

## Back-of-the-envelope estimation for Redis

A "back-of-the-envelope estimation" for Redis typically involves quick calculations to assess hardware, memory requirements, or system scalability based on your use case. Here's a step-by-step approach to estimate Redis usage:

---

### 1. Data Size Per Item

- Calculate the approximate size of a single item (key + value + metadata).
- **Key Size:** Estimate the average key size in bytes.
- **Value Size:** Estimate the average value size in bytes.
- **Metadata Overhead:** Redis adds a small metadata overhead per key-value pair (around 16–64 bytes, depending on the data structure).

**Example:**

- Key: "user:12345" (~10 bytes)
  - Value: A JSON object of 150 bytes.
  - Overhead: ~50 bytes.
  - **Total per item:** ~210 bytes.
- 

### 2. Number of Items

- Estimate the total number of items you expect to store.
  - **Example:** 10 million items.
- 

### 3. Memory Usage

- Multiply the total size per item by the number of items.
  - **Example:**  $210 \text{ bytes/item} \times 10,000,000 \text{ items} = 2.1 \text{ GB}$   
 $210 \text{ bytes/item} \times 10,000,000 \text{ items} = 2.1 \text{ GB}$
- 

### 4. Replication and Persistence Overheads

- Redis often uses replication for high availability:
  - **Replication:** Double the memory for one replica (e.g., primary + replica =  $2 \times \text{memory size}$ ).
- **Persistence** (if using snapshots or AOF): Add ~1.5x memory for backups.

**Example:**

- For replication:  $2.1 \text{ GB} \times 2 = 4.2 \text{ GB}$
- For persistence:  $4.2 \text{ GB} \times 1.5 = 6.3 \text{ GB}$

## 5. Cluster and Sharding

- For large datasets, Redis supports clustering.
- Divide the memory across shards to calculate memory per node.

## 6. Network and CPU

- Redis is lightweight, but network I/O and CPU usage depend on the request rate.
- Estimate based on:
  - **Reads/writes per second.**
  - **Command complexity** (e.g., GET is faster than ZRANGE).

### Final Quick Estimation Example:

- **Data size:** 2.1 GB (single instance).
- **Replication:** 4.2 GB (primary + replica).
- **Persistence:** ~6.3 GB with backups.
- **Nodes:** If clustered, divide 6.3 GB by the number of shards/nodes.

This gives you a starting point for evaluating Redis's requirements for your project. For a more detailed analysis, use monitoring tools (e.g., redis-cli INFO) during a pilot deployment.

## Back-of-the-envelope estimation for Redis shards

When estimating the number of Redis shards, the goal is to distribute your data evenly while ensuring that each shard fits within the constraints of your hardware and operational requirements. Here's how to perform a "back-of-the-envelope" estimation for Redis shards:

### 1. Determine Total Data Size

- Calculate the **total size** of your dataset, including key-value pairs and metadata overhead.
  - Example: If each item averages 210 bytes and you have 100 million items:  
 $\text{Total size} = 210 \text{ bytes/item} \times 100,000,000 \text{ items} = 21 \text{ GB}$

---

## 2. Account for Replication

- Redis typically uses **replication** for high availability. Each replica adds a full copy of the data.
  - Example: With one replica, the total memory requirement becomes:  
$$\text{Total replicated size} = 21 \text{ GB} \times 2 = 42 \text{ GB}.$$

---

## 3. Estimate Shard Size

- **Determine memory limits per shard:**
  - Consider the available memory on your hardware.
  - Leave room for **Redis overhead**, **OS memory**, and **buffering** (Redis recommends using ~50% of available memory).
  - Example: If each server has 32 GB RAM:  
$$\text{Usable memory per shard} \approx 32 \text{ GB} \times 0.5 = 16 \text{ GB}.$$

---

## 4. Calculate the Number of Shards

- Divide the total memory requirement (with replication) by the usable memory per shard:  
$$\text{Number of shards} = \frac{\text{Total replicated size}}{\text{Usable memory per shard}}.$$
  - Example: For 42 GB total replicated size and 16 GB per shard:  
$$\text{Number of shards} = \frac{42}{16} \approx 3 \text{ shards}.$$

---

## 5. Adjust for Future Growth

- Redis handles growth well with clustering, but it's best to overprovision slightly.
- Factor in expected growth (e.g., a 50% increase over the next year):  
$$\text{Adjusted size} = 42 \text{ GB} \times 1.5 = 63 \text{ GB}.$$
  - Recalculate the number of shards: 
$$\text{Number of shards} = \frac{63}{16} \approx 4 \text{ shards}.$$

---

## 6. Evaluate Shard Distribution

- Ensure each shard handles a similar load (uniform hash partitioning).
- Monitor shard performance to ensure no shard is overloaded.

