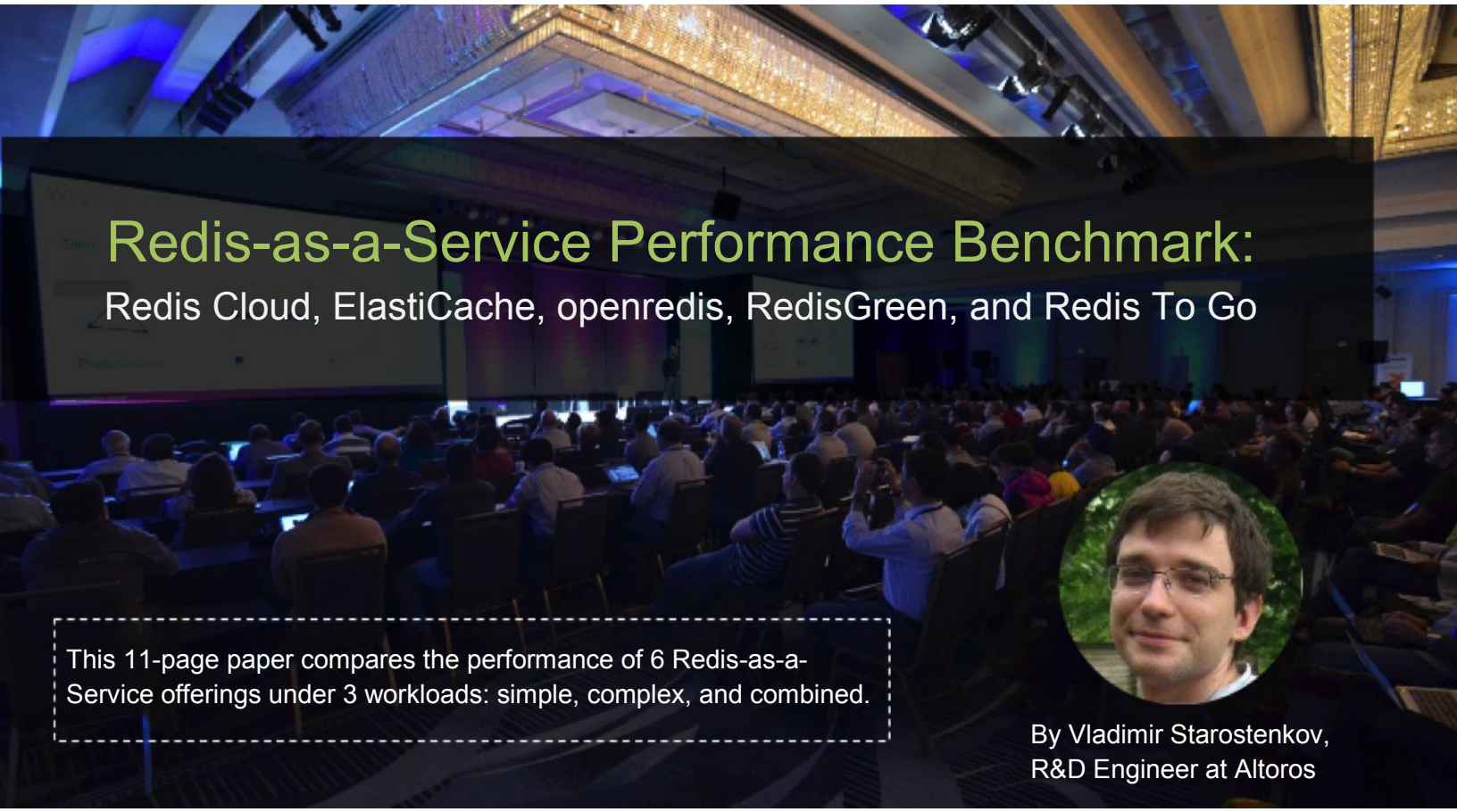
ALTOROS



Redis-as-a-Service Performance Benchmark:

Redis Cloud, ElastiCache, openredis, RedisGreen, and Redis To Go

This 11-page paper compares the performance of 6 Redis-as-a-Service offerings under 3 workloads: simple, complex, and combined.

