Understanding the Top 5 Redis Performance Metrics

Using critical Redis metrics to troubleshoot performance issues

Conor Branagan, Software Engineer, Datadog
Patrick Crosby, Product Marketer, Datadog

Introduction

Yesterday's web was mostly populating static pages with contents from relational databases. Today's web is all about near real-time interactions and low latency. The demands for real-time and the constraints of computer architectures have dictated the need to put in-memory data stores at the center of modern applications.

When you need low-latency, persistent, cross-platform storage no other system is as widely used as Redis. The popularity of Redis stems from its rich API and its strength in providing low-latency while offering persistence usually seen in traditional relational databases. Redis has many prominent users including Yahoo!, Twitter, Craigslist, Stack Overflow and Flickr[1].

The objective of this guide is to explain the five most important metrics that will allow you to more effectively use Redis. Understanding these metrics will help when troubleshooting common issues. In this guide, you will learn what these metrics are, how to access them, how to interpret their output, and how to use them to improve your Redis performance.

The first few pages describe what Redis is, why developers use it, and some of the common alternatives. You can jump straight into the top 5 Redis performance metrics on page 7.

Redis: A fast, easy-to-use key-value store

Redis is an open source, in-memory advanced key-value store with optional persistence to disk. Two common uses for Redis are caching[2] and publish-subscribe queues (Pub/Sub)[3]. When a user logs into a web application, rather than having to query a database each time user information is needed, Redis allows developers to store user attributes in-memory enabling much faster data retrieval. For Pub/Sub, Redis offers primitives to implement the publish-subscribe messaging model with no effort.

Key-value stores: Faster, smaller NoSQL databases

Key-value data consist of two objects: 1) a string which represents the key and 2) the actual data associated with that key. For example, you can set the key "`lastname_2078`" to the value "`Miller`" or the key "`boroughs_of_new_york`" to the value "`bronx, brooklyn, manhattan, queens, staten_island`". Key-value stores have in short time become essential to quickly get an application in front of users by removing the up-front need for carefully designed, structured data tables.

---

[1] http://redis.io/topics/whos-using-redis
[2] https://en.wikipedia.org/wiki/Web_cache
[3] http://en.wikipedia.org/wiki/Publish/subscribe

Redis: the speed of a cache with the persistence of a traditional DB

The popularity of Redis stems from its speed, its rich semantics and its stability. Redis is faster than other schema-less NoSQL databases such as MongoDB, yet designed to act as a system of record rather than a simple, volatile cache. Despite being an in-memory data store, Redis can persist its data to disk for durability.

Persistence is the ability for a process to recover its state following a shutdown. Data stored in-memory does not survive a server shutdown. The real insight of the Redis architecture is to realize that it works to offer predictable, low latencies and the same rich APIs as traditional disk-centric databases. Because memory access is faster than disk access (0.1 µs vs. 10 ms), Redis performs extremely well when compared to disk-centric databases.

On top of the high performance, Redis gives you two options to persist its data durably: (1) using snapshots and/or (2) using an append-only file.

Redis snapshots are point-in-time images of the in-memory data. Rather than writing every change to disk first, Redis creates a copy of itself in memory (using fork) so that the copy can be saved to disk as fast as possible. Data that was modified between now and the last successful snapshot is always at risk, thus running frequent snapshots is recommended if the disk is fast enough. The optimal frequency depends on your application; the first question to ask yourself when you set that parameter is how much the application would suffer if it lost that sliver of data-at-risk.

Redis also offers append-only file (AOF) as a more reliable persistence mode. With AOF, every change made to the dataset in memory is also written to disk. With careful disk configuration, this mechanism ensures that all past writes will have been saved to disk. However, the higher frequency of disk writes will have a significant negative impact on performance. Between AOF, snapshotting, and using Redis without persistence, there is a trade-off between 100% data integrity in the case of a crash versus speed of operation while the system runs.

Redis stores more than just strings

Redis differs from many basic key-value stores because it can store a variety of data types. Supported data types include:

- **Strings:** sequences of characters
- **Lists of strings:** ordered collections of strings sorted by insertion time
- **Sets:** non-repeating unordered collections of strings
- **Sorted sets:** non-repeating ordered collections of strings sorted by user defined score
- **Hashes:** associative arrays that maps string fields and string values

Redis hashes are particularly useful because small hashes (with fewer than 100 fields) are encoded in a very memory-efficient way. As a result, hash fields are an efficient way to represent the multiple data fields of objects in a single key. For example, if you're using Redis as a web cache to store information about visitors to your site it is more efficient to store all the attributes for a given user in a hash with a single key. The various attributes (username, email, account type) would be stored in separate fields of the same hash.

Redis scales with ease

Thanks to its key-value data model, a Redis database lends itself well to partitioning. You can split your keyed data across multiple Redis instances, each instance holding a subset. With partitioning you can use Redis to store more data than would fit in your most sizable server, memory-wise.

