Redis in the Yahoo! Cloud Serving Benchmark

A Swiss Army knife with a lot of screws and bolts

Robert Lehmann

robert.lehmann@student.hpi.uni-potsdam.de

Benchmark documentation

September 14, 2011

supervised by Johannes Lorey

CONTENTS

ABSTRACT

Redis is a versatile key/value store which is hosted primarily in-memory. We examine how it performs in the *Yahoo! Cloud Serving Benchmark* (YCSB) when run on Amazon Web Services infrastructure. Benchmarks show that, while it is a great in-memory database, its performance degrades dramatically when writing to disk.

⁂

1 YCSB IN A NUTSHELL

Traditionally, there have been a number of benchmarks for relational databases but these are at best ill-suited for the non-relational landscape as such. The *Yahoo! Cloud Serving Benchmark*[1] tries to alleviate that problem by providing a generic framework to induce load on a data store.

The assumptions on a database backend are then basically that it provides a two-tier document mechanism and the CRUD type of operations, ie., creation, retrieval, modification and deletion of individual documents with contained items. These are called *records* and *fields*, respectively, in YCSB-speak. For the built-in `CoreWorkload`, a sample record might be called `user1234567890` and contain the field `field3` with 100 random bytes; the maximum number of fields in a record is usually ten.

NB. There *is* a notion of a third tier: tables. The default table is called `usertable` and is usually not changed.

2 INTRODUCTION

Redis features sophisticated operations on its built-in datatypes, namely strings, lists, sets, sorted sets, and hashes.

It trades in availability for consistency and partition tolerance but argues that all operations are just fast enough because of its in-memory nature. [2]

For a more in-depth discussion of Redis' general features check out *Redis, from the Ground Up*. [3]

2.1 History

Let me diverge shortly on Redis' history to give you an understanding of how it all came together.

Back in 2008 LLOOGG, an Internet startup offering services à la Google Analytics in a real-time fashion, was powered by a traditional MySQL setup. Salvatore "antirez" Sanfilippo, its CEO and author of hping[1] and Jim[2], was unsatisfied with their high server load. He first deployed Redis in June 2009 and said it to be *"working very well for the application domain it was conceived for. [They] never experienced any stability problem* [...]*."* [4]

After its 1.0 release the community continued to grow and the project matured rapidly. Salvatore Sanfilippo and Pieter Noordhuis, a long-time contributor, are being sponsored by VMware, Inc. since March 2010.

2.2 Persistence

Redis is designed as an in-memory database and thus supposed to keep the entire key space in memory during operation. When terminated (willfully or inadvertently, fex. by a system crash) all data will be purged from memory. Such volatility is not suitable for production setups. In order to achieve data persistence and fault tolerance, Redis uses periodic *snapshots* and append-only *journaling* (AOF).

Performance is usually favored over memory footprint: in-memory representations of values are much larger than Redis dumps (`*.rdb`). They are usually stored in a hybrid manner to speed up certain queries.

Internally, the key space is a huge hash table. When such a data structure reaches its capacity limits it is said to degenerate and lose its constant-time lookup performance. It consequently needs to be resized and rehashed — that means the number of buckets is adjusted and all items are reorganized into their new buckets. Incremental rehashing helps Redis to keep the performance

[1] a command-line oriented TCP/IP packet assembler/analyzer, see hping.org

[2] a small-footprint implementation of the Tcl programming language, see jim.berlios.de

penalties of such an operation low by temporarily provisioning two hash tables and slowly migrating the keys during normal operation. [5]

2.3 Virtual Memory

Excess data is handled by a handcrafted virtual memory implementation since the 2.0 release; the rationale for which can be aptly summarized as "Salvatore is much brighter than all of Microsoft Research." On a more serious note, they rolled their own swapping mechanisms because Redis strongly fragments its memory space. Pages provided by the operating system do not match Redis objects particularly well and caches will usually end up cold. [6]

The VM implementation imposes certain restrictions upon data sets which do not make it suitable for all use case scenarios: the global hash table needs to stay in-memory at all times and cannot be swapped out. If an object is designated to be swapped its value needs to be written out in its entirety. Imagine you have a large object, for example a sorted set, with a non-uniform access distribution on its elements. Such an object would never be swapped out, even though the rarely accessed parts of it probably could.