---

### Final Example

- **Dataset size:** 21 GB (raw data).
- **With replication:** 42 GB.
- **Shard memory limit:** 16 GB (on 32 GB servers).
- **Shards required:** 3, or 4 with growth projection.

This provides a rough but actionable estimate to plan your Redis cluster. Fine-tuning is necessary during deployment to account for specific workloads and query patterns.

40

## Back-of-envelope estimation for the number of nodes in Redis shards

To estimate the number of **Redis nodes** for a sharded setup, consider the data size, replication, and memory capacity of each node. Here's how you can approach this calculation:

---

### 1. Estimate Total Memory Requirement

#### Data size per shard:

- Calculate the **raw dataset size**.
  - Example: 100 million items at 210 bytes/item:  
 $\text{Dataset size} = 100,000,000 \times 210 = 21 \text{ GB}$ .  
 $\text{Dataset size} = 100,000,000 \times 210 = 21 \text{ GB}$ .

#### Replication:

- Factor in replication for high availability.
  - With one replica:  $\text{Total memory (with replication)} = 21 \text{ GB} \times 2 = 42 \text{ GB}$ .

#### Buffer for overhead:

- Redis typically needs ~20–30% extra memory for internal overhead.  
 $\text{Total usable memory required} = 42 \text{ GB} \times 1.3 = 54.6 \text{ GB}$ .

---

### 2. Determine Memory Capacity Per Node

- Use only ~50–60% of the total physical memory for Redis to allow room for:

- OS processes.
- Buffering and overhead.
- Caching and bursts.

**Example:** On a 32 GB RAM node:

Usable memory per node =  $32 \text{ GB} \times 0.6 = 19.2 \text{ GB}$ .  
 $\text{Usable memory per node} = 32 \text{ GB} \times 0.6 = 19.2 \text{ GB}$ .

---

### 3. Calculate the Number of Nodes

- Divide the total memory requirement by the usable memory per node:  

$$\text{Number of nodes (per replica set)} = \frac{\text{Total usable memory required}}{\text{Usable memory per node}}$$

$$\text{Number of nodes (per replica set)} = \frac{54.6 \text{ GB}}{19.2 \text{ GB}} \approx 2.84 \approx 3 \text{ nodes}$$

**Example:** For 54.6 GB total memory and 19.2 GB usable per node:

Number of nodes (per replica set) =  $\frac{54.6}{19.2} \approx 2.84 \approx 3$  nodes.  
 $\text{Number of nodes (per replica set)} = \frac{54.6}{19.2} \approx 2.84 \approx 3 \text{ nodes}$ .

**Including Replicas:**

- Each replica set includes  $N + R \times N$  nodes, where  $N$  is the number of primary shards and  $R$  is the replication factor.

**Example:** With 1 replica:

Total nodes =  $3 \text{ (primaries)} \times 2 \text{ (primary + replica)} = 6 \text{ nodes}$ .  
 $\text{Total nodes} = 3 \times 2 = 6 \text{ nodes}$ .

---

### 4. Adjust for Scaling

- Consider **future growth** in data size and load (e.g., 50% increase):  

$$\text{Future total memory required} = 54.6 \text{ GB} \times 1.5 = 81.9 \text{ GB}$$

$$\text{Future total memory required} = 54.6 \text{ GB} \times 1.5 = 81.9 \text{ GB}$$
    - Recalculate the number of nodes:  

$$\text{Future nodes (per replica set)} = \frac{81.9}{19.2} \approx 4.26 \approx 5 \text{ nodes}$$

$$\text{Future nodes (per replica set)} = \frac{81.9}{19.2} \approx 4.26 \approx 5 \text{ nodes}$$
      - With replication:  $5 \times 2 = 10 \text{ nodes}$ .  $5 \times 2 = 10 \text{ nodes}$ .
- 

**Final Example**

- **Dataset size:** 21 GB.
- **With replication:** 42 GB.
- **Memory overhead:** 30% → 54.6 GB.
- **Usable memory per node:** 19.2 GB (on 32 GB RAM nodes).
- **Nodes required:** 3 primary nodes + 3 replicas = **6 nodes**.
- **With growth:** Up to **10 nodes** (including replicas).

This estimation ensures that your cluster is adequately provisioned for both current needs and anticipated growth.

