Detect and terminate long-running queries

Long-running commands in Redis can block the single-threaded event loop, causing delayed responses or complete unresponsiveness in production environments like Elestio. Monitoring and handling these commands is critical for maintaining performance and reliability. This guide explains how to detect, analyze, and terminate blocking or slow commands in Redis using terminal tools, Docker Compose setups, and Redis's built-in logging features. It also includes prevention strategies to avoid performance bottlenecks in the future.

Monitoring Long-Running Commands

Redis does not support multitasking like traditional SQL databases, so any command that takes too long blocks the entire server. To inspect active commands and see which clients may be running long operations, use the Redis CLI.

Check active clients and their current commands

```
redis-cli -h <host> -p <port> -a <password> CLIENT LIST
```

This command shows all connected clients, including their IP address, command in progress (cmd), idle time, and total duration. Focus on clients with high idle or age values while still actively running commands.

Detect current command load using MONITOR

To observe commands in real time:

```
redis-cli -a <password> MONITOR
```

This outputs every operation in real time. It's useful for spotting blocking commands but should be used only in staging or during short troubleshooting sessions, as it consumes significant CPU.

Terminating Problematic Commands Safely

Redis provides tools to close problematic connections or interrupt Lua scripts that run for too long.

Kill a specific client connection

If a client is running a blocking or long operation, you can terminate its connection using its client ID:

```
CLIENT KILL ID <id>
```

> You can find the <id> from the CLIENT LIST command.

This will drop the connection and stop any running command associated with that client.

Stop a long-running Lua script

If a Lua script is stuck or taking too long:

```
SCRIPT KILL
```

This stops the currently executing script. If the script has modified data, Redis will return an error to avoid leaving the database in an inconsistent state.

> If the script is not killable (e.g., during a write operation), Redis will return an error. Always use SCRIPT KILL cautiously.

Managing Inside Docker Compose

If your Redis service is running inside a Docker Compose setup on Elestio, you'll need to access the container before you can inspect or kill commands.

Access the Redis container

```
docker-compose exec redis bash
```

Inside the container, connect to Redis using:

```
redis-cli -a $REDIS_PASSWORD
```

Then, use CLIENT LIST, SCRIPT KILL, or CLIENT KILL just like from the host.

Using the Redis Slowlog Feature

Redis includes a built-in slowlog that logs commands that exceed a specific execution threshold.

Enable and configure slowlog in redis.conf

```
slowlog-log-slower-than 10000        # Log commands slower than 10ms
slowlog-max-len 128                  # Keep 128 slow entries
```

Update these settings in redis.conf, or set them at runtime:

```
CONFIG SET slowlog-log-slower-than 10000
CONFIG SET slowlog-max-len 128
```

View the slowlog

```
SLOWLOG GET 10
```

This shows the 10 most recent slow commands with their timestamp, execution time, and command details.

Clear the slowlog

```
SLOWLOG RESET
```

Use this to reset the log after reviewing or during maintenance.

Analyzing Command Latency Over Time

Redis includes latency tracking features to help you understand when and why delays occur.

Generate a diagnostic latency report

```
LATENCY DOCTOR
```

This gives you a summary of observed latency spikes and their causes (e.g., command execution, AOF rewrite, background saves).

View detailed latency history by event

```
LATENCY HISTORY command
```

You can replace command with any tracked event like fork, aof-write, or expire-cycle.

Best Practices to Prevent Long-Running Commands

Preventing long-running commands is critical since Redis handles all operations on a single thread.

- **Avoid full key scans**: Never use KEYS * or SMEMBERS on large sets in production. Use SCAN instead for incremental iteration.
- **Limit Lua script duration**: Break complex scripts into smaller steps and test for performance in staging.
- **Use pipelining**: Send multiple commands in one round-trip to reduce overall time spent per operation.
- **Limit list and set access**: Use ranges or batch operations for large data structures.

```
LRANGE mylist 0 99   # Good
LRANGE mylist 0 -1   # Risky on large lists
```

- **Enable eviction policies**: To avoid OOM errors that can freeze Redis, enable LRU or LFU eviction:

```
CONFIG SET maxmemory-policy allkeys-lru
```

- **Monitor regularly**: Use CLIENT LIST, SLOWLOG, and LATENCY in combination to detect problematic patterns early.

Revision #1
Created 2025-05-20 10:26:37 UTC
Updated 2025-05-20 10:50:03 UTC