While the Redis documentation is all abuzz virtual memory and its various tweaking options[3], Redis core developer Pieter Noordhuis recently expressed his concerns and the future directions for on-disk storage in Redis[4]:

> *the vm has not been working out as well as expected* [...] *so instead of continuing to support an ill working feature, it is dropped* [...] *for the better of the overall project* [...] *we may investigate going to disk at a later point in time, but this will have larger overall impacts* [...] *so for now, RAM-only*

2.4 Replication

Redis supports horizontal scaling through master-slave replication. Redis 3.0 promises to bring a whole new suite of replication features to the table with its `redis-trib` tool. [7]

3 RELATED WORK

Redis comes with a benchmark utility itself: `redis-benchmark`. It is used for continuous feedback how a particular feature fares in terms of query performance.

[3] For the 2.4 release, the VM implementation has officially been deprecated.
[4] http://redis.hackyhack.net/2011-07-26.html#598/h598,600,601,603,604

J. Ramirez has compiled a catalog of useful configuation tweaks for deployment on the Amazon Elastic Compute Cloud. [8]

4 CONNECTOR

A naïve mapping of YCSB documents to Redis would generate some unique string from a record/field pair, so that, eg., `user1234567890`'s `field3` would end up in the Redis key `user1234567890:field3` as a normal string object.

Upon closer investigation, it turns out Redis has a much better fit for YCSB documents in store (no pun intended): hash tables. All records can be mapped to an individual hash and run just fine. For example, the previously mentioned query is satisfied by:

```
HSET user1234567890 field3 "value"
=> OK
HGET user1234567890 field3
=> "value"
```

This actually plays down the full strength of the benchmark's default workloads, where a single read operation usually tries to fetch a set of fields. Fortunately, Redis directly offers a command to retrieve multiple fields of a hash, called `HMGET`. For a full-record read we can utilize the `HGETALL` command.

Insert and update operations can be implemented in a similar fashion, using `HMSET`.

Records are deleted with an unqualified `DEL` call which does not care about the internal object type of the key.

4.1 Tables

Every YCSB call supplies a table name arranging where the supplied document is to be inserted. Redis, though, does not support named tables; it only has a fixed number of databases. The connector should map between names and indices— and could do so lazily and on-demand —and switch to the determined database using the `SELECT` command. For the `CoreWorkload`, supporting only one table in the default database is fine and our connector does not need to incur any such overhead.

The exact amount of databases available to a Redis instance can be configured through the `databases` directive before launch. Redis' memory footprint is not affected by the number of databases.

4.2 Scans

Scan operations over a key space pose another challenge in implementing the YCSB API. The built-in Voldemort connector goes out of its way and just raises a warning (and happens to distort the benchmark in its favor):

> Voldemort does not support Scan semantics

Redis, though, is not only a key/value store — it also supports a couple of intricate data structures. Among lists, sets and the previously mentioned hashes we are especially interested in sorted sets. A sorted set is a a collection of items where each member has an associated *score* determining its position in the set.

Ordering and value uniqueness make sorted sets a perfect fit for indices. Under the hood, sorted sets are stored like normal sets: as a hash table. Hash tables provide us with membership tests in $\mathcal{O}(1)$ amortized time but render range scans $\mathcal{O}(n)$.

For efficiency reasons, sorted sets are also stored as skip lists in memory. Skip lists are much like linked lists but offer to skip a number of elements when performing a search by storing far-reaching pointers. Conceptually, this is not far from spanning a balanced tree over the list. It yields performance characteristics close to amortized $\mathcal{O}(\log n)$ for range scans which is much more favorable to a full-table scan.

⁂

We build an index in the following way. Whenever a new key arrives (ie., for any insert operation):

1. Pick an index key. We statically choose `_indices` for now; in any real world scenario we would determine that based on our record type.
2. Calculate a hash for that particular, newly-inserted key. This could probably be some unique ID or the byte value of the first three letters; our primitive implementation uses Java's `hashCode` method. Designing a clever hashing algorithm is the crux here: for our `CoreWorkload` it would be clever to pick the user ID (remember, records are always named à la `user1234567890`) but that approach would not scale to other workloads. The exact implementation is not too important anyways as we just care about any scan semantics at all — it matches the desired performance characteristics asymptotically.
3. Insert key into sorted set through `ZADD`. While updating the hash table can be done in amortized constant time, seeking the right position in the skip list requires logarithmic effort.