4o

**You said:**

back of the envelope estimation for redis scaling for IOPS

**ChatGPT said:**

ChatGPT

Estimating Redis scaling for IOPS (Input/Output Operations Per Second) involves understanding your workload, traffic patterns, and hardware capabilities. Here's a structured back-of-the-envelope approach:

---

### 1. Understand the Workload

- **Request Types:** Redis operations vary in complexity; simple GET/SET operations are lightweight, while commands like ZRANGE or MGET are more resource-intensive.
- **Read vs. Write Ratio:** Determine the ratio of reads to writes. For example:
  - Reads: 80%
  - Writes: 20%
- **Operation Size:** Average size of keys, values, and responses. This affects the network and memory bandwidth.

---

### 2. Estimate IOPS Per Node

- Redis performance is heavily dependent on CPU and memory:
    - Modern Redis nodes can handle **80,000–100,000 IOPS** for simple commands on a well-provisioned server.
    - Complex commands reduce IOPS capacity.
  - **Example:** Assume a node can handle 100,000 IOPS for simple GET/SET operations.
-

### 3. Calculate Required IOPS

- Estimate the total number of operations per second based on your workload:
    - **Example Workload:**
      - 10 million daily operations.
      - Peak traffic is 10x the average.
      - **Peak Operations per Second (OPS):**  
$$\text{Peak OPS} = \frac{\text{Total daily operations}}{\text{seconds in a day}} \times \text{Peak multiplier}$$
$$\text{Peak OPS} = \frac{10,000,000}{86,400} \times 10 \approx 1,160 \text{ OPS}$$
  - **Growth Consideration:**
    - Plan for 5x growth:  $\text{Future Peak OPS} = 1,160 \times 5 = 5,800 \text{ OPS}$
- 

### 4. Determine Nodes for IOPS

- Divide the total required IOPS by the per-node capacity:
  - For **simple commands**, each node can handle 100,000 IOPS:  
$$\text{Nodes required for IOPS} = \frac{\text{Peak OPS}}{\text{Per-node IOPS}}$$
$$\text{Nodes required for IOPS} = \frac{1,160}{100,000} \approx 1 \text{ node}$$

#### Example:

- Nodes required for current peak = 1,160 / 100,000 ≈ 1 node
  - Nodes required for future peak = 5,800 / 100,000 ≈ 1 node
- 

### 5. Include Replication Overheads

- Replication doubles the operations each node must process:

Effective IOPS per node (with replication) =  $\frac{\text{Per-node IOPS}}{(1 + \text{replication factor})}$

Effective IOPS per node (with replication) =  $\frac{100,000}{(1 + 1)} = 50,000$

- With 1 replica: Effective IOPS per node =  $\frac{100,000}{2} = 50,000$ . Effective IOPS per node =  $\frac{2100,000}{50,000} = 42$ .

#### Revised Nodes:

- Nodes for current peak =  $\frac{1,160}{50,000} \approx 0.0232$  node. Nodes for current peak =  $\frac{50,000}{50,000} = 1$  node.
- Nodes for future peak =  $\frac{5,800}{50,000} \approx 0.116$  node. Nodes for future peak =  $\frac{50,000}{50,000} = 1$  node.

## 6. Factor in Complexity and Growth

- For more complex commands, reduce IOPS/node by 50% or more:
  - Adjusted capacity: 50,000 IOPS for complex workloads.
- Adjust further if high latency is unacceptable or for unpredictable spikes.

#### Final Example

- **Current Peak:** 1,160 OPS → 1 node required (simple commands).
- **Future Peak:** 5,800 OPS → 1–2 nodes required (depending on workload complexity).
- **Replication:** With 1 replica, use **2–4 nodes** to handle scaling and redundancy.

This estimation ensures enough headroom for both IOPS scaling and fault tolerance.

40

## Redis shard and scaling back of the envelope estimation

We combine calculations for data distribution and IOPS capacity to perform a back-of-the-envelope estimation for Redis shard and scaling requirements. Here's a step-by-step process:

### 1. Estimate Data Storage Per Shard

#### Total Dataset Size:

- Compute the raw size of your data.  

$$\text{Total data size} = \text{number of items} \times (\text{key size} + \text{value size} + \text{metadata overhead})$$

$$\text{Total data size} = \text{number of items} \times (\text{key size} + \text{value size} + \text{metadata overhead})$$

#### Example:

- Number of items: 100 million.
- Average key + value + metadata size: 210 bytes.



Total data size=100,000,000×210=21 GB.  $\text{Total data size} = 100,000,000 \times 210 = 21 \text{ GB}$ .  
Total data size=100,000,000×210=21GB.

#### Replication:

- Add a replication factor (e.g., 1 replica):  
Total replicated size=Total data size×(1+replica factor)  $\text{Total replicated size} = \text{Total data size} \times (1 + \text{replica factor})$   
Total replicated size=Total data size×(1+replica factor) **Example:**  
Total replicated size=21 GB×2=42 GB.  $\text{Total replicated size} = 21 \text{ GB} \times 2 = 42 \text{ GB}$ .  
Total replicated size=21GB×2=42GB.