Lastly the Redis protocol is simple and consistent. Almost every programming language has a library for using Redis. This means that Redis can be used as a general purpose data store across your entire infrastructure, without the fear of locking yourself in a particular stack.

Redis Alternatives

There are alternative solutions to the problems tackled by Redis, but none fully overlap in the unique mix of capabilities that Redis offers; multithreaded caching system alternatives like Memcached do not persist to disk and typically support only string data types as values. More feature-rich database alternatives like MongoDB are typically more resource-intensive. Additional detail on these differences is provided below.

Memcached

Memcached is an open sourced, in-memory, multithreaded key-value store often used as a cache to speed up dynamic web applications. If no persistence is needed and you're only storing strings, Memcached is a simple and powerful tool for caching. If your data benefits from richer data structure, you will benefit from using Redis. In terms of performance, Memcached can perform better or worse than Redis depending on the specific use case[4][5][6].

MongoDB

MongoDB is an open-source, NoSQL database that supports richer data types. It is by some measure the most popular NoSQL database[7]. From the relational database

[4] http://oldblog.antirez.com/post/redis-memcached-benchmark.htm
[5] http://systoilet.wordpress.com/2010/08/09/redis-vs-memcached/
[6] http://dormando.livejournal.com/525147.html
[7] http://db-engines.com/en/ranking

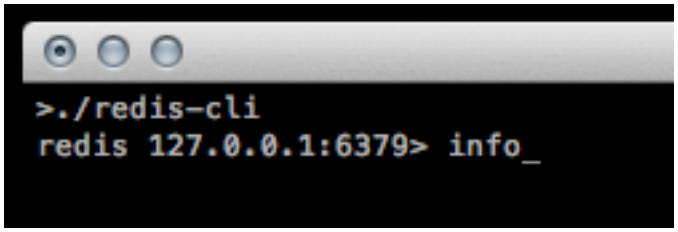
world it borrows the ability to run relatively complex queries without having to worry (too much) about what path the database will take to access the data.

Table 1: A comparison between popular data stores

	Redis	Memcached	MongoDB
In-memory	X	X	
Persistent	X		X
Key-value store	X	X	
Supports more than strings	X		X
Multithreaded		X	X
Supports larger-than-memory dataset			X
As fast as	Memory	Memory	Disk

How to access Redis metrics

To analyze the Redis metrics described below, you will need to access the actual data. Redis metrics are accessible through the Redis command line interface (`redis-cli`). Use the `info` command to print information and metrics about your Redis server.



The `info` command output is grouped into 10 sections:

- `server`
- `clients`
- `memory`
- `persistence`
- `stats`
- `replication`
- `cpu`
- `commandstats`

- `cluster`
- `keyspace`

This guide will cover important metrics from the memory and stats sections.

Note that `info` *does not* provide any metric of latency. We will discuss how to get to those particular metrics as well.

If you are only interested in seeing information related to a specific section, simply add the relevant section title as a parameter to the `info` command. For example, the command `info memory` would return only the statistics related to memory.

```
redis 127.0.0.1:6379> info memory
# Memory
used_memory:1065296
used_memory_human:1.02M
used_memory_rss:299008
used_memory_peak:1065232
used_memory_peak_human:1.02M
used_memory_lua:31744
mem_fragmentation_ratio:0.28
mem_allocator:libc
redis 127.0.0.1:6379>
```

The top 5 Redis performance metrics

To help you spot and remedy issues quickly we have selected the five metrics that report on the most common Redis performance issues we've encountered. These are detailed below.

1. Memory Usage: `used_memory`

The `used_memory` metric reports the total number of bytes allocated by Redis. The `used_memory_human` metric gives the same value in a more readable format.

```
# Memory
used_memory:1099952
used_memory_human:1.05M
used_memory_rss:299008
used_memory_peak:1099888
used_memory_peak_human:1.05M
used_memory_lua:31744
mem_fragmentation_ratio:0.28
mem_allocator:libc
```


These metrics reflect the amount of memory that Redis has requested to store your data and the supporting metadata it needs to run. Due to the way memory allocators interact with the operating system, the metrics do not account for waste due to memory fragmentation. In other words, the amount of memory reported in this metric will almost always be different than the total amount of memory that Redis has been allocated by the operating system.

Interpreting memory usage: Detect performance issues due to memory swapping

Memory usage is a critical component for Redis performance. If an instance exceeds available memory (`used_memory` > total available memory), the operating system will begin swapping and old/unused sections of memory (called pages) will be written to disk to make room for newer/active pages. Writing or reading from disk is up to 5 orders of magnitude slower than writing or reading from memory (0.1 µs for memory vs. 10 ms for disk). If swapping happens to the Redis process, its performance and the performance of any applications relying on Redis data will be severely impacted. By showing how much memory Redis is using, the `used_memory` metric will help determine if the instance is at risk to begin swapping or if it is swapping already.

