ArangoDB

the **multi-model** NoSQL DB

Scaling ArangoDB to Gigabyte/s bandwidth on Mesosphere

We show that an ArangoDB cluster with 640 vCPUs can sustain a write load of 1.1M JSON documents per second which amounts to approximately 1GB of data per second, and that it takes a single short command and a few minutes to deploy such an 80 node cluster using the Mesosphere DCOS.

In this white paper we collect all the more detailed information which has been left out of the corresponding blog post for brevity.

Introduction

The purpose of this work is to show that an ArangoDB cluster scales out well when used as a distributed document (JSON) store. It complements the benchmark series here, which looked into the performance of a single instance of ArangoDB as a multi-model database, comparing it with pure document stores, graph databases and other multi-model solutions. "Scaling out well" means two things, one is a linear growth of performance, the other is ease of deployment and operational convenience, which we achieve by being the first and so far only operational database that is fully certified for the Mesosphere DCOS.

We would like to emphasize right away that this is a preview of Version 3.0 of ArangoDB, which is going to be published in Q1 of 2016 and which will contain a lot of cluster improvements with respect to functionality, stability and performance. Now, in November 2015, we are using the current development version and so everything we do will already work in Version 2.8 to be released soon. This version will include the next iteration of our integration with the

Mesosphere DCOS, making it possible to deploy an ArangoDB cluster now with asynchronous replication and automatic failover on a Mesosphere cluster with a single command within 10 minutes.

For this benchmark we only investigate single document reads, writes and a mixed load. Ideally, we would like to see the throughput grow linearly with the number of vCPUs we throw in whilst keeping the latency at bay. Obviously, we now expect a performance level that only a distributed system can sustain, ranging into the area of Gigabytes and millions of independent requests per second. Thus, we also have to invest a larger effort on load generation, as well as on network, computing and storage infrastructure, for details see the "Background" and "Test setup" sections below.

We would like to put the experiments at hand into a slightly broader context. In the age of Polyglot Persistence, software architects carefully devise their microservice based designs around different data models like key/value, documents or graphs, considering the different scalability capabilities. With ArangoDB's native multi-model approach they can deploy a taylor made cluster- or single-instance of ArangoDB for each microservice.

In this context it is important to emphasize that the multi-model idea does not imply a monolithic design! To the contrary, some services might require a large document store cluster, others might employ a single server graph database with asynchronous replicas and a load balancer for reads, say. Even services using a mixture of data models are easily possible. The multi-model approach avoids the additional operational complexity of using and deploying multiple different data stores in a project. One can use a single uniform query language across all instances and data models, and has the additional option to combine multiple data models within a single service.

We have published all test code at
https://github.com/ArangoDB/1mDocsPerSec.

Background of this benchmark

Typical use cases for a distributed document store would be that single document operations are used as the atomic operations, and so we will use documents with 20 attributes. We aim to sustain a write load of a million 788 byte JSON documents per second in the largest cluster configuration. However, we are interested more in the scalability than in the absolute throughput numbers.

A distributed system capable to sustain such traffic needs an infrastructure that is well-equipped with

- CPU power for processing
- RAM for caching and indexing
- disk space to make writes durable
- network traffic capacity
- disk I/O bandwidth, preferably in the form of fast local SSDs

and there must not be a shortage in any of these to achieve peak performance. To get all this, we use a deployment on Amazon Web Services (AWS), which at the time of this writing has the most mature and reliable offering for such endeavors. We use a Cloud Formation Template by Mesosphere to set up and scale a Mesosphere DCOS cluster, on which we can easily deploy an ArangoDB cluster as well as the load servers. For details see the "Deployment" section below.

Details of the test setup

To simulate such a high load we use a C++ program using libcurl to perform the HTTP/REST requests, which measures the latency for each request and does the statistical analysis on it. We start multiple instances of this client on several machines to produce a large amount of traffic. We test read only, write only and 50% read/50% write in a pseudo-random pattern. The tested loads are typical for an application dealing with single document reads and writes.