#### Per Shard Storage:

- Determine usable memory per shard:
  - Usable memory = Node RAM×0.6  $\text{Node RAM} \times 0.6$   
(reserve 40% for OS, overhead, and safety). **Example:** For a node with 32 GB RAM:

Usable memory per shard=32 GB×0.6=19.2 GB.  $\text{Usable memory per shard} = 32 \text{ GB} \times 0.6 = 19.2 \text{ GB}$ .  
Usable memory per shard=32GB×0.6=19.2GB.

- Calculate the number of shards for storage:

Number of shards (storage)=Total replicated sizeUsable memory per shard.  $\text{Number of shards (storage)} = \frac{\text{Total replicated size}}{\text{Usable memory per shard}}$   
Number of shards (storage)=Usable memory per shardTotal replicated size.

#### Example:

Number of shards (storage)=4219.2≈3 shards.  $\text{Number of shards (storage)} = \frac{42}{19.2} \approx 3 \text{ shards}$ .  
Number of shards (storage)=19.242≈3shards.

---

## 2. Estimate IOPS Per Shard

#### Workload Analysis:

- Identify peak operations per second (OPS):
  - Total daily operations: 50 million.
  - Peak multiplier: 10x average.
  - Seconds in a day: 86,400.

Peak OPS=Total daily operationsSeconds per day×Peak multiplier.  $\text{Peak OPS} = \frac{\text{Total daily operations}}{\text{Seconds per day}} \times \text{Peak multiplier}$   
Peak OPS=Seconds per dayTotal daily operations×Peak multiplier. **Example:**  
Peak OPS=50,000,00086,400×10≈5,800 OPS.  $\text{Peak OPS} = \frac{50,000,000}{86,400} \times 10 \approx 5,800 \text{ OPS}$ .  
Peak OPS=86,40050,000,000×10≈5,800OPS.

#### Per Node IOPS Capacity:

- A single Redis node can handle:

- **Simple commands:** ~100,000 IOPS.
- **Complex commands:** ~50,000 IOPS.

#### Shards for IOPS:

- Calculate the number of shards required to meet peak IOPS:  

$$\text{Number of shards (IOPS)} = \frac{\text{Peak OPS}}{\text{Per-node IOPS capacity}}$$

$$\text{Number of shards (IOPS)} = \frac{\text{Peak OPS}}{\text{Per-node IOPS capacity}}$$
**Example:**
  - For simple commands:  

$$\text{Number of shards (IOPS)} = \frac{5,800}{100,000} \approx 1 \text{ shard}$$

$$\text{Number of shards (IOPS)} = \frac{5,800}{100,000} \approx 1 \text{ shard}$$

### 3. Combine Storage and IOPS Requirements

- Take the maximum of the two estimates to ensure adequate capacity: **Example:**
  - Number of shards for storage: 3.
  - Number of shards for IOPS: 1.

Required shards =  $\max(3, 1) = 3$  shards.  

$$\text{Required shards} = \max(3, 1) = 3$$

### 4. Adjust for Replication

- Each shard has  $1 + R$  nodes, where  $R$  is the replication factor.
  - With 1 replica: Total nodes = Shards  $\times (1 + R)$ .  

$$\text{Total nodes} = \text{Shards} \times (1 + R)$$

**Example:** Total nodes =  $3 \times 2 = 6$  nodes.  

$$\text{Total nodes} = 3 \times 2 = 6$$

### 5. Adjust for Growth

- Factor in expected data growth (e.g., 2x in the next year):
  - Future total replicated size:

Future size = Current size  $\times$  Growth factor.  

$$\text{Future size} = \text{Current size} \times \text{Growth factor}$$

#### Example:

Future size =  $42 \text{ GB} \times 2 = 84 \text{ GB}$ .  

$$\text{Future size} = 42 \text{ GB} \times 2 = 84 \text{ GB}$$

- Recalculate required shards and nodes:

- Future shards for storage:  $8419.2 \approx 5 \text{ shards}$   $\frac{84}{19.2} \approx 5$ ,  $\text{shards}$   $19.284 \approx 5 \text{ shards}$ .
- Total nodes with replication:  $5 \times 2 = 10 \text{ nodes}$ .  $5 \times 2 = 10$ ,  $\text{nodes}$ .  $5 \times 2 = 10 \text{ nodes}$ .

---

### Final Example

- **Current Dataset:** 21 GB, with 42 GB replicated.
- **Current Shards:** 3 shards (storage-limited).
- **Total Nodes:** 6 nodes (with 1 replica).
- **Future Growth:** Plan for 5 shards and 10 nodes to handle scaling.