Tracking memory usage to resolve performance issues

If you are using Redis without periodical snapshots, your instance is at risk of not only losing its data on shutdown but also being partially swapped in and out when memory usage exceeds 95% of total available memory.

Enabling snapshots requires Redis to create a copy of the current dataset in memory before writing it to disk. As a result, with snapshots enabled memory swapping becomes risky if you are using more than 45% of available memory; this can be exacerbated further by more frequent updates to your Redis instances. You can reduce Redis' memory footprint and thus the risk of swapping using the following tricks[8]:

1. **Use a 32-bit Redis instance if your data fits in less than 4GB:** Because pointers are half as big on a 32-bit Redis instance, its memory footprint is much smaller. However you become limited to 4GB per instance, even if you have more than 4GB in RAM. If your Redis instance is sharing the server with another piece of your infrastructure that requires or is more efficient with 64-bit processing, switching to 32-bit may not be practical. However, Redis dump files are compatible between 32-bit and 64-bit instances so you can try 32-bit and switch later if it makes sense for your system.
2. **Use hashes when possible:** Because Redis encodes small hashes (up to 100 fields) in a very memory-efficient manner you should use hashes when you

[8] http://redis.io/topics/memory-optimization

do not need set operations or the push/pop functionality of lists[9]. For example, if you have objects representing user information in a web application it is more efficient to use a single key with hash fields for each user attribute than to store attributes like name, email, password in separate keys[10]. In general, related data that are currently being stored in a string format with multiple keys should be converted to a single key with multiple hash fields. Examples of related data includes: attributes for a single object or various pieces of information for a single user. With the commands `HSET(key, fields, value)` and `HGET(key, field)` you can set and retrieve specific attributes from the hash.

3. **Set expirations for keys:** One simple way to decrease memory usage is to make sure that an expiration is set whenever storing transient data: If your information is only relevant for a known period of time, or if older keys are unlikely to be needed again, use the Redis expirations commands (`expire, expireat, pexpire, pexpireat`) to instruct Redis to automatically remove them once they are expired. Or, if you know how many new key-value pairs are created every second, you can tune the time-to-live (TTL) for your keys and limit Redis memory usage to a given amount.
4. **Evict keys:** In the Redis config file (usually called `redis.conf`) you can set a limit on the Redis memory usage by setting the "`maxmemory`" option and restarting the instance. You can also set `maxmemory` from the redis-cli by typing `config set maxmemory <value>`. If you are using snapshots, you should set `maxmemory` to 45% of available memory. This leaves half of your memory available for copying the dataset when snapshots are taken and 5% available for overhead. Without snapshotting, `maxmemory` can be set as high as 95% of available memory.

You must also choose how keys are selected for eviction for when the `maxmemory` limit is reached. The setting is called "`maxmemory-policy`" in the `redis.conf` file. If you are effectively expiring keys the "`volatile-ttl`" policy will likely be a good choice. If keys are not being expired quickly enough or at all, "`allkeys-lru`" will allow you to remove keys that have not been recently used regardless of their expiration state. The other modes are described below.

---

[9] http://instagram-engineering.tumblr.com/post/12202313862/storing-hundreds-of-millions-of-simple-key-value-pairs

[10] http://instagram-engineering.tumblr.com/post/12202313862/storing-hundreds-of-millions-of-simple-key-value-pairs


```
# MAXMEMORY POLICY: how Redis will select what to remove when maxmemory
# is reached. You can select among five behaviors:
#
# volatile-lru -> remove the key with an expire set using an LRU algorithm
# allkeys-lru -> remove any key accordingly to the LRU algorithm
# volatile-random -> remove a random key with an expire set
# allkeys-random -> remove a random key, any key
# volatile-ttl -> remove the key with the nearest expire time (minor TTL)
# noeviction -> don't expire at all, just return an error on write operations
#
```

By setting "`maxmemory`" to 45% or 95% of available memory (depending on your persistence policy) and by choosing "`volatile-ttl`" or "`allkeys-lru`" for eviction policy (depending on your expiration process), you can strictly limit Redis memory usage and in most case use Redis in a cache-like manner that ensures no memory swaps. The `noneviction` mode only makes sense if you cannot afford to lose keys to memory limits.

2. Number of commands processed: `total_commands_processed`

The `total_commands_processed` metric gives the number of commands processed by the Redis server. Commands come from the one or more clients connected to the Redis server. Each time the Redis server completes any of the over 140 possible commands from the client(s), `total_commands_processed` is incremented. For example, if the Redis servers complete two commands from client_`x` and three commands from client_`y`, the total commands metric would increase by five.

```
# Stats
total_connections_received:4
total_commands_processed:21949
instantaneous_ops_per_sec:0
rejected_connections:0
expired_keys:0
evicted_keys:0
keyspace_hits:0
keyspace_misses:0
pubsub_channels:0
pubsub_patterns:0
latest_fork_usec:0
redis 127.0.0.1:6379> _
```

Interpreting number of commands processed: Diagnose increased latency

Tracking the number of commands processed is critical for diagnosing latency issues in your Redis instance. Because Redis is single-threaded, command requests are processed sequentially. The typical latency for a 1Gb/s network is about 200 µs. If you are seeing slow response time for commands and latency that is significantly higher than 200 µs, it could be because there are a high number of requests in the command queue.