By Vladimir Starostenkov,
R&D Engineer at Altoros

redis

redis

January 2015

Table of Contents

1. Executive Summary

Redis is a very popular open source, advanced key-value cache and store. It can be operated independently or obtained from one of the providers that offer Redis as a service. However, it is not always easy to judge the performance of as-a-Service products: vendors' claims are often biased, the available benchmarks are extremely simple, and Redis is usually treated as a caching-only solution. This makes it rather difficult to settle on one public Redis provider, even when you know what kind of workload you need to handle.

Unlike most performance tests that focus on single get/set operations, this benchmark demonstrates how Redis performs under three workload scenarios. It compares the performance of six major Redis-as-a-Service offerings that are available on the Amazon Web Services cloud:

- Redis Cloud Standard (by Redis)
- Redis Cloud Cluster (by Redis)
- ElastiCache (by Amazon)
- openredis (by Amakawa)
- RedisGreen (by Stovepipe Studios)
- Redis To Go (by Exceptional Cloud Services / Rackspace)

Note: While most Redis providers offer a single master or simple master-slave/multiple-slave Redis configurations, Redis also provides the Redis Cloud Cluster solution, which is claimed to be faster than Redis Cloud Standard. Therefore, we decided to test both.

This paper describes the configuration, tools, workloads, and results of the benchmark. It includes three comparative tables and eight performance diagrams that show how each provider performs in close to real-life use cases.

2. Configuration, Tools, and Workloads

For this study, each vendor provided us with their best performing Redis-as-a-Service plan, using no more than a single AWS EC2 instance. All of the server instances were located in the same region as the benchmarking client. In addition, ElastiCache and Redis Cloud have a mechanism that allows users to select on which AWS Availability Zone a Redis instance will be created.

Two different tools were used to generate workloads for this research:

1) memtier_benchmark, an open source traffic generator written in C++. With a variety of options, memtier_benchmark allows for combining simple get/set operations into a very specific workload. It supports both Redis and Memcached protocols.

2) A Java-based stress tool that simulates a more complex workload. It imitates a business case that uses Redis as a database performing some data aggregation via built-in operations on sets/sorted sets, such as union and intersect.

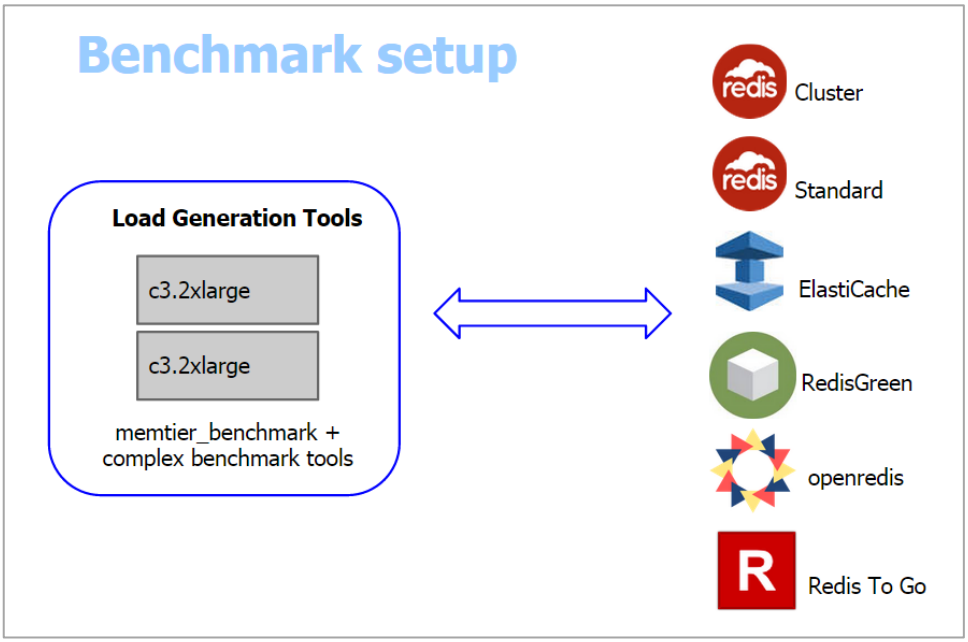


Figure 1. Benchmark configuration and tools

Though we commonly use Yahoo! Cloud Serving Benchmark (YCSB) for evaluation of NoSQL databases, for this comparison we decided to try more Redis-specific tools. Despite the fact that both of these tools are written by Redis, none of them uses vendor-specific features or configurations. Both are open source and anyone can reproduce the results or play with the code/settings.