This estimate provides a solid starting point for planning Redis shards and scaling requirements!

40

## Back of the envelope estimation for connection pool of Redis nodes and shard

Estimating the size of the **connection pool** for Redis nodes and shards involves analyzing the number of concurrent clients, their behavior, and the number of Redis shards and replicas. Here's a step-by-step approach:

---

### 1. Understand Connection Pool Basics

- Each Redis client requires a TCP connection to a specific Redis node.
- A connection pool holds reusable client connections, reducing overhead from creating/destroying connections for every request.

---

### 2. Identify Key Metrics

#### Total Number of Concurrent Clients:

- Estimate the **peak number of concurrent clients**:
  - Example: 50,000 users, with 10% concurrently active.

Concurrent clients =  $50,000 \times 0.1 = 5,000 \text{ clients}$ .  $\text{Concurrent clients} = 50,000 \times 0.1 = 5,000$ ,  $\text{clients}$ .  $\text{Concurrent clients} = 50,000 \times 0.1 = 5,000 \text{ clients}$ .

#### Connection Pool Size Per Client:

- Each client may maintain a pool of connections to Redis to:
  - Handle parallel requests.
  - Reduce latency.

Typical values:

- Simple workloads: 555 connections per client.
- Heavy workloads: 10+10+10+ connections per client.

**Example:**

Total connections per client pool=5.  
 $\text{Total connections per client pool} = 5.$

**Peak Requests Per Second (RPS):**

- Redis processes are lightweight, but connection pools must support peak RPS.
    - Example: 10,000 RPS.
- 

### 3. Account for Shards

Each Redis shard is an independent database, requiring its own connections. Clients must maintain separate pools for each shard.

- **Example:**
  - Number of shards: 555.
  - Connections per shard per client pool: 555.

Total connections per client pool across shards:

Connections per client (all shards)=Connections per shard×Number of shards.  
 $\text{Connections per client (all shards)} = \text{Connections per shard} \times \text{Number of shards}.$

**Example:**

Connections per client (all shards)=5×5=25.  
 $\text{Connections per client (all shards)} = 5 \times 5 = 25.$

---

### 4. Adjust for Replication

If Redis uses replication:

- Each shard has 1+R1 + R1+R nodes (primary + replicas).
- Connections must be distributed across all nodes for read and write traffic.

**Example:**

- Number of shards: 555.
- Replicas: 111.
- Total nodes:  $5 \times 2 = 105 \times 2 = 105 \times 2 = 10.$
- Connections per node: 555 per client pool.

---

## 5. Calculate Total Connections

The total number of connections is:

$\text{Total connections} = \text{Concurrent clients} \times \text{Connections per client (all shards)}$   
 $\text{Total connections} = \text{Concurrent clients} \times \text{Connections per client (all shards)}$

### Example Calculation:

- Concurrent clients: 5,000
- Connections per client (all shards): 25

$\text{Total connections} = 5,000 \times 25 = 125,000$   
 $\text{Total connections} = 5,000 \times 25 = 125,000$

---

## 6. Per Node Connections

Distribute the connections across nodes:

$\text{Connections per node} = \frac{\text{Total connections}}{\text{Total nodes}}$   
 $\text{Connections per node} = \frac{\text{Total connections}}{\text{Total nodes}}$

### Example:

- Total nodes: 10
- Total connections: 125,000

$\text{Connections per node} = \frac{125,000}{10} = 12,500$   
 $\text{Connections per node} = \frac{125,000}{10} = 12,500$

---

## 7. Consider Scaling and Overheads

- Redis nodes can typically handle 10,000–50,000 connections per node (depending on hardware and workload).
- For large-scale systems:
  - Scale horizontally by adding shards or nodes.
  - Use **proxy layers** (e.g., Redis Sentinel or Redis Cluster clients) to optimize connection usage.

---

### Final Example

- **Concurrent Clients:** 5,000.
- **Connections per Client Pool (all shards):** 25.

- **Total Connections:** 125,000.
- **Nodes:** 10.
- **Connections per Node:** 12,500.

Ensure your nodes and network are provisioned to handle the calculated connection load.

## Maximum number of connections in Redis per node

The **maximum number of connections per Redis node** depends on several factors, including Redis configuration, server hardware, operating system limits, and workload characteristics. Here's how to analyze it:

---

### 1. Default Maximum Connections

Redis has a configuration parameter, `max clients`, which defines the maximum number of simultaneous client connections a Redis instance can handle.

#### Defaults:

- **Default value:** 10,000.
- This value includes connections for replicas, monitoring tools, and clients.

If `max clients` is reached, new connections are rejected until existing ones are closed.