In that case, you would see slow responses and a spike in the number of total commands processed. If instead, slowness and increased latency are caused by one or more slow commands, you would see your total commands metric drop or stall completely as Redis performance degrades. Diagnosing these problems requires that you track command volume and latency over time.

For example, you could setup a script that periodically logs the `total_commands_processed` metric and also measures latency. You can then use this log of historical command volume to determine if your total number of commands processed increased or decreased when you observed slower response times.

Using number of commands processed to resolve increases in latency

If you are seeing an increase or decrease in total commands processed as compared to historical norms, it may be a sign of high command volume (more commands in queue) or several slow commands blocking the system. Here are three ways to address latency issues caused by high command volume and slow commands:

1. **Use multi-argument commands:** If Redis clients send a large number of commands to the Redis server in a short period of time, you may see slow response times simply because later commands are waiting in queue for the large volume of earlier commands to complete. One way to improve latency is to reduce the overall command volume by using Redis commands that accept multiple arguments. For example, instead of adding 1,000 elements to a list using a loop and 1,000 iterations of the Redis command `LSET`, you could create a separate list completely on the client side with each of the 1,000 elements and use a single Redis command, `LPUSH` or `RPUSH`, to add all 1,000 elements at once. The table below highlights several Redis commands that can be used only for single elements and corresponding multiple element commands that can help you minimize overall command volume.

Table 2: Redis single-argument commands and their corresponding multi-argument alternatives

Single-Argument Command	Single-Argument Description	Multi-Argument Alternative	Multi-Argument Description
`SET`	Set the value of a key	`MSET`	Set multiple keys to multiple values
`GET`	Get the value of a key	`MGET`	Get the values of all the given keys
`LSET`	Set value of an element in a list	`LPUSH, RPUSH`	Prepend/append multiple values to a list

LINDEX	Get an element from a list	LRANGE	Get a range of elements from a list
HSET	Set the string value of a hash	HMSET	Set multiple hash fields to multiple values
HGET	Get the value of a hash field	HMGET	Get the values of all the given hash fields

2. **Pipeline commands:** Another way to reduce latency associated with high command volume is to pipeline several commands together so that you reduce latency due to network usage. Rather than sending 10 client commands to the Redis server individually and taking the network latency hit 10 times, pipelining the commands will send them all at once and pay the network latency cost only once. Pipelining commands is supported by the Redis server and by most clients. This is only beneficial if network latency is significantly larger than your instance's.
3. **Avoid slow commands for large sets:** If increases in latency correspond with a drop in command volume you may be inefficiently using Redis commands with high time-complexity. High time-complexity means the required time to complete these commands grows rapidly as the size of the dataset processed increases. Minimizing use of these commands on large sets can significantly improve Redis performance. The table below lists the Redis commands with highest time-complexity. Specific command attributes that affect Redis performance and guidelines for optimal performance are highlighted for each of the commands.

Table 3: Redis commands with high time complexity

Command	Description	Improve Performance By
ZINTERSTORE	intersect multiple sorted sets and store result	reducing the number of sets and/or the number of elements in resulting set
SINTERSTORE	intersect multiple sets and store result	reducing size of smallest set and/or the number of sets
SINTER	intersect multiple sets	reducing the size of smallest set and/or the number of sets
MIGRATE	transfer key from one Redis instance to another	reducing the number of objects stored as values and/or their average size
DUMP	return serialized value for a given key	reducing the number of objects stored as values and/or their average size

ZREM	remove one or more members from sorted set	reducing the number of elements to remove and/or the size of the sorted set
ZUNIONSTORE	add multiple sorted sets and store result	reducing the total size of the sorted sets and/or the number of elements in the resulting set
SORT	sort elements in list, set, or sorted set	reducing the number of element to sort and/or the number of returned elements
SDIFFSTORE	subtract multiple sets and store result	reducing the number of elements in all sets
SDIFF	subtract multiple sets	reducing the number of elements in all sets
SUNION	add multiple sets	reducing the number elements in all sets
LSET	set value of an element in a list	reducing the length of the list
LREM	remove elements from a list	reducing the length of the list
LRANGE	get range of elements from a list	reduce the start offset and/or the number of elements in range

3. Latency

Latency measures the average time in milliseconds it takes the Redis server to respond. This metric is not available through the Redis info command. To see latency, go to your `redis-cli`, change directory to the location of your Redis installation, and type the following:

```
./redis-cli --latency -h 'host' -p 'port'
```

where "host" and "port" are relevant numbers for your system. While times depend on your actual setup, typical latency for a 1GBits/s network is about 200 µs.

```
>./redis-cli --latency -h 127.0.0.1 -p 6379
min: 0, max: 15, avg: 0.27 (2480 samples)_
```

Interpreting latency: Tracking Redis performance

Performance, and more specifically, its predictable low latency is one of the main reasons Redis is so popular. Tracking latency is the most direct way to see changes in Redis performance. For a 1Gb/s network, a latency greater than 200 µs likely points to a problem. Although there are some slow I/O operations that run in the

background, a single process serves all client requests and they are completed in a well-defined sequential order. As a result, if a client request is slow all other requests must wait to be served.