We used a complex scenario that involved aggregation of data from different structures. This scenario was designed to simulate a business case where several objects are merged to retrieve and analyze specific data. In particular, the scenario aggregates data from different Redis's sorted sets into a single one. This type of aggregation is often employed by financial applications for aggregating transactions from multiple accounts that belong to the same business, as well as by games where the need is to merge the results from different tournaments and users into a single coherent view.

The benchmark consisted of three workloads:

a) **Simple.** This workload was made up of the SET and GET operations in a proportion of 1:1 and used different pipeline sizes. The workload was generated by *memtier_benchmark* (please see the Appendix for the actual arguments that were used). The Simple workload was mostly network bound. It was generated from two independent client nodes running in parallel.

b) **Complex.** The second workload was generated by the Java tool mentioned above. It mainly consisted of the ZUNIONSTORE operations on four sorted sets, having 20,000 elements each. The Complex workload was mostly CPU bound.

c) **Combined.** The third workload was a combination of Simple and Complex loads, each generated from a separate client node. This type of scenario probably best emulates real Redis usage in load-intensive production environments, since most systems based on Redis perform both simple (SET/GET) and complex operations.

3. Benchmark Results

The diagrams and tables below demonstrate how the six Redis solutions performed under the three workloads.

3.1 The Simple workload

Since most applications use pipelining to improve performance, we tested the Simple workload with pipeline sizes of 4 and 50. Pipeline 4 seems to be more common today, while Pipeline 50 is used to show how to maximize performance of Redis with a reasonable latency value. Each latency value in the table below is a pipeline latency divided by the pipeline length.

Table 1. The Simple workload, latency vs. pipeline length

Pipeline	Redis Cloud Standard		Redis Cloud Cluster		ElastiCache		openredis		RedisGreen		Redis To Go	
	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)
4	398,357	0.50	300,526	0.66	262,836	0.76	226,498	0.88	261,058	0.77	172,389	1.16
50	656,137	0.31	1,138,635	0.18	611,068	0.33	301,355	0.66	536,391	0.36	331,931	0.60