---

### 2. Practical Connection Limits

#### Redis Limits:

Redis is designed to handle tens of thousands of simultaneous connections, but practical limits depend on:

#### 1. Memory Usage:

- Each connection consumes a small amount of memory (~1 KB per connection for overhead).
- For 10,000 connections, this would be ~10 MB of memory.

#### 2. CPU Usage:

- Redis is single-threaded for most operations, so a high number of connections with heavy workloads can saturate the CPU.

#### 3. Latency:

- Many idle connections (e.g., in connection pools) are fine, but active connections can increase latency.

#### Operating System Limits:

The operating system may impose limits on the number of file descriptors (sockets) Redis can open.

- **Linux Default:** 1,024 file descriptors per process.
- Increase this with the `ulimit` command or by modifying `/etc/security/limits.conf`.

#### Example Tuning:

- Increase the file descriptor limit:

```
ulimit -n 100000
```

- Adjust `maxclients` in `redis.conf`:

```
maxclients 50000
```

---

### 3. Hardware Considerations

- Redis can handle **50,000+ connections** on modern servers, but:
  - **Memory:** Ensure enough memory for connection overheads + dataset.
  - **Network Bandwidth:** High connection counts may saturate network capacity.
  - **CPU:** More connections require more CPU for managing I/O.

#### Example:

On a server with:

- 16 GB RAM,
- A 1 Gbps network interface,
- 8-core CPU,

Redis can comfortably handle **50,000–100,000 mostly idle connections** or **10,000–20,000 active connections** under typical workloads.

---

### 4. Scaling Connections

If a single Redis instance cannot handle the required connections:

1. **Add Redis Nodes:**
  - Use **Redis Cluster** or **Sharding** to distribute connections across multiple nodes.
2. **Use a Proxy:**
  - Tools like **Twemproxy** or **Envoy** can multiplex client connections, reducing the load on Redis.
3. **Connection Pooling:**

- Use connection pooling libraries in clients to reduce the number of simultaneous connections.
- 

## 5. Theoretical Upper Limit

With proper tuning:

- **Maximum connections:** Redis can handle **hundreds of thousands** of connections if:
  - Idle connections dominate.
  - Sufficient memory and file descriptors are configured.

**Real-world practical limits:**

- ~50,000 **active connections** per node in high-throughput scenarios, depending on workload and hardware.

## Redis ElastiCache RCU WCU

In **Amazon ElastiCache for Redis**, the concepts of **RCU** (Read Capacity Unit) and **WCU** (Write Capacity Unit) are not explicitly defined as they are in DynamoDB. However, these can be loosely estimated based on **read and write throughput requirements** for Redis workloads. Redis operates in-memory, so its performance is typically constrained by CPU, network, and node configuration.

Here's how to think about **RCU** and **WCU** in the context of ElastiCache for Redis:

---

### 1. Redis Read Capacity Unit (RCU)

An **RCU** for Redis corresponds to the throughput required for **read operations** (e.g., GET, MGET, or SCAN commands).

**Key Factors:**

1. **Query Type:**
  - Simple GET commands are lightweight and fast.
  - Complex reads like SORT, SCAN, or fetching large data structures (e.g., hashes, lists) consume more resources.
2. **Item Size:**
  - Larger objects consume more bandwidth per read operation.
3. **Read Operations Per Second (RPS):**
  - Peak RPS determines the overall read capacity.

**Estimation:**



- Each GET operation is ~0.1–0.3 KB in overhead (depending on key size and protocol).
- A single node (e.g., cache.r6g.large) can handle:
  - **~90,000 GET requests/second** for small keys/values (~1 KB).
  - Higher-end instances (e.g., r6g.4xlarge) can handle **~400,000+ GETs/sec**.

#### Approximation:

1 **RCU** could be defined as:

- 1 simple read operation (e.g., GET) of 1 KB per second.

For a workload of **50,000 RPS (simple GETs)**:

$\text{RCUs} = 50,000$ .

---

## 2. Redis Write Capacity Unit (WCU)

A **WCU** for Redis corresponds to the throughput required for **write operations** (e.g., SET, HSET, INCR, or R PUSH commands).

#### Key Factors:

1. **Command Type:**
  - Simple commands like SET are lightweight.
  - Commands involving multiple keys or large structures (e.g., LPUSH, ZADD, or MULTI/EXEC transactions) consume more resources.
2. **Data Size:**
  - Larger values consume more bandwidth and memory during writes.
3. **Write Operations Per Second (WPS):**
  - Peak WPS determines the overall write capacity.

#### Estimation:

- A single Redis node (e.g., cache.r6g.large) can handle:
  - **~70,000 SET operations/second** for small values (~1 KB).
  - Bulk writes (e.g., pipelines) improve throughput significantly.