Removing keys is analogous.

Subsequent scan operations are trivial. Scans always supply a starting key and a number of following records they want to read. We calculate the starting key's hash as before and issue a `ZRANGEBYSCORE`. That particular command requires a start and end score to narrow down the selection and optionally allows an SQL-esque `LIMIT` keyword to page through the results. We supply an open-ended, inclusive range beginning at the determined hash and fetch the first `recordcount`-sized page.

Indexing can be quite a performance hit as it degrades inserts from $\mathcal{O}(1)$ to $\mathcal{O}(\log n)$. Index maintenance carries no weight for most benchmarks, though, as insert operations are only part of the load phase which is excluded from the measurements. For this part we observed a 50 % performance overhead to each insert operation though.

⁂

5 SETUP

YCSB comes with six predefined workloads, all resembling another more or less realistic real world scenario. We executed them in the following order:

1. Load with workload A
2. Run workload A
3. Run workload B
4. Run workload C
5. Run workload F
6. Run workload D
7. Run workload E

32-bit builds of Redis only support up to 4 GB of memory (as per operating system restrictions), requiring virtual memory for the other portion of the dataset. 64-bit builds do not expose such restrictions but suffer from doubled pointer size. Interestingly, dumps and journals are cross-compatible.

In isolated, up-front tests, VM degraded performance by about 50 %; AOF journaling caused another 20 % blow. Tuning `hash-max-zipmap-value` to 100— the size of a YCSB field —resulted in 50 % memory savings[5].

Background saving may cause intermittent client failure. This is unattractive for the load phase (considering that the YCSB client does not cope with

[5] For a couple of documents, the database consumed 1.62 MB instead of 2.54 MB (including the 534.61 KB initial data set size).

retransmission too well). For high-load bulk operations, one should always disable snapshotting[6].

5.1 Scalability

Redis only supports *master-slave* replication. All changes made to a master node are propagated to all connected slaves[7] but not the other way around.

The YCSB benchmark is ill-suited for such a scenario as it would render all documents inconsistent unless you only, exclusively talk to the master. Redis proposes slaves to be used for read-only queries such as `SORT`, or redundancy (since a slave can pitch in for a corrupting master at any time).

⁂

6 RESULTS

To fully leverage Redis' in-memory capabilities we first run the YCSB benchmark on a *High-Memory Double Extra Large* instance[8].

MACHINE High-Memory Double Extra Large (`m2.2xlarge`)
MEMORY 34.2 GB
PROCESSOR 4 cores, 64-bit
OPERATING SYSTEM Ubuntu Natty Narwhal
AVAILABILITY ZONE US-East 1b
REDIS 2.0.1
ITEMS 2,000,000
OPERATIONS 100,000

During the load phase we achieve a pretty stable 430 ops/sec in a single-threaded deployment. Multi-threaded setups[9] quickly reached a tipping point of over 9000 operations per second at about 16 threads. 28 threads seemed like the sweet spot between client and server scaling.

Memory utilization scaled linearly with the stored data set size.

Scans (workload E, which has been trimmed to 10,000 operations) exhibit quite peculiar performance characteristics: 300 ops/sec tops, with a constant

[6] Periodic snapshotting is activated by default. Removing all `save` directives from the configuration circumvents snapshotting completely.

[7] The mechanism is incredibly simple: after transferring a dump file of the current master state, which is usually created for persistence anyways, a slave is just being proxied all issued commands. See redis.io/topics/replication for details.

[8] See aws.amazon.com/ec2/instance-types/ for all available types.

[9] The number of threads is configued by the `-threads` parameter to the YCSB client.