Using the latency command to improve performance

Once you have determined that latency is an issue, there are several measures you can take to diagnose and address performance problems:

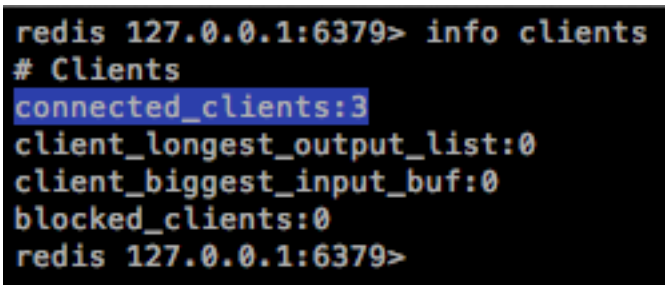
1. **Identify slow commands using the slow log:** The Redis slow log allows you to quickly identify commands that exceed a user specified execution time. By default, commands that take longer than 10 ms are logged. For the purposes of the slow log, execution time does not include I/O operations so this log would not log slow commands solely due to network latency. Given that typical commands, including network latency, are expected to complete in approximately 200 µs, commands that take 10 ms for just Redis execution are more than 50 times slower than normal.

   To see all commands with slow Redis execution times, go to your `redis-cli` and type `slowlog get`. The third field of the result gives the execution time in microseconds. To see only the last 10 slow commands, type `slowlog get 10`. See page 11 for details on how to address latency issues related to slow commands.

   ```
   redis 127.0.0.1:6379> slowlog get
   1) 1) (integer) 0
      2) (integer) 1374868535
      3) (integer) 320
      4) 1) "config"
         2) "set"
         3) "slowlog-log-slower-than"
         4) "100"
   ```

   For a more detailed look at slow commands, you can adjust the threshold for logging. In cases where few or no commands take longer than 10 ms, lower the threshold to 5 ms by entering the following command in you `redis cli`: `config set slowlog-log-slower-than 5000`.

2. **Monitor client connections:** Because Redis is single-threaded, one process serves requests from all clients. As the number of clients grows, the percentage of resource time given to each client decreases and each client spends an increasing amount of time waiting for their share of Redis server time. Monitoring the number of clients is important because there may be applications creating client connections that you did not expect or your application may not be efficiently closing unused connections. To see all clients connected to your Redis server go to your `redis-cli` and type `info clients`. The first field (`connected_clients`) gives the number of client connections.



The default maximum number of client connections is 10,000. Even with low client command traffic, if you are seeing connection counts that are above 5,000, the number of clients may be significantly affecting Redis performance. If some or more of your clients are sending large numbers of commands the threshold for affecting performance could be much lower.

3. **Limit client connections:** In addition to monitoring client connections, Redis versions 2.6 and greater allow you to control the maximum number of client connections for your Redis server with the `redis.conf` directive "`maxclients`". You can also set the `maxclients` limit from the `redis-cli` by typing `config set maxclients <value>`. You should set `maxclients` to between 110% and 150% of your expected peak number of connections, depending on variation in your connections load. Connections that exceed your defined limit will be rejected and closed immediately. Setting a maximum is important for limiting the number of unintended client connections. In addition, because an error message is returned for failed connection attempts, the `maxclient` limit helps warn you that a significant number of unexpected connections are occurring. Both are important for controlling the total number of connections and ultimately maintaining optimal Redis performance.
4. **Improve memory management:** Poor memory management can cause increased latency in Redis. If your Redis instance is using more memory than is available, the operating system will swap parts of the Redis process out of physical memory and onto disk. Swapping will significantly increase latency. See page 8 for more information on how to monitor and reduce memory usage.
5. **Metric correlation:** Diagnosing and correcting performance issues often requires you to correlate changes in latency with changes in other metrics. A spike in latency that occurs as the number of commands processed drops likely indicates slow commands blocking the system. But if latency increases as memory usage increases you are probably seeing performance issues due to swapping. For this type of correlated metric analysis, you need historical perspective in order for significant changes in metrics to be perceptible as well as the ability to see all relevant metrics across your stack in one place. To do this with Redis, you would create a script that calls the info command periodically, parses the output, and records key metrics in a log file. The log can be used to identify when latency changes occurred and what other metrics changed in tandem.