#### Approximation:

1 **WCU** could be defined as:

- 1 simple write operation (e.g., SET) of 1 KB per second.

For a workload of **10,000 WPS (simple SETs)**:

$\text{WCUs} = 10,000$ .

---

### 3. Scaling RCUs and WCUs with ElastiCache

#### Scaling Reads:

##### 1. Read Replicas:

- Redis supports **replication**. Each read replica adds capacity for RCUs.
- With RRR read replicas, total read capacity =  $(1+R) \times \text{RCUs per node}$

##### 2. Cluster Mode:

- In **cluster mode**, Redis shards distribute keys across multiple nodes, increasing both read and write capacity.

#### Scaling Writes:

##### 1. Cluster Mode:

- Write capacity scales with the number of **primary shards**.
- Each shard handles an independent subset of keys.

##### 2. Node Type:

- Higher-tier nodes provide more memory, CPU, and network bandwidth, increasing WCUs.
- 

### 4. Practical Example

For a workload with:

- **Reads:** 20,000 RPS (1 KB objects, simple GETs).
- **Writes:** 5,000 WPS (2 KB objects, simple SETs).
- Replication Factor = 1 (1 replica per shard).

#### Step 1: Estimate RCUs and WCUs

- **RCUs:** 20,000
- **WCUs:**  $5,000 \times 2 = 10,000$

#### Step 2: Node Throughput

- Assume a cache.r6g.large node can handle:
  - 90,000 RCUs.
  - 70,000 WCUs.

#### Step 3: Shard Configuration

##### 1. Read Scaling:

- With replication factor  $R=1$ , each shard provides  $90,000 \times 2 = 180,000$  RCUs.
- Sufficient for 20,000 RCUs.

## 2. Write Scaling:

- Each primary node provides 70,000 WCUs.
- Sufficient for 10,000 WCUs.

## Final Configuration:

- **Shards:** 1 shard is sufficient for the given RCU/WCU requirements.
- **Nodes:** 1 primary + 1 replica = **2 nodes**.

## 5. Elasticity and Scaling Tips

### 1. Monitor Performance:

- Use **CloudWatch metrics** to track:
  - CPU usage (should stay below 80%).
  - Network throughput.
  - Replica synchronization lag.

### 2. Plan for Growth:

- Anticipate increases in RCU/WCU by adding shards or upgrading instance types.

### 3. Use Pipelines:

- Batch commands to maximize throughput and reduce per-operation overhead.

By estimating RCUs and WCUs this way, you can design a scalable and cost-efficient ElastiCache for Redis setup.

## Amazon ElastiCache configurations estimation

When estimating **Amazon ElastiCache** configurations, particularly for **Redis**, you need to consider factors such as throughput requirements, memory capacity, redundancy, and scaling strategies. Here's a step-by-step guide to help you estimate the appropriate configuration for your workload:

### 1. Determine Your Workload Requirements

#### a. Read Throughput (RCUs)

Estimate how many **read operations** your application needs to perform per second:

- **GET commands:** The most common read operation.
- **Read-heavy workload:** Higher read throughput (e.g., frequent cache lookups).

Factors to consider:

- **QPS (Queries Per Second).**
- **Command complexity** (simple GET vs. complex queries or aggregations).
- **Item size:** Larger objects will consume more memory and bandwidth.

## b. Write Throughput (WCUs)

Estimate how many **write operations** your application will perform per second:

- **SET commands:** The most common write operation.
- **Write-heavy workload:** Frequent updates to the cache.
- **Bulk writes:** For workloads that need batch processing.

Factors to consider:

- **WPS (Writes Per Second).**
- **Command complexity** (simple SET vs. complex HSET, RPush).
- **Item size:** Larger writes consume more resources.

## c. Memory Requirements

Redis is an in-memory data store, so the amount of data you store in Redis determines the memory requirements. Estimate:

- **Average object size.**
- **Total number of keys.**
- **Data structure** complexity (e.g., strings, hashes, lists, sorted sets).

## 2. ElastiCache Instance Types

Choose the appropriate **ElastiCache instance type** based on the required throughput and memory:

- **Memory Optimized Instances** (e.g., r6g, r5, m6g): Better suited for Redis as they provide more memory and better performance for memory-intensive workloads.
- **Compute Optimized Instances** (e.g., c6g): Suitable for compute-bound workloads but typically not ideal for Redis since it's memory-bound.

Here are common instance types for Redis:

- cache.t3.micro, cache.t3.small — for low traffic, small-scale deployments.
- cache.r6g.large, cache.r6g.xlarge — for medium to large traffic, high memory needs.
- cache.r6g.2xlarge, cache.r6g.4xlarge — for heavy, high-throughput applications.