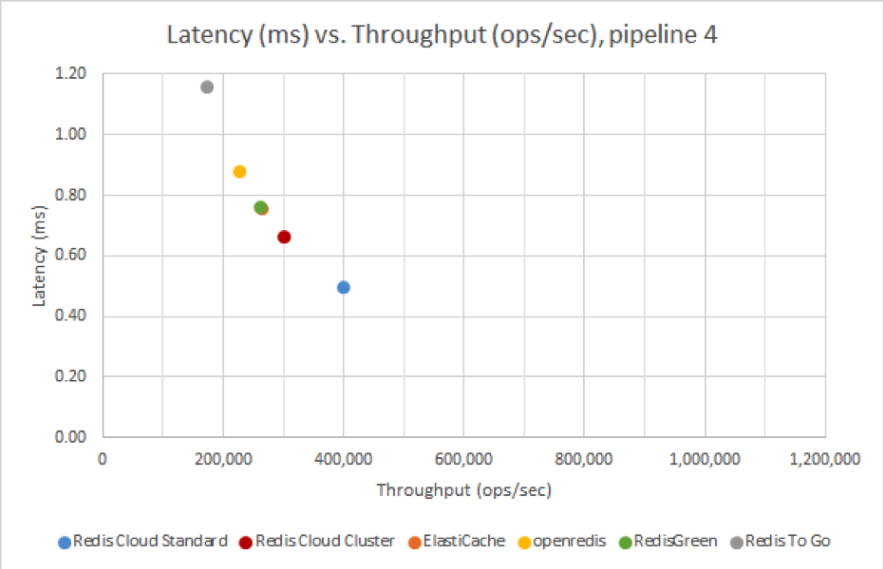


Figure 2. Latency (ms) vs. throughput (ops/sec), pipeline 4

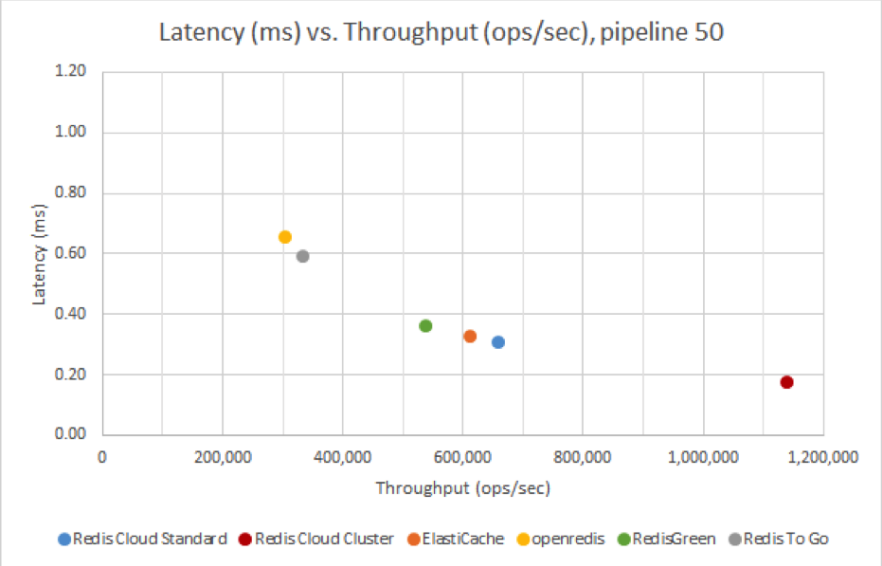


Figure 3. Latency (ms) vs. throughput (ops/sec), pipeline 50

3.2 The Complex workload

Under this workload, we varied the number of client threads—in order to assess scalability of the solutions. (Increasing the number of client threads increases the number of the complex workload requests that Redis needs to process.)

Table 2. The Complex workload: latency and throughput vs. number of threads

Threads	Redis Cloud Standard		Redis Cloud Cluster		ElastiCache		openredis		RedisGreen		Redis To Go	
	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)
1	12	85	13	79	14	71	8	130	13	75	5	214
3	11	266	33	92	13	225	8	386	13	239	5	603
5	11	442	46	109	13	380	8	655	13	401	5	1,022