4. Fragmentation Ratio

The `mem_fragmentation_ratio` metric gives the ratio of memory used as seen by the operation system (`used_memory_rss`) to memory allocated by Redis (`used_memory`).

$$Memory\ Fragmentation\ Ratio = \frac{Used\ Memory\ RSS}{Used\ Memory}$$

The `used_memory` and `used_memory_rss` metrics both include memory allocated for:

- **User-defined data:** memory used to store key-value data
- **Internal overhead:** internal Redis information used to represent different data types

RSS stands for *Resident Set Size* and is the amount of physical memory the operating system has allocated to your Redis instance. In addition to user-defined data and internal overhead, the `used_memory_rss` metric includes memory consumed by memory fragmentation. Memory fragmentation is caused by inefficiencies in physical memory allocation/deallocation by the operating system.

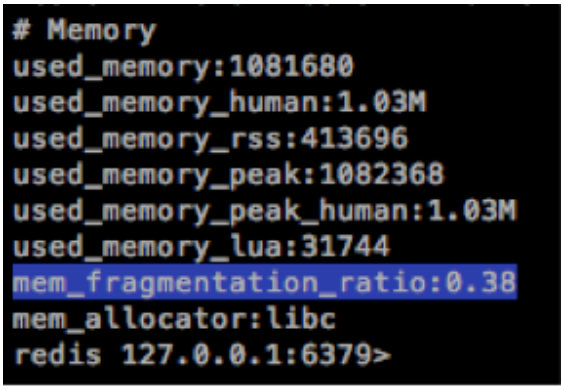
The operating system is in charge of allocating physical memory to the various processes it is hosting. The actual mapping between Redis' memory and physical memory is handled by the operating system's virtual memory manager through the mediation of a memory allocator.

For example while Redis would benefit from contiguous physical memory pages for a 1GB database, no such contiguous segment may be available so the operating system will have to map that 1GB into multiple smaller memory segments.

The memory allocator adds another layer of complexity in that it will often pre-allocate memory blocks of various sizes to be able to service the application as fast as possible.

Interpreting fragmentation ratio: Understand resource performance

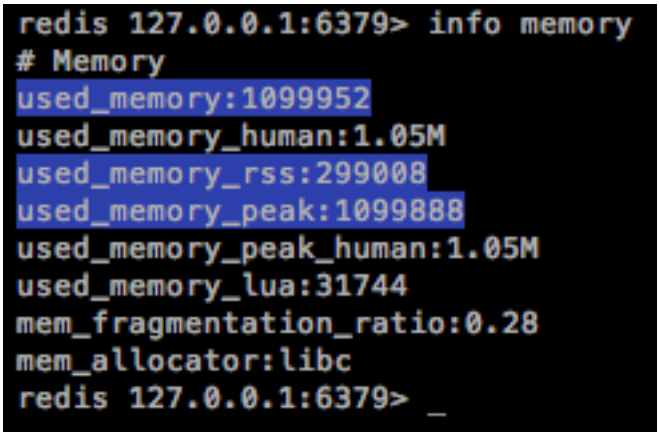
Tracking fragmentation ratio is important for understanding the resource performance of your Redis instance. The fragmentation ratio should be slightly greater than 1 which indicates low memory fragmentation and no memory swapping. A fragmentation ratio above 1.5 indicates significant memory fragmentation since Redis consumes 150% of the physical memory it requested. A fragmentation ratio below 1 tells you that Redis memory allocation exceeds available physical memory and the operating system is swapping. Swapping will cause significant increases in latency (See page 8, "Tracking memory usage to resolve performance issues").



Using the fragmentation ratio to predict performance issues

If the fragmentation ratio is outside the range of 1 to 1.5, it is likely a sign of poor memory management by either the operating system or by your Redis instance. Here are three ways to correct the problem and improve Redis performance:

1. **Restart your Redis server**: If your fragmentation ratio is above 1.5, restarting your Redis server will allow the operating system to recover memory that is effectively unusable because of external memory fragmentation. External fragmentation occurs when Redis frees blocks of memory but the allocator (the piece of code responsible for managing memory distribution), does not return that memory to the operating system. You can check for external fragmentation by comparing the values of the `used_memory_peak`, `used_memory_rss` and `used_memory` metrics. As the name suggests, `used_memory_peak` measures the largest historical amount of memory used by Redis regardless of current memory allocation. If `used_memory_peak` and `used_memory_rss` are roughly equal and both significantly higher than `used_memory`, this indicates that external fragmentation is occurring. All three of these memory metrics can be displayed by typing info memory in your `redis-cli`.



To restart the server, use the `redis-cli` and type `shutdown save` to save your dataset and shut down the Redis server. Once you restart the Redis server your data set should be available after it has been loaded from disk.