### 3. Estimating Redis Memory Requirements

For memory estimation, calculate the total size of data stored:

- **Total Memory Needed:**

$\text{Total Memory} = \text{Average Object Size} \times \text{Total Number of Objects}$ .  
 $\text{Total Memory} = \text{Average Object Size} \times \text{Total Number of Objects}$ .

If you expect to store 1,000,000 keys, each of size 1 KB:

$\text{Total Memory} = 1,000,000 \times 1 \text{ KB} = 1,000,000 \text{ KB} = 1 \text{ GB}$ .  
 $\text{Total Memory} = 1,000,000 \times 1 \text{ KB} = 1,000,000 \text{ KB} = 1 \text{ GB}$ .

Ensure that your ElastiCache instance type has enough RAM for this data plus some overhead for Redis operations, indexes, and replication.

---

### 4. ElastiCache Cluster Mode vs. Non-Cluster Mode

- **Non-Cluster Mode:** All keys are stored in a single shard. Useful for smaller datasets or when sharding is not necessary.
- **Cluster Mode:** Data is distributed across multiple shards. Ideal for larger datasets or higher scalability needs.

#### Cluster Mode Scaling:

- **Number of Shards:** Decide how many primary shards you need based on memory and throughput needs.
  - Each shard will have its own memory, CPU, and IOPS.
  - Redis distributes keys across these shards.

If your total dataset size exceeds the memory of a single node, you'll need more shards.

---

### 5. ElastiCache Replication and High Availability

- **Replication:** Use **1 or more replicas** per primary node to ensure high availability and increased read throughput.
  - **Replication Factor:** Typically, you'll have one replica per shard to balance the load and provide failover capabilities.
  - If you have **3 shards**, with a **replication factor of 1**, you would need **6 nodes** in total (3 primary and 3 replica nodes).
- **Automatic Failover:** If using replication, enable **automatic failover** to automatically promote replicas to primary if a failure occurs.

---

### 6. Estimating ElastiCache Configuration

Based on your workload estimates, here's a **back-of-the-envelope** example:

#### Example 1: Low Traffic Redis Setup

- **Estimated read throughput:** 10,000 GETs per second.
- **Estimated write throughput:** 2,000 SETs per second.
- **Total memory required:** 10 GB for data storage.
- **Instance type:** cache.t3.medium (2 vCPUs, 8 GB RAM).
- **Replication factor:** 1 (1 replica per shard).
- **Cluster mode:** Non-clustered, as the dataset is relatively small.

#### Example 2: High Traffic Redis Setup

- **Estimated read throughput:** 100,000 GETs per second.
- **Estimated write throughput:** 30,000 SETs per second.
- **Total memory required:** 100 GB for data storage.
- **Instance type:** cache.r6g.2xlarge (8 vCPUs, 64 GB RAM).
- **Replication factor:** 1 (1 replica per shard).
- **Cluster mode:** Clustered, 5 primary shards to distribute data.

#### Example 3: Very High Traffic with Bulk Writes

- **Estimated read throughput:** 500,000 GETs per second.
- **Estimated write throughput:** 100,000 SETs per second.
- **Total memory required:** 500 GB for data storage.
- **Instance type:** cache.r6g.12xlarge (48 vCPUs, 384 GB RAM).
- **Replication factor:** 1 (1 replica per shard).
- **Cluster mode:** Clustered, 15 primary shards to distribute data.

---

## 7. Monitor and Optimize

Once your ElastiCache cluster is up and running:

- Use **Amazon CloudWatch** metrics to monitor performance.
  - Key metrics: CPU utilization, memory usage, network throughput, read/write operations per second.
- Adjust the configuration based on actual usage:
  - Scale vertically by switching to a higher instance type if CPU or memory is maxed out.
  - Scale horizontally by adding more shards or replicas as needed.

- Enable **auto discovery** in clients to manage dynamic scaling and failover automatically.
- 

### Final Configuration Example for ElastiCache

For a workload with:

- **50,000 GETs per second.**
- **10,000 SETs per second.**
- **50 GB data.**

### Recommended Setup:

1. **Instance Type:** cache.r6g.xlarge (4 vCPUs, 32 GB RAM).
2. **Cluster Mode:** Enable (5 primary shards to distribute load).
3. **Replication:** 1 replica per shard (5 replicas in total).
4. **Total Nodes:** 10 nodes (5 primary + 5 replica).
5. **Monitoring:** Set up CloudWatch alarms for CPU, memory, and network metrics.

This setup ensures you have enough memory, CPU, and throughput to handle your read and write operations efficiently, with built-in redundancy and failover.