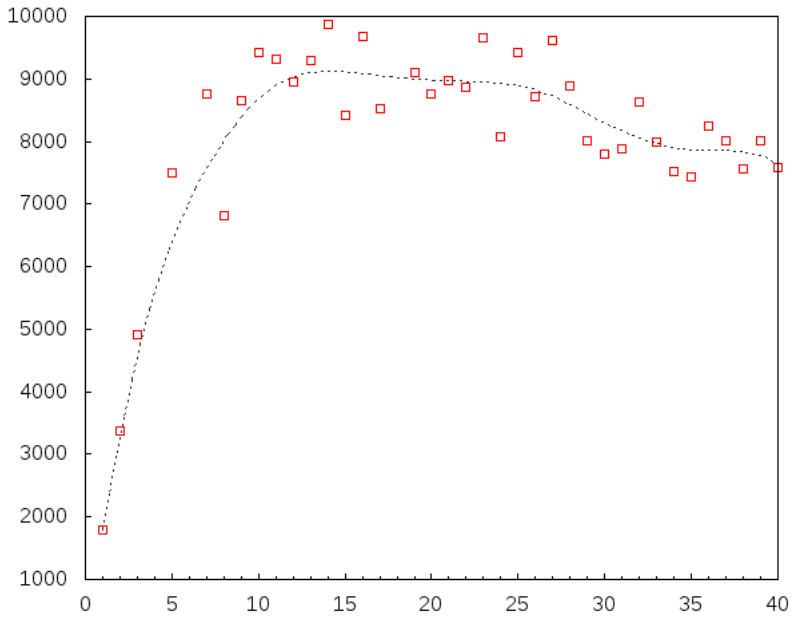

Figure 1: throughput in ops/sec relative to # of threads

latency of about 100 milliseconds. Associated inserts could not break that throughput limit, too, but only experienced about 5 milliseconds of latency.

The latency plots (Fig. 1) show how the server was vastly underutilized. In an overload scenario, the latency would scale worse than linearly.

The client machine was under a constant load factor of 6 (meaning 600 % processor demand) when occupied by 40 threads. A rehearsal of the measured throughput cap identified the client machine as the culprit — two clients reached similar throughputs, individually, when run concurrently.

workload	A		C	
concurrent	✘	✔	✘	✔
throughput (ops/sec)	16,000	12,200	13,800	9,800
latency (msec)	1.6	2.2	1.9	2.8

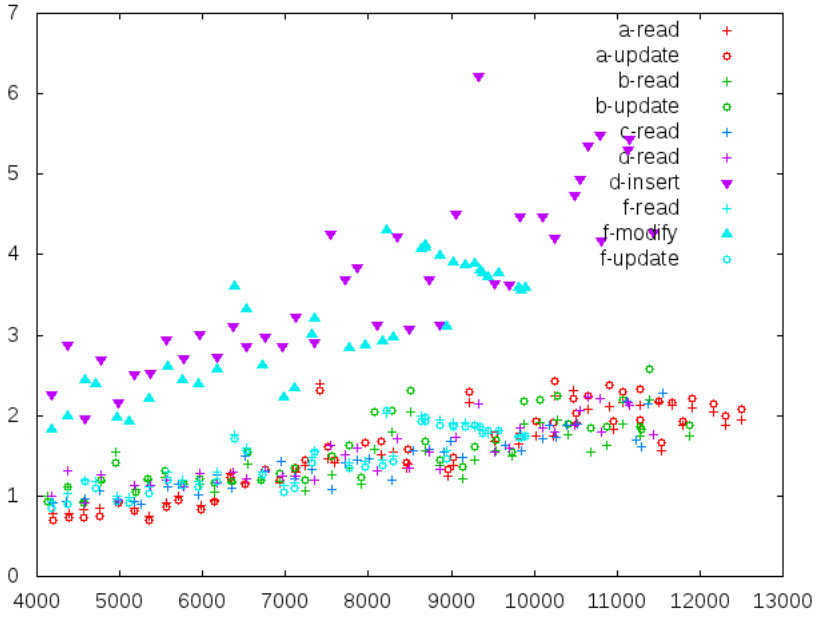

Figure 2: latency (ms) vs. throughput (ops/sec)

6.1 System under Load

Another interesting setup is the *Extra Large* instance type.
MACHINE Extra Large (`m1.xlarge`)
MEMORY 15 GB

As these machines are not equipped with enough memory to keep the whole data set in-memory we need to enable swapping:

```
vm-enabled yes
vm-max-memory 8g
vm-page-size 1000
vm-thread 5
```

Redis occupies about 50 % of additional RAM[10] with hybrid data structures (ie. redundant data) to speed up certain queries. If Redis is allotted 10 GB of memory it will easily burst the memory boundary with auxiliary data structures and invoke the OOM killer.