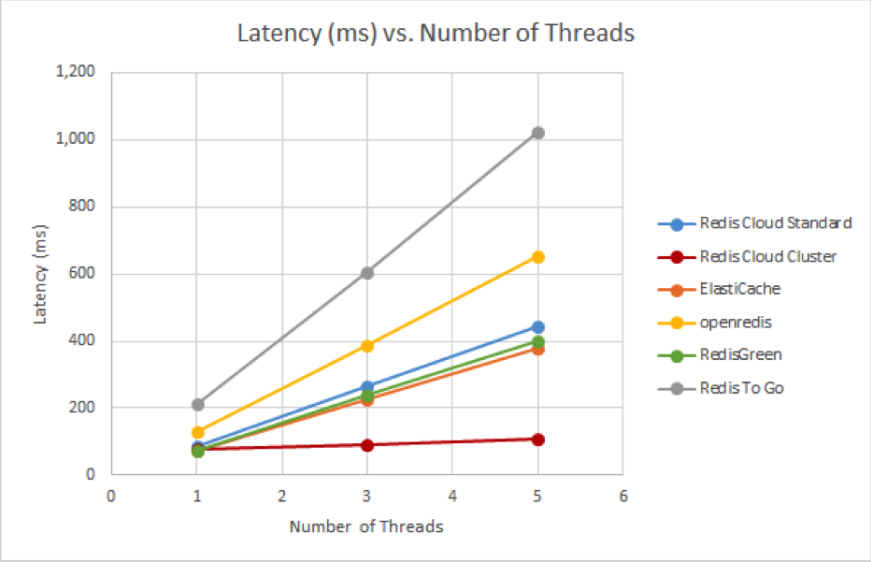


Figure 4. Latency (ms) vs. number of threads

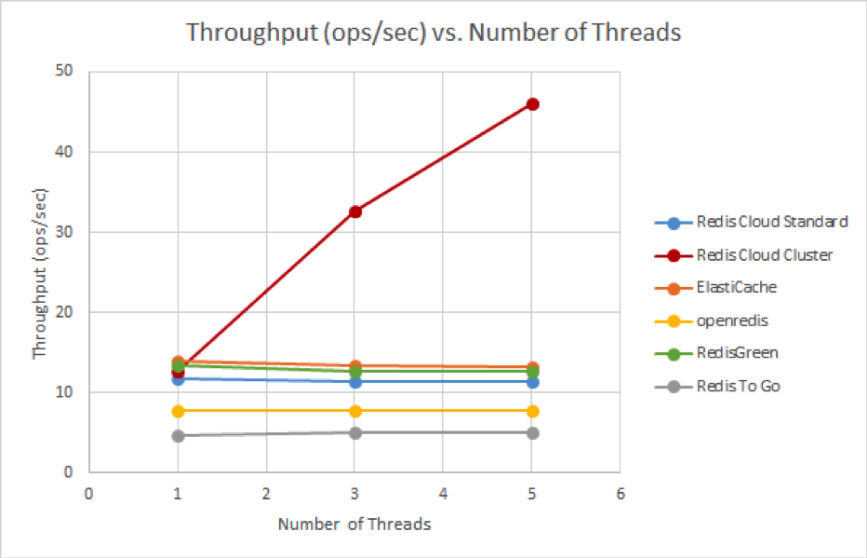


Figure 5. Throughput (ops/sec) vs. number of threads

3.3 The Combined workload

As mentioned earlier, the combination of both simple and complex workloads—two different types of queries running concurrently—is more representative, since it imitates a real-life Redis use case. We expect the growing number of threads running computationally intensive complex queries to suppress the performance of simple set and get methods.

Table 3. The Combined workload (Simple + Complex)

Threads	Redis Cloud Standard				Redis Cloud Cluster				ElastiCache			
	Complex		Simple		Complex		Simple		Complex		Simple	
	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)
1	12	86	5,527	54	13	80	15,169	18	11	87	4,211	67
3	11	270	2,457	41	33	91	7,928	13	13	237	2,121	47
5	11	472	1,851	18	46	108	5,452	7	13	393	1,503	24
Threads	**openredis**				**RedisGreen**				**Redis To Go**			
	Complex		**Simple**		**Complex**		**Simple**		**Complex**		**Simple**	
	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)	Throughput (ops/sec)	Latency (ms)
1	7	139	2,876	85	11	91	4,004	52	5	188	2,485	74
3	7	412	1,374	73	12	247	2,393	42	5	572	1,324	76
5	7	680	1,176	35	12	408	1,932	25	5	936	1,349	40