2. **Limit memory swapping**: If the fragmentation ratio is below 1, your Redis instance is probably partially swapped out to disk. Swapping will ruin Redis' performance so you should take steps to increase available memory or reduce memory usage. See page 8 for more information on effective memory management with Redis.
3. **Change your memory allocator:** Redis supports several different memory allocators (glibc's malloc, jemalloc[11], tcmalloc) each with different behavior in terms of internal and external fragmentation. Not for the faint of heart, this will require you to (1) understand clearly the differences between allocators and (2) re-compile Redis. Just know that it is an option once you have clearly established that the memory allocator is working against Redis.

5. Evictions

The `evicted_keys` metric gives the number of keys removed by Redis due to hitting the `maxmemory` limit. `maxmemory` is explained in more detail on page 9. Key evictions only occur if the `maxmemory` limit is set.

When evicting a key because of memory pressure, Redis does not consistently remove the oldest data first. Instead, a random sample of keys are chosen and either the least recently used key ("LRU[12] eviction policy") or the key closest to expiration ("TTL[13] eviction policy") within that random set is chosen for removal.

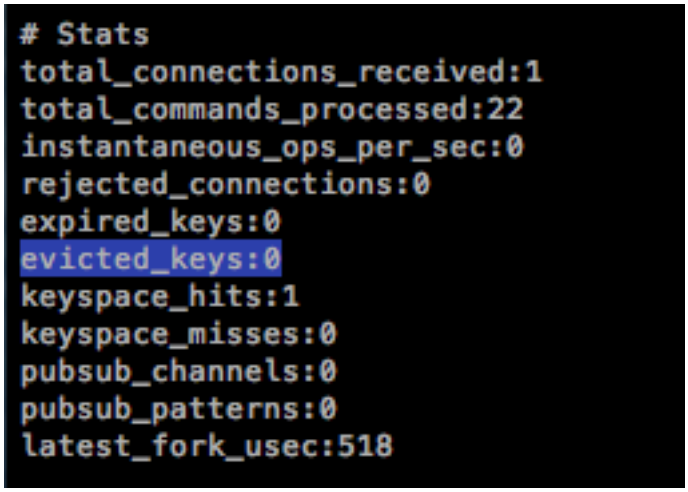
You have the option to select between the lru and ttl eviction policies in the config file by setting `maxmemory-policy` to "`volatile-lru`" or "`volatile-ttl`" respectively. The TTL eviction policy is appropriate if you are effectively expiring keys. If you are not using key expirations or keys are not expiring quickly enough it makes sense to use the lru policy which will allow you to remove keys regardless of their expiration state.

---

[11] For more details on jemalloc https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919

[12] Least recently used

[13] Time to live



Interpreting evictions: Detecting performance issues due to key deletion

Tracking eviction is important because Redis must commit limited processing resources in order to evict a key. If your `evicted_keys` metric is consistently above zero, you are likely to see increased latency for commands in queue because Redis must process frequent evictions in addition to processing incoming client commands.

Note that evictions are not as detrimental to performance as memory swaps. If the choice is between forcing memory swaps and allowing evictions, you should let Redis evict keys by choosing an appropriate `maxmemory_policy` (see page 9).

Using eviction reductions to improve performance

Reducing evictions can be a straightforward way to improve Redis performance. Here are two ways to decrease the number of evictions Redis must perform:

1. **Increase `maxmemory` limit:** If you are using snapshots, `maxmemory` can be set as high as 45% of available physical memory with little risk of causing memory swaps. Without snapshots (but AOF), it makes sense to set `maxmemory` to 95% of available memory. See page 8 for more information on snapshots and `maxmemory` limits. If you are below the 45% or 95% thresholds increasing `maxmemory` limit will allow Redis to hold more keys in memory and can dramatically decrease evictions. However, if `maxmemory` is already set at or above the recommended thresholds, increasing `maxmemory` may not improve Redis performance. Instead, it may cause memory swaps to occur which will increase latency and degrade performance. You can set the value of `maxmemory` from the `redis-cli` by typing `config set maxmemory <value>`.
2. **Partition your instance:** Partitioning is a method of splitting your data across multiple instances. Each of the resulting instances will contain a

subset of the entire dataset. By partitioning, you can combine the resources of many computers, increase available physical memory, and allow Redis to store more keys without evictions or memory swaps. If you have large datasets and `maxmemory` is already set at or above the recommended thresholds, this can be an effective way to reduce evictions and improve Redis performance. Partitioning can be done in a variety of ways. Here are a few options for implementing Redis partitioning.

a. **hash partitioning:** a simple method accomplished by using a hash function to map key names into a range of numbers which correspond to specific Redis instances.
b. **client side partitioning**: a commonly used method in which Redis clients select which instance to read and write for a given key.
c. **proxy partitioning**: Redis client requests are sent to a proxy which determines which Redis instance to use based on a configured partitioning schema.