We use AWS instances of type i2-2xlarge for the servers that actually run ArangoDB instances, and type m3-2xlarge for the load servers. The former have very fast local SSDs, 8 vCPUs each and 61 GB of RAM, and are optimized for high network, CPU and I/O loads. The latter are more generic general-purpose instances, which are cheaper. We use half as many instances for the load servers as for the ArangoDB servers and modify the number of client connections to load each ArangoDB endpoint with approximately 80 connections, equally distributed over the load servers.

The JSON documents we read and write are all different but with a similar structure, they have 20 string attributes each with length 26, so that the total document needs around 800 bytes. In the beginning, we produce 100,000 such documents and the reads work on these with a random access pattern. The writes add further similar but different documents. We measure for one minute of load and report the average number of documents read or written per second over the

complete cluster and all clients. At the same time we measure the time each request needs (latency) and report the median and the 95% percentile.

We first perform 30s of pure reads, then a 30s of pure writes and finally a 30s with a mixed load of 50% reads and 50% writes. We have repeated this with varying numbers of TCP connections from each load server. More connections generally mean higher latencies, but up to a number of about 80 incoming connections for each ArangoDB node, the throughput grows with the number of connections. For each cluster size, we only report about a certain number of connections close to 80 incoming connections for each ArangoDB node.

To collect results and coordinate the tests we use a single further ArangoDB instances outside of the cluster, which we call the "result server".

Each load server runs a single Docker image in which a program called "the Waiter" runs that queries the result server every half second. Thus, as soon as a test job is created on the result server, all load server instances start to work within a second. The job data includes information about the ArangoDB cluster, the particular test case, the number of TCP connections to open, and in particular the test duration. As soon as a job has been completed, the results (throughput and latency) are sent to the result server, such that one has the information about what happened in a central place. Therefore, we can control the complete testing procedure with a single JavaScript program running on the result server. This approach makes the deployment of the load servers particularly simple.

The initial setup of the data to read in the ArangoDB cluster works with the same technology, except that only one of the load servers actually performs the initialization procedure. The coordination about which load server does the work is done via the result server as well.

Details for the deployment on Mesosphere

With the new Apache Mesos framework/service for ArangoDB the deployment is phantasically convenient, provided you already have a Mesos or Mesosphere cluster running. We use the `dcos` command line utility, which in turn taps into the ArangoDB framework and the `dcos` subcommand.

Starting up an ArangoDB cluster is as simple as typing

```
dcos package install --options=config.json arangodb
```

For this, we use a relatively short JSON configuration file similar to the following:

```
{ "arangodb": {
    "async-replication": true,
    "nr-dbservers": 80,
    "nr-coordinators": 80,
    "framework-cpus": 0.5,
    "role": "arangodb",
    "principal": "pri",
    "minimal-resources-agent": "mem(*):512;cpus(*):0.5;disk(*):512",
    "minimal-resources-dbserver": "mem(*):8192;cpus(*):3;disk(*):8192",
    "minimal-resources-secondary": "mem(*):8192;cpus(*):1;disk(*):8192",
    "minimal-resources-coordinator": "mem(*):8192;cpus(*):3;disk(*):8192",
    "secondaries-with-dbservers": true,
    "docker-image": "arangodb/arangodb-mesos:devel"
  }
}
```

This essentially specifies the number of ArangoDB instances to use, switches on asynchronous replication and configures necessary minimal resources. The ArangoDB framework will use the new persistence primitives (since Apache Mesos 0.23) to make dynamic reservations (read here about Reservations) and ask for persistent volumes (read here about Persistent Volumes) for the actual database servers. The setup of asynchronous replicas works fully automatically.

Apache Mesos manages the cluster resources and finds a good place to deploy the individual "tasks". If the machines in your Mesos cluster have the right sizes, then Apache Mesos will find the exact distribution you intend to use.

The DCOS command

```
dcos arangodb endpoints
```

shows us the addresses, under which the ArangoDB endpoints can be reached (from within our Mesosphere cluster). For each deployment, we copy this output to the configuration used in the JavaScript program running on the result server. This is the way in which the load servers eventually find out where to put their load.