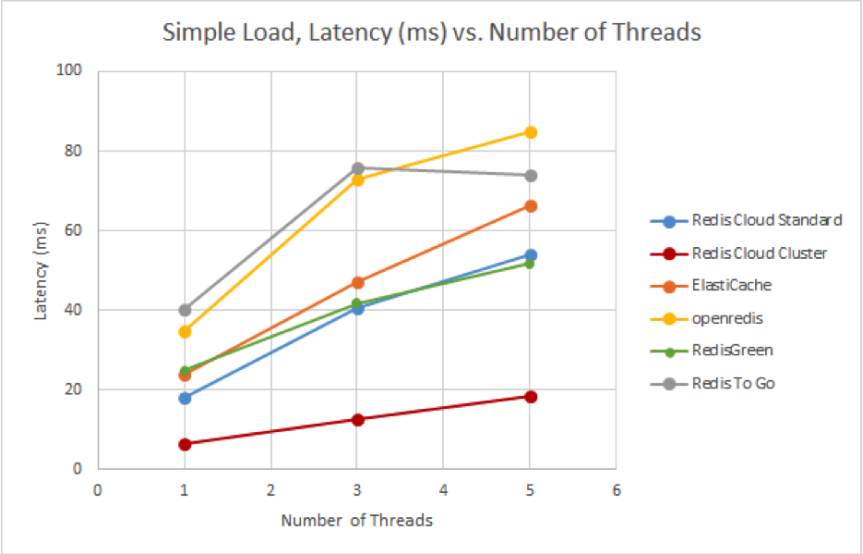


Figure 6. The Simple workload, latency (ms) vs. number of threads

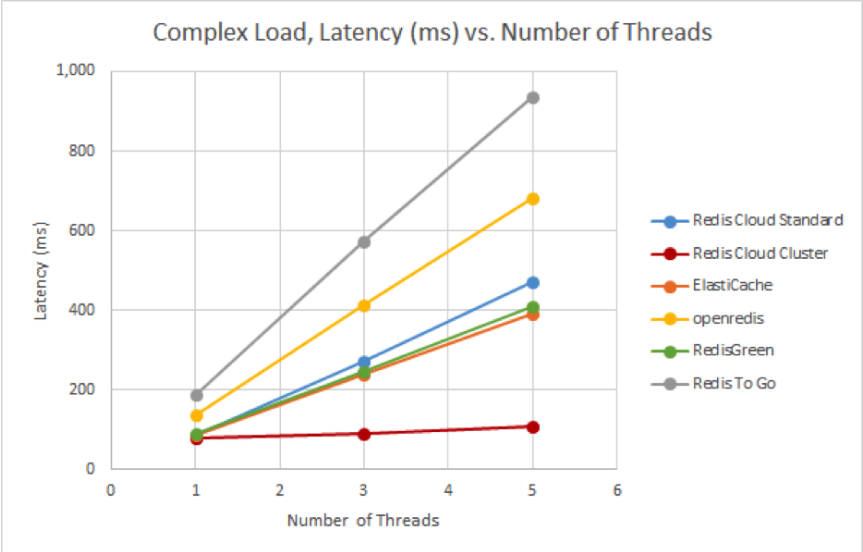


Figure 7. The Complex workload, latency (ms) vs. number of threads

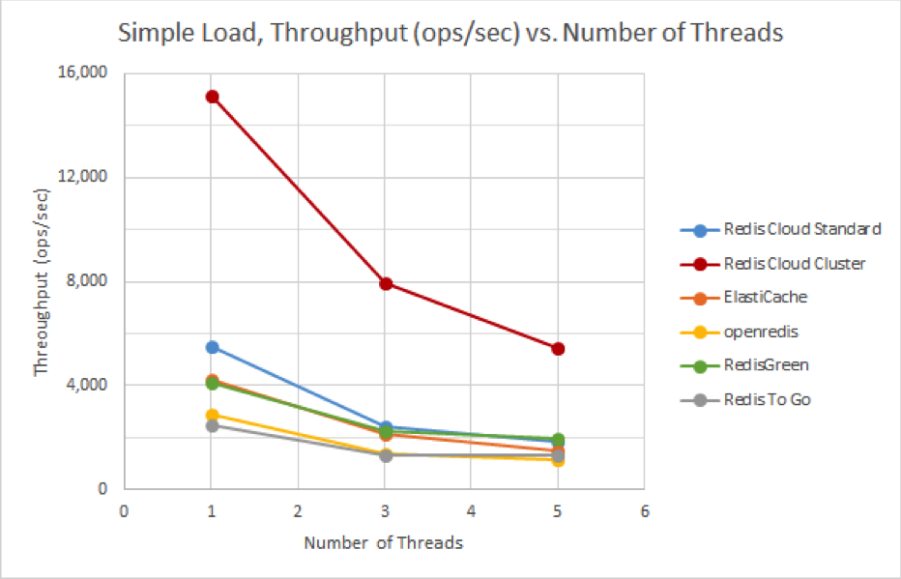


Figure 8. The Simple workload, throughput (ops/sec) vs. number of threads

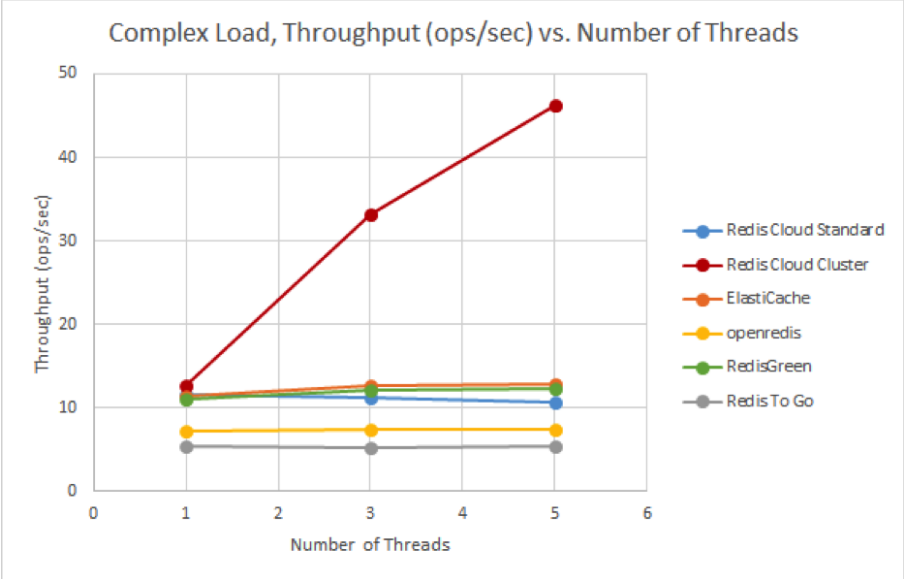


Figure 9. The Complex workload, throughput (ops/sec) vs. number of threads

4. Summary

The results of the benchmarks clearly demonstrate that for complex workloads **Redis Cloud Cluster** outperformed all the other solutions, both in throughput and latency. Its performance grew almost linearly as the number of client threads was increasing. The test results of the Combined workload also justify the solution's "Cluster" postfix. By employing sharding, **Redis Cloud Cluster** was capable of scaling in a near-linear fashion under the mixed workload.

For simpler workloads, one should start RaaS evaluation with **ElastiCache**, **RedisGreen**, or **Redis Cloud Standard**. Redis Cloud Standard provided the best results with a pipeline size of four commands. However, with the larger pipeline of 50 commands, Redis Cloud Cluster performed two times faster than Redis Cloud Standard, RedisGreen, and ElastiCache.

Table 4. Overall overview of the Redis-as-a-Service offerings

Performance	Redis Cloud Standard	Redis Cloud Cluster	ElastiCache	openredis	RedisGreen	Redis To Go
Simple workloads	good	good	average	poor	average	poor
Complex workloads	average	good	average	poor	average	poor
Combined workloads	poor	good	poor	poor	poor	poor

During our research, we were not able to identify a workload that would demonstrate the best use cases for **openredis** and **Redis To Go**.

To get more details about this benchmark, feel free to contact us at engineering@altoros.com. Or, download other NoSQL comparisons at www.altoros.com/research-papers.

5. Appendix: Testing Configuration