[10] For our 20 GB data set we ended up with a resource usage of about 29 GB. The 8 GB setting left less than 100 MB free on the smaller machine, with no other system services consuming a considerable amount of memory.

Hashes — or all Redis objects for that matter — can only be swapped in their entirety. Now with the default page size of 32 bytes that would result in cache thrashing of sorts and require multiple, semantically unrelated pages to be swapped in for a single record field. A virtual memory page should accomodate an entire YCSB document.

The thread count allows the scheduler to saturate all cores by always having an enqueued job. The swap file was allocated on the instance store for performance reasons.

⁂

Latencies are several orders of magnitude worse compared with their in-memory counterparts and hit a real barrier this time around.

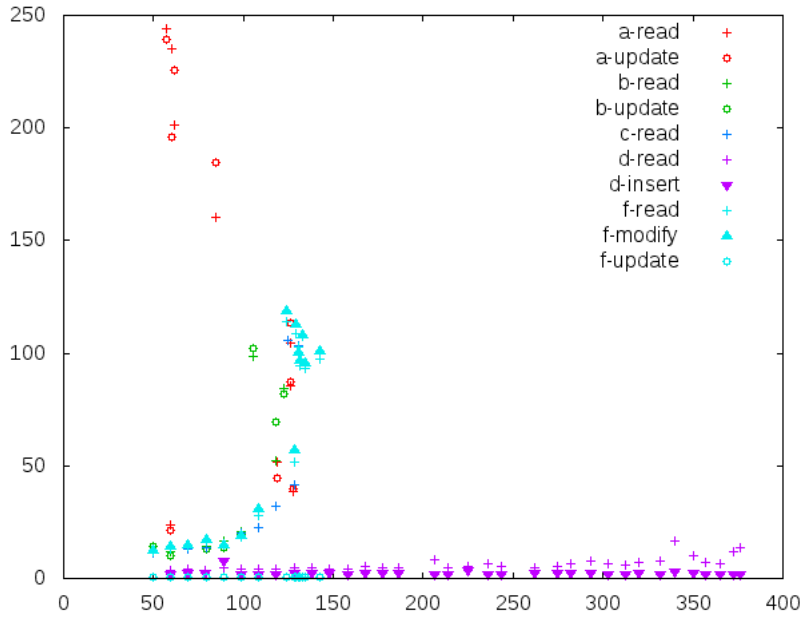

Figure 3: latency (ms) vs. throughput (ops/sec)

Scans (workload E) were as abysmal as expected: swapping the index did take way too long. Tests usually timed out abruptly after a couple of operations (literally).

Workload D performed extraordinarily well in contrast to the other operations (which quickly degraded at about 150 ops/sec) and reached up to 400 ops/sec where it simply stopped scaling in terms of throughput, not latency. Caching might have been a big factor in this benchmark, as records are

accessed repeatedly. It is the only benchmark which uses the *latest* request distribution strategy.