We use the following `curl` command to deploy the load servers to our Mesosphere cluster:

```
curl -X POST http://<ADDRESS-OF-MESOS-MASTER>/service/marathon/v2/apps \
     -d @load.json -H "Content-type: application/json"
```

and use a file `load.json` as in this example:

```
{
  "id": "loadserver",
  "cpus": 7.5,
  "mem": 8192.0,
  "ports": [],
  "instances": 40,
  "args": [],
  "env": {},
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "neunhoef/waiter",
      "forcePullImage": true,
      "network": "HOST"
    }
  },
  "acceptedResourceRoles": [
    "slave_public"
  ]
}
```

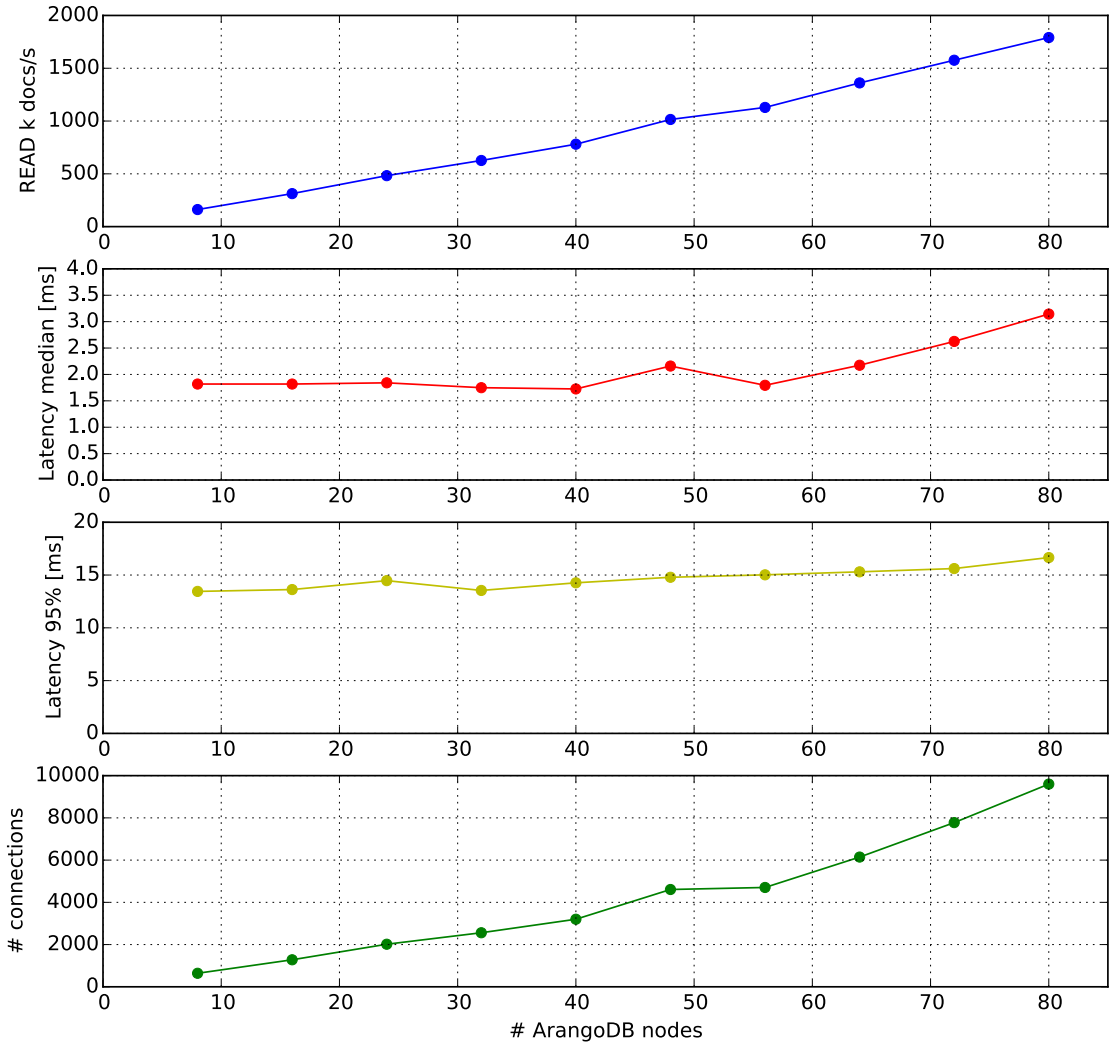
Note that no further information about the test protocol, the servers running ArangoDB etc. are needed here, since all this is distributed via the result server.

