENX / COPY` commands is allowed only when the key’s original and new values are mapped to the same slot.

Updated: October 8, 2021