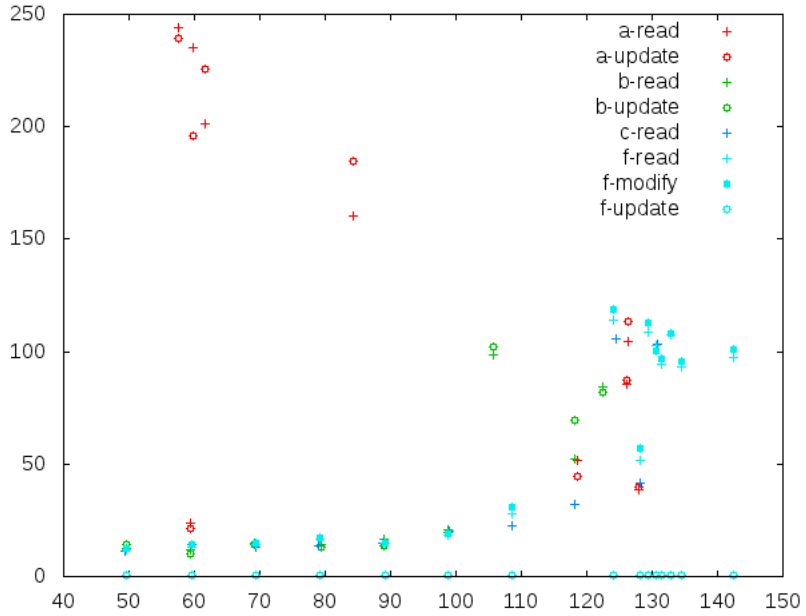

Figure 4: latency (ms) vs. throughput (ops/sec) without workload D

7 CLOSING REMARKS

Redis is an incredibly lean and fun tool to use. They have a momentum that other databases can only dream of — if there is a valid strategy to solve any given problem, Redis has probably gone down that route and tackled it full-front.

It is a great in-memory database but has no clue how to properly handle a disk device. Even solid-state drives only provide limited performance improvements. [9] Enough memory is crucial in its operations.

The `INFO` command and the shipped `redis-cli` tool are incredibly helpful in quickly validating hypotheses about Redis' status.

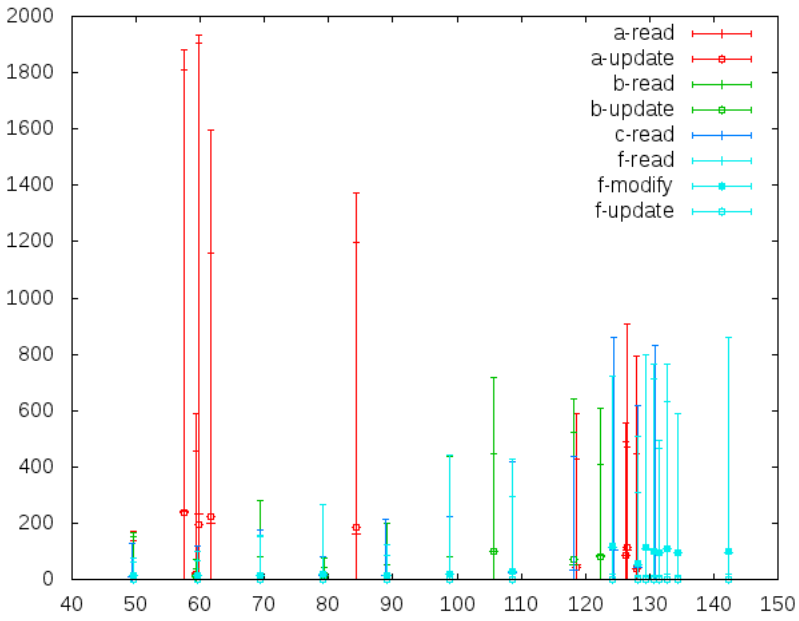

Figure 5: latency (ms) vs. throughput (ops/sec) with variations

And a warning for the kids — don't try this at home: stopping a 20 GB instance of Redis is a time-consuming task and might well lock up your machine.

REFERENCES

[1] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152

[2] N. Hurst. (2010, March) Visual guide to NoSQL systems. [Online]. Available: http://blog.nahurst.com/visual-guide-to-nosql-systems

[3] M. J. Russo. (2010, October) Redis, from the ground up. [Online]. Available: http://blog.mjrusso.com/2010/10/17/redis-from-the-ground-up.html

[4] S. Sanfilippo. (2009, June) How Redis is behaving in production with lloogg.com. [Online]. Available: http://groups.google.com/group/redis-db/msg/17c21c48642e4936
[5] ——. (2010, April) Redis weekly update #4: Non blocking rehashing. [Online]. Available: http://antirez.com/m/p.php?i=209
[6] ——. (2010, February) Redis virtual memory: the story and the code. [Online]. Available: http://antirez.com/post/redis-virtual-memory-story.html
[7] ——. (2011, January) Redis cluster. [Online]. Available: http://redis.io/presentation/Redis_Cluster.pdf
[8] J. Ramirez. (2011, January) Prime time redis 101: Set up. [Online]. Available: http://jramirez.tumblr.com/post/2589232577/prime-time-redis-101-set-up
[9] S. Sanfilippo *et al.* (2010, October) Redis and SSD. [Online]. Available: http://groups.google.com/group/redis-db/browse_thread/thread/5b6d7d913fff9c8f