For the different cluster sizes, we use the Amazon Cloud Formation template to scale the Mesosphere cluster up or down and then in turn deploy the ArangoDB cluster and the load servers conveniently.

Detailed results

Here we show all results as graphs and tables. Throughput is in thousand documents per second for the whole cluster. Latency times are in milliseconds, we give the median (50% percentile) and the 95% percentile. Furthermore, we plot the number of connections used. All graphs in a figure are aligned horizontally, such that the different values for the same cluster size appear stacked vertically. The actual absolute numbers follow in the table below each plot.

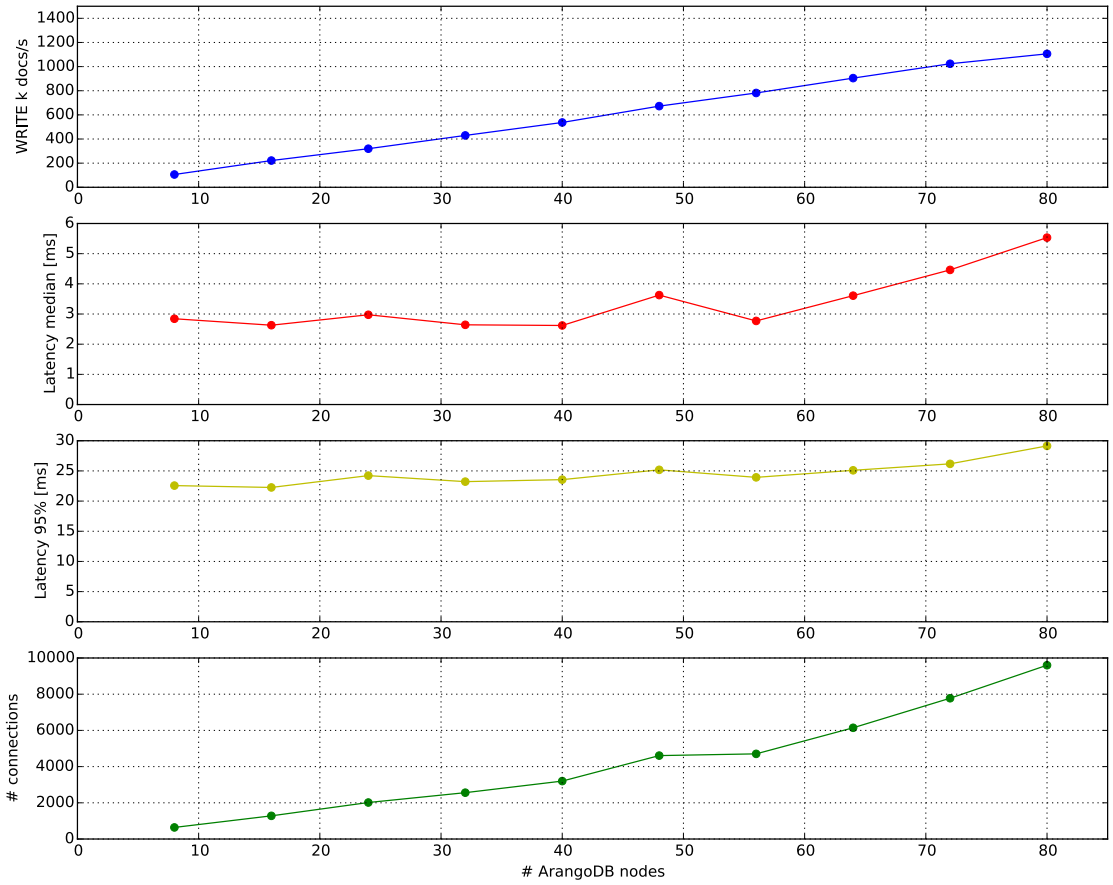
We start with the 100% read test:



Throughput, latency and number of connections for 100% reads.