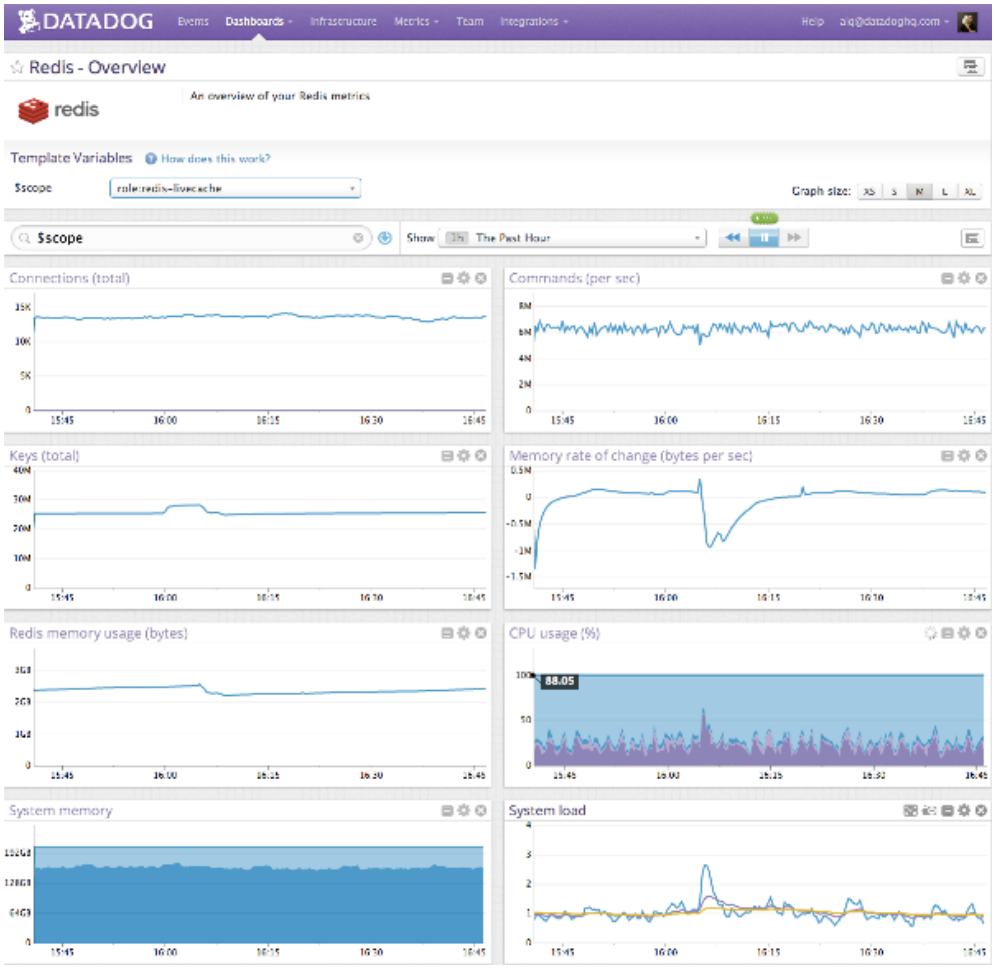
Conclusion

For developers, Redis provides fast in-memory, key-value store with an easy to use programming interface. However, in order to get the most out of Redis, you should understand what factors can negatively impact performance and what metrics can help you avoid pitfalls. After reading this guide, you should understand some of the key Redis metrics, how to view them, and most importantly how to use them for detecting and solving Redis performance issues.

How Datadog Can Help

Datadog is a monitoring service that natively understands Redis and its metrics.

Within a few minutes, Datadog lets you collect graph and alert on any Redis metric (including the five we have presented in this article). By freeing you from the time-consuming collection process, Datadog lets you focus on understanding the performance of your Redis instances and how they affect the rest of your applications.



An example of Redis-centric dashboard in Datadog.

Datadog is compatible with all major linux distributions.

About the Authors

Conor Branagan, Software Engineer, Datadog

Conor is a Software Engineer at Datadog. Equally at ease with frontend and backend code, he's made Redis his data store of choice. He started at Datadog through the HackNY intern program in New York. In his free time he enjoys hiking and enjoying the great food of Astoria, Queens.

Patrick Crosby, Product Marketer, Datadog

Patrick is a Product Marketer at Datadog. Prior to joining Datadog, he was a Sales Engineer at Innography, a venture backed software company in Austin, TX. Patrick managed relationships for Innography's 60 largest global clients, led strategy and

training for the company's lead generation team, and developed product requirements for entry into the $800 million pharmaceutical IP-intelligence market. Patrick is a Fulbright Scholar, holds a BS in Electrical Engineering and Computer Science from Duke University, and will graduate with an MBA from Columbia Business School in 2014.

About Datadog

Datadog unifies the data from servers, databases, applications, tools and services to present a unified view of on-premise and cloud infrastructure. These capabilities are provided on a SaaS-based monitoring and data analytics platform that enables Dev and Ops teams working collaboratively on the infrastructure to avoid downtime, resolve performance problems and ensure that development and deployment cycles finish on time.

To find out more, visit www.datadog.com.