The overall size of the benchmarked database was approximately 4 GB, which contained 5,000,000 simple keys as well as sorted sets—managing 24,000,000 entries in total.

The client node was located in the same region as the tested solutions: *us-east-1*.

Table 5. Vendor plans and types of instances featured in the benchmark

RaaS Provider	Instance type
Redis Cloud Standard	r3.2xlarge
Redis Cloud Cluster (5 shards)	Pay-as-You-Go
ElastiCache	r3.2xlarge
openredis	A dedicated master/slave server (4 GB)
RedisGreen	15 GB X-Large (us-east-1)
Redis To Go	A 10 GB instance

5.1 memtier_benchmark

The following is an example of memtier_benchmark and command line arguments:

```
$memtier_benchmark --server=<hostname> --port=<port> --ratio=1:1 --test-time=120 -
-data-size=100 --threads=2 --clients=50 --pipeline=4 --key-pattern=S:S
```

5.2 A Java-based tool for the complex load

Command line arguments:

```
$java -jar saasbenchmark-1.0-SNAPSHOT-jar-with-dependencies.jar run <hostname>
<port> 120 <number of threads> 300
```

6. About the Authors

Vladimir Starostenkov is a Senior R&D Engineer at Altoros. He is focused on implementing complex software architectures, including data-intensive systems and Hadoop-driven apps. Having background in computer science, Vladimir is passionate about artificial intelligence and machine learning algorithms. His NoSQL and Hadoop studies were published in NetworkWorld, CIO.com, and other industry media.

Altoros brings Cloud Foundry-based “software factories” and NoSQL-driven “data lakes” into organizations through training, deployment, and integration. With 250+ employees across 9 countries in Europe and Americas, Altoros is the company behind some of the world’s largest Cloud Foundry and NoSQL deployments. In 2014, Altoros was recognized as one of top big data, BI, and Hadoop consultants (by Clutch, a Washington, DC-based research agency). For more, please visit www.altoros.com or follow @altoros.