ArangoDB	Load	Connections	Reads [kDocs/s]	Median lat.[ms]	95% lat.[ms]
8	4	640	161.727	1.818	13.449
16	8	1,280	312.604	1.818	13.625
24	12	2,016	482.669	1.841	14.470
32	16	2,560	626.485	1.749	13.540
40	20	3,200	780.554	1.725	14.264
48	24	4,608	1,014.850	2.159	14.788
56	28	4,704	1,128.881	1.794	15.020
64	32	6,144	1,360.975	2.174	15.303
72	36	7,776	1,575.976	2.625	15.614
80	40	9,600	1,790.919	3.144	16.656

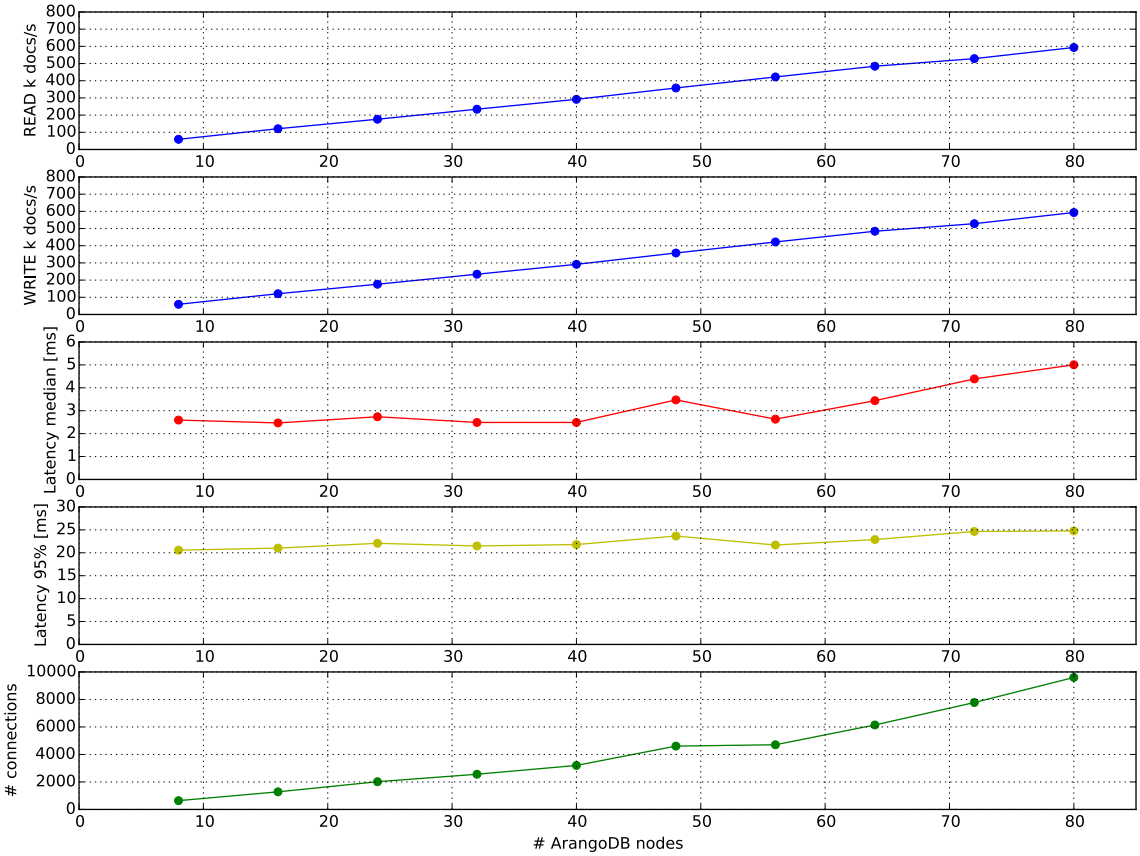
We continue with the 100% write test:



Throughput, latency and number of connections for 100% writes.

ArangoDB	Load	Connections	Writes [kDocs/s]	Median lat.[ms]	95% lat.[ms]
8	4	640	105.881	2.843	22.567
16	8	1,280	221.468	2.629	22.260
24	12	2,016	319.681	2.975	24.220
32	16	2,560	429.624	2.642	23.224
40	20	3,200	537.005	2.620	23.553
48	24	4,608	672.502	3.627	25.179
56	28	4,704	781.421	2.771	23.932
64	32	6,144	904.561	3.608	25.100
72	36	7,776	1,023.445	4.462	26.171
80	40	9,600	1,105.655	5.532	29.141

We conclude with the 50% read and 50% write test, where we plot both the reads and the writes:



Throughput, latency and number of connections for 100% writes.

ArangoDB	Load	Conns.	Reads [kDocs/s]	Writes [kDocs/s]	50% lat.[ms]	95% lat.[ms]
8	4	640	59.146	59.124	2.592	20.577
16	8	1,280	120.760	120.718	2.467	21.019
24	12	2,016	175.979	175.910	2.736	22.070
32	16	2,560	234.317	234.233	2.489	21.488
40	20	3,200	291.813	291.702	2.488	21.800
48	24	4,608	357.829	357.674	3.472	23.657
56	28	4,704	422.007	421.864	2.631	21.695
64	32	6,144	484.486	484.285	3.438	22.881
72	36	7,776	528.533	528.263	4.387	24.651
80	40	9,600	593.460	593.131	5.004	24.786

We observe very good scalability indeed: Already with the second largest cluster size of 72 nodes we hit our performance aim of 1,000,000 document writes per second using 576 vCPUs. Pure read performance is significantly better, which is not really surprising, since write operations are more expensive. The throughput for the mixed load is between the read and the write load, as is to be expected.

The latency is reasonable, the median is between 2 and 5 ms, the 95% percentile is still below 30ms, even for the pure write load. Looking at these latency numbers, we as developers of ArangoDB immediately spot some more homework to do, namely to decrease the variance in latency.

We have included the total number of connections to the distributed database in the tables and plots. In the graphs one can see that this total number of connections does not grow linearly. This is because our setup requires that every load server opens an equal number of connections to each ArangoDB node. Therefore, for the larger clusters we cannot vary the total number of connections continuously. For example, for the largest configuration we could only choose between 2 or 3 connections to each ArangoDB node per load server. Choosing 2 would give a total of $40 \cdot 80 \cdot 2 = 6{,}400$ connections, choosing 3 would give a total of $40 \cdot 80 \cdot 3 = 9{,}600$ connections. That is why the latency for the larger clusters increases slightly, because as a rule of thumb we observe that both throughput and latency increases with the number of connections. This is another area where we have identified possible improvements.

Details about network and I/O bandwidth

Note that for example for the pure write load this means a peak total network transfer of 0.95 GB per second from the load servers to the distributed database and 140 MB per second for the answers. This comes from 788 bytes for the JSON payload and 139 bytes for the HTTP headers, and 133 for the HTTP responses.

Roughly the same amount of network transfer happens within the ArangoDB cluster for the primary storage, since each JSON document write has to be forwarded from the client facing instance to the actual database server holding the data. For the asynchronous replication this amount must be transfered another time, however, this uses batched transfer and can thus essentially reduce the transfer to the amount of data stored, which is about 0.81 GB per second. If we add all this up, then we find a total bandwidth for the network switches of about 2.98 GB per second in the 80 node pure write case.

Furthermore, all this data has to be written to durable storage. One of the documents used needs around 872 bytes in the data files (including all overheads for alignment and such), so the combined I/O write rate to SSD across all primary database servers is 0.90 GB per second, and the same amount has to written by the asynchronous replicas. Due to the internal write ahead log of ArangoDB we have to multiply this number by 2. Therefore, this means that each of the 80 ArangoDB nodes has to write around 46 MB per second of pure application data to SSD, not taking into account any write amplification! That is the reason that we use the rather expensive i2-2xlarge instance types, because only these offer enough I/O bandwidth to pull off the presented benchmark results.

Details about throughput per vCPU

The peak throughput performance is 1,105,655 documents per second for the pure write load and each document uses about 788 bytes in JSON format. We have achieved this with a cluster size of 80 instances, which amounts to 640 vCPUs. Thus the peak throughput per vCPU is around 1,730 documents per vCPU. A similar calculation applies for the pure read and for the mixed load.

This is achieved whilst keeping the median of the latency below 5ms and the 95% percentile below 30ms.

If we compare this result to other published benchmarks like the one by FoundationDB, Netflix using Cassandra, Google using Cassandra, Aerospike or Couchbase, we find that our ArangoDB cluster is up there with the pack.

For example, the FoundationDB benchmark achieved 14.4M key/value writes per second and bundled them into transactions of 20. A sensible comparison here would be to compare their number of transactions per second (14.4M/20=720,000) to our number of single document writes. They used 960 vCPUs and thus achieve around 750 write transactions per vCPU. As far as we understand their description, they do not do any replication, though.

Netflix in their Cassandra benchmark claimed 1.1M writes per second using Datastax's Enterprise edition of Cassandra and a replication factor 3 with the consistency level "One", which is very similar to our asynchronous replication. They used 1,140 vCPUs and therefore this amounts to writing 965 documents per second and vCPU with a median latency of 5ms and a 95% percentile of 10ms. We did not find anything about the size of their individual data items in the publication. Google's benchmark uses more vCPUs, more consistency but only achieves fewer documents per second and vCPU.

Another example is the Couchbase benchmark, they can sustain 1.1M writes per second when using a replication factor of 2 and use 800 vCPUs for their cluster, using local SSDs. This means they achieve 1,375 documents per second with a median latency of 15ms and a 95% percentile of 27ms.

Last but not least, the Aerospike benchmark claimed 1M writes per second which amounts to 2,500 documents per second and vCPU with a median latency of less than 16ms. Their JSON documents have only a size of 200 bytes and they do not mention any replication or disk synchronization. They seem to optimize for low costs by using slow disks.

All this means that the ArangoDB results presented here actually outperform the competition considerably in the document writes per vCPU comparison.

Publication of the test code

All the test code is published in this repository:

https://github.com/ArangoDB/1MDocsPerSec

and this is also used to produce the Docker image

```
neunhoef/waiter
```

which can be downloaded from the Docker Hub. We welcome comments and suggestions as well as other people reproducing our results.

References

1. ArangoDB:
   https://arangodb.com

2. Benchmark:
   https://www.arangodb.com/nosql-performance-blog-series/

3. FoundationDB:
   http://web.archive.org/web/20150320063641/
   http://blog.foundationdb.com/databases-at-14.4mhz

   Netflix C*:
   http://techblog.netflix.com/2014/07/revisiting-1-million-writes-per-second.html

   Google C*:
   http://googlecloudplatform.blogspot.com/2014/03/cassandra-hits-one-million-writes-per-second-on-google-compute-engine.html

   Aerospike:
   http://googlecloudplatform.blogspot.com/2014/12/aerospike-hits-one-million-writes-Per-Second-with-just-50-Nodes-on-Google-Compute-Engine.html

   Couchbase:
   http://blog.couchbase.com/couchbase-server-hits-1m-writes-with-3b-items-with-50-nodes-on-google-cloud

4. Mesos reservations:
   http://mesos.apache.org/documentation/latest/reservation/

5. Mesos persistent volumes:
   http://mesos.apache.org/documentation/latest/persistent-volume/

6. Blog post:
   http://mesosphere.com/blog/2015/11/30/arangodb-benchmark-dcos

7. Github:
   https://github.com/ArangoDB/1mDocsPerSec