Lineate

Denis Nelubin, Director of Technology, Lineate
Ben Engber, CEO, Lineate

Ultra-High Performance NoSQL Benchmarking

Analyzing Durability and Performance Tradeoffs

TABLE OF CONTENTS

Overview

As companies deal with ever larger amounts of data and increasingly demanding workloads, a new class of databases has taken hold. Dubbed “NoSQL”, these databases trade some of the features used by traditional relational databases in exchange for increased performance and/or partition tolerance. But as NoSQL solutions have proliferated and differentiated themselves (into key-value stores, document databases, graph databases, and “NewSQL”), trying to evaluate the database landscape for a particular class of problem becomes more and more difficult. In this paper we attempt to answer this question for one specific, but critical, class of functionality - applications that need the highest possible raw performance for a reliable storage engine.

There have been a few attempts to provide standardized tools to measure performance or other characteristics, but these have been hobbled by the lack of a clear mandate on exactly what they’re testing, plus an inability to measure load at the highest volumes. In addition, there is an implicit tradeoff between the consistency and durability requirements of an application and the maximum throughput that can be processed. What is needed is not an attempt to quantify every NoSQL solution into on e artificial bucket, but a more systemic analysis of how some of these databases can achieve under assumptions that mirror real-world application needs.

We attempted to provide a comprehensive answer to one specific set of use cases for NoSQL databases - consumer-facing applications which require extremely high throughput and low latency, and whose information can be represented using a key-value schema. In particular, we look at to common scenarios. We consider applications that have strong durability needs, where every transaction must be committed to disk and replicated in case of node failure, and we also consider applications that are willing to relax these requirements in order to achieve the

highest possible speed.

Key-value storage is of course a subset of what NoSQL databases can do. Nevertheless, it is a real-world need currently demanded by most companies in the Ad Tech industry as well as an increasing number of other applications ranging from e-commerce “flash sale” applications to social platforms to financial engines.
We plan to follow up this paper with additional studies into other major NoSQL use cases, including document-oriented applications, graph-based applications, and big data analytics. Attempting to evaluate them all together conflates too many different needs with too many different kinds of required optimizations, and prevents meaningful comparison.

Test Description

In this test, we analyzed performance characteristics of four key-value datastores: Cassandra, Couchbase (versions 1.8 and the just-released 2.0), Aerospike, and MongoDB[1]. We performed the test using a modified version of the Yahoo Cloud Serving Benchmark (YCSB) from Yahoo!
Research, which has become something of a standard for trying to measure NoSQL performance.

Our goal was to measure their suitability for classes of applications that have extremely high transactional loads and who can architect their applications

---

1. MongoDB is actually a document database. While it can be used as a key-value store, it is not specifically optimized for this scenario. However, we see significant interest from our customers in using it as a key value store, so we include it here.

around a basic set of operations in order to achieve this scale. This is a real-world problem in industries ranging from real-time bidding markets to gaming. In particular, we tested how such applications behave under hardware that would be used in the real world, i.e. non-virtualized systems with data either in RAM or stored to solid state drives, and investigated tradeoffs between raw speed and durability.

We opted for a narrow set of tests that could run against all systems, in order to provide a baseline of functionality. Our concern was to address those applications that need the absolute highest performance. Features such as secondary indexes (in Couchbase 2.0, Cassandra, and MongoDB), while valuable and worthy of study, do not directly impact the question at hand.

Similarly, at every phase of the project we elected to limit the number of variables in play. YCSB offers a range of data distributions, operations, workloads, all of which have implications downstream at the database server level. Rather than providing a bunch of numbers around a bunch of theoretical models, we attempted to narrow down on a very specific set of assumptions and optimize the systems around them. We felt such an experiment provided more actionable data than a broad but untuned set of graphs and numbers.

Why another benchmark?

There have been a few NoSQL benchmarking studies published along similar lines. YCSB was the foundation for most of these, and Yahoo! Research's 2010 paper Benchmarking Cloud Serving Systems with YCSB is excellent reading for anyone interested in the topic. A more recent study by Altoros entitled A vendor-independent comparison of NoSQL databases: Cassandra, HBase, MongoDB, Riak was

recently published in Network World. However, we felt that both of these studies, as well as others we encountered, had two major issues preventing them from addressing the questions we were trying to answer. The first was that they did not focus on specific use cases, and tried to provide a broad metric across a wide variety of problem classes. The second was that they were run on hardware with rotational drives, which seemed like a poor assumption for those companies looking to handle transactional data at the scale being discussed. The software configurations in these studies were also not necessarily optimized for the use cases considered here, and no concentrated effort was made in order to normalize the durability and replication settings across systems. In contrast, we preferred to test a narrower set of databases but in a highly optimized way. Our goal with this study was purely to determine how these key-value stores perform under extremely high loads. That meant not only using raw hardware and solid state drives, but configuring each database in question to optimally handle the workload. We did both to the best of our ability, and by soliciting input from each of the vendors.

One last major point: The YCSB tool, which is rapidly becoming the standard for benchmarking NoSQL databases, was unsuited for testing at the highest volumes. It provided limited support for scaling across multiple clients, and the code had design limitations which prevented meaningful tests from being run at the rate needed to fully load these servers. As part of this project, we altered and extended the YCSB tool and supporting scripts to overcome these limitations. We contributed this back to the community, so that others can reproduce our results more easily. Details of our changes can be found in Appendix E.

Database Selection

For the test we selected 3 common open source NoSQL databases that are in widespread use: Cassandra, Couchbase, and MongoDB. We also included the commercial Aerospike database, which is proprietary and has had less exposure than the other databases, but seemed like an excellent candidate for the test because it specifically targets high-volume processing and is optimized for SSDs.

A brief description of the databases:

1. **Cassandra** is a column family store operating under the auspices of the Apache Software Foundation. Initially developed at Facebook, its goals are availability and the ability to scale to a very large size. With flexible consistency models, its architecture is particularly conducive to write operations.
2. **Couchbase** is the company formed by the merger of two databases, CouchDB and Membase. We tested both the 1.8 version, which is purely a distributed key-value store built around the hugely popular memcached cache, and the 2.0 version, which was released during our benchmarking and adds secondary index support as well as performance improvements.
3. **MongoDB** is different from the other products in that it is primarily a document database. It has extensive support for a variety of different kinds of secondary indices, strong features around documenting and a very different approach to scaling and durability. We included it in this study because in our experience clients often consider it for similar kinds of applications.
4. **Aerospike** is another key-value store with its origins in the ad tech space. A commercial product, it positions itself as the market leader in raw performance and was a natural candidate to see if those claims were justified.

A critical thing to note is that these databases are optimized for different things. For example, Couchbase and MongoDB are intended to run on hardware where

most of the working set is cached in RAM. Aerospike, in contrast, is optimized for direct writes to SSD. For durable writes, we would expect Aerospike to dominate the performance results. These databases all make different assumptions and take different approaches to consistency / availability tradeoffs. For example, Aerospike is intended for use with synchronous replication and write directly to disk, providing strong durability and consistency. Couchbase by default takes a different approach, maximizing throughput by putting data in RAM and persisting data to disk and updating replicas asynchronously (but still providing consistency by querying the data master). Cassandra's consistency is tunable on a query-by-query basis. Our goal was to establish apples-to-apples comparisons as often as possible, but given the different database architectures, a fair amount of judgment went into each test. As a general rule, when in doubt, we relaxed consistency or durability to provide the highest numbers within the rough confines of each test. It is impossible to list all the various configuration tradeoffs that can be made within this document, but we do list the configurations used in Appendix C. In the interest of full disclosure, we should point out that Lineate has strategic and/or commercial relationships with Aerospike and 10gen (the makers of MongoDB) as well as other NoSQL vendors not included in this test. Whenever practical, we reached out to the vendors directly to confirm our choices of settings and resolve the bugs or problems we encountered. Both Aerospike and Couchbase provided YCSB plugins for their respective databases. We shared the performance results with Aerospike, Couchbase, and 10gen and incorporated their suggestions wherever possible. Aerospike sponsored the changes to YCSB, and rented the hardware to us. However, the system setup and systems administration, tools and software used, database configuration and setup, test methodology and design, and analysis were all done in a clean room fashion by our own engineers.

Methodology

As with the studies mentioned above, we used the YCSB client as the basis for all our tests. In order to preserve a common baseline as much as possible, we used the same kinds of data sets and record sizes as the other studies, but focused on a subset of workloads.

The overall approach was as follows:

1. Provision the hardware, operating systems, and environment using our best estimates of what customers will run in their data centers and the best practices, where specified, of the various databases.

2. Install a database on the 4-node cluster. Configure it optimally and ensure it is functioning as a single cluster.

3. Load a large dataset to disk (SSD) by inserting records individually but as fast as possible.

4. Perform an overall throughput test to determine the maximum load the cluster can handle, using the strongest durability guarantees practical.

5. Perform a stepwise load to determine how latency behaves as load increases.

6. Repeat the test for both read-heavy and balanced read-write workloads.

7. Repeat steps 3-6 for a dataset that fits into RAM. This will provide higher throughput at the expense of durability.

All the databases provide built-in sharding and replication, though not all support all replication modes. We set each database to store 2 copies of the data. When a node goes down, all data should be preserved in at least one place, and when it rejoins the cluster, it will take some time to get up to date copies of the replicated data. YCSB also includes functionality to support range scans to access groups of keys. We did not include such tests in our analysis because the feature is not universally supported across databases and is of questionable value in our opinion.

Client & Workload Description

Data Sets

The data was loaded to the database using the "load" phase of the YCSB tool. We used a replication factor of 2 for each database, so each record was stored two times.

Record description: Each record consisted of 10 string fields, each 10 bytes long and with a 2-byte name

Record size: 120 bytes

Key descrip ion: The key was the word "user" followed by a 64-bit Fowler-Noll-Vo hash (in decimal notation)

Key size: 23 bytes

The records were intentionally small to prevent network bandwidth from becoming a factor in the tests.

Disk-backed Data Set

This data set was purposefully sized higher than the available RAM in the cluster (after considering replication). The idea was to force a significant chunk of operations to go to disk while using small enough records to ensure network bandwidth did not become a bottleneck.

Number of records: 500 million

Total amount of raw data: approximately 60GB

For Couchbase, we used 200 million records instead. The reason for this was that Couchbase required 120 bytes of metadata per row to fit into RAM or it will not operate. The 200 million number was selected because it meant that RAM comfortably held the metadata but not the underlying dataset.

In-memory Data Set

This data set was identical in structure but one-tenth the size. The goal was that most if not all of the data are kept in RAM, which is the preferred setup for Couchbase and MongoDB.

Number of records: 50 million
Total amount of raw data: approximately 6 GB

Workloads

We ran two workloads: A balanced workload of reads and writes and a read-heavy workload. For each of the workloads, records were selected using a random Zipfian distribution. This distribution selects a small subset of popular records very frequently while the majority is hit infrequently with a roughly long tail. It is intended to approximate Internet usage where a large number of users are active at a given time and using a user's random cookie ID as the key. For each of the workloads, we began each test by running a brief warm-up period to prime any caches. This was intended to start measurements against a system that is similar to a 2unning production system.

Workload A - Balanced

Read operations: 50% **Update operations:** 50%

Workload B - Read Heavy

Read operations: 95% **Update operations:** 5%

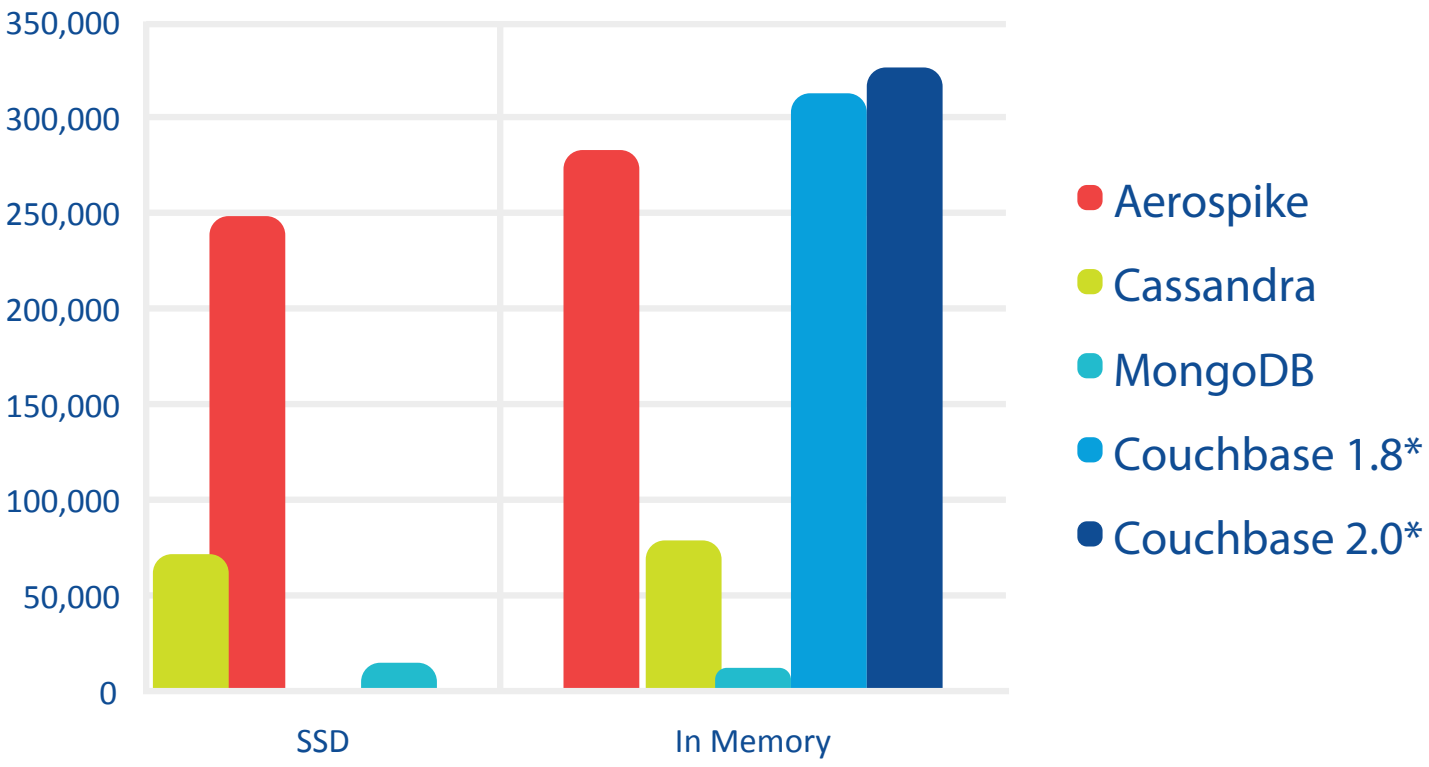
The number of operations actually performed depended on the throughput achievable on each test. Each workload performed a minimum of 10 million operations, but for very high-throughput scenarios we increased the load as high a 200 million operations to ensure a reasonable test duration.

Results

Test 1: Loading Data

The first step was to load each of the databases with the working set. We did this by running YCSB's load routine for each of the full data sets. This performed a series of individual inserts as quickly as possible, using standard settings for each database. The primary goal of this was to prep the database for the read and update tests, and we used the relatively standard setting for each database in the load. The in-memory asynchronous numbers show that both Aersopike and Couchbase performed extremely well, all over 250 thousand inserts per second. Couchbase had a 10-15% advantage in that scenario. In the SSD scenario, neither

Figure 1: Insert Throughput



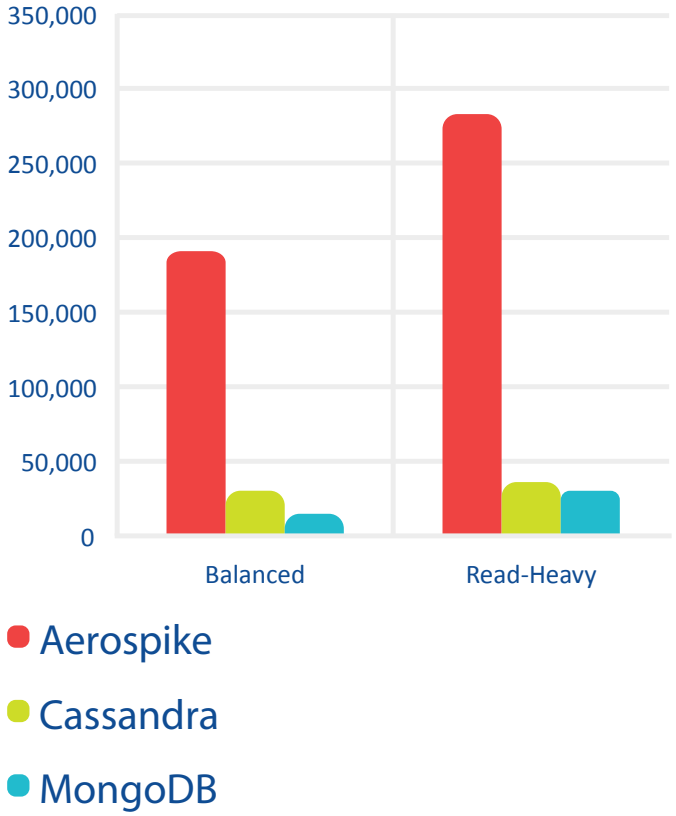
*For Couchbase 2.0, SSD throughput numbers are based on smaller sample size and asynchronous. Couchbase 1.8 was unable to load even the reduced data set.

Couchbase 1.8 nor 2.0 was able to load the data synchronously, so we reverted to asynchronous operation to load the data for Couchbase 2.0. In that scenario, the load time was still excellent, though not easily comparable to Aerospike given the configuration differences. In both scenarios, both MongoDB and Cassandra lagged far behind. Once the databases were loaded, we proceeded to the "meat" of our tests by measuring both the maximum throughput each database can achieve, and the average latency of each database at a given traffic level.

Test 2: Durable Scenario

We elected to start our throughput tests with a strong durability model, using a dataset that, when replicated, would be significantly larger than the server's RAM. This test is intended to model usage for transactional data that requires strong durability guarantees. The goal was to provide the highest throughput given this hard requirement. We expected Aerospike to dominate this category since it is specifically optimized for SSDs, and expected Cassandra to do quite well for writes since it is a write-optimized and durable database. We expected Couchbase and MongoDB to struggle since theses are both designed to keep the working set in memory. We ran the tests by using YCSB to perform each of the workloads as quickly as possible. We experimented with the proper number of client machines needed to maximize database load, and found that for most of the databases the ideal number of clients was eight, though Cassandra and MongoDB showed only minor performance gains after four clients. Before measuring, we performed an approximately 10 minute warm-up period (reading random records) to bring the database into a state that is not "at rest" and ensure any caches were properly primed. We then ran both the balanced Read/Write Workload at full capacity for 10 minutes (or until 10 million operations had been performed, whichever is slower) followed by the Read Heavy Workload with the same parameters.

Figure 2: Maximum Throughput - SSD-backed Data Set



For Aerospike, Cassandra, and MongoDB, replication was done synchronously, providing the strongest possible durability[2]. We were forced to exclude Couchbase from the official results for this test, since when run with either disk or replica durability on it was unable to complete the test[3].

In short, Aerospike was the dominant performer in this test, showing durable, replicated behavior 5-10 times faster than what the others could achieve. Even when others were set to asynchronous replication, Aerospike retained this huge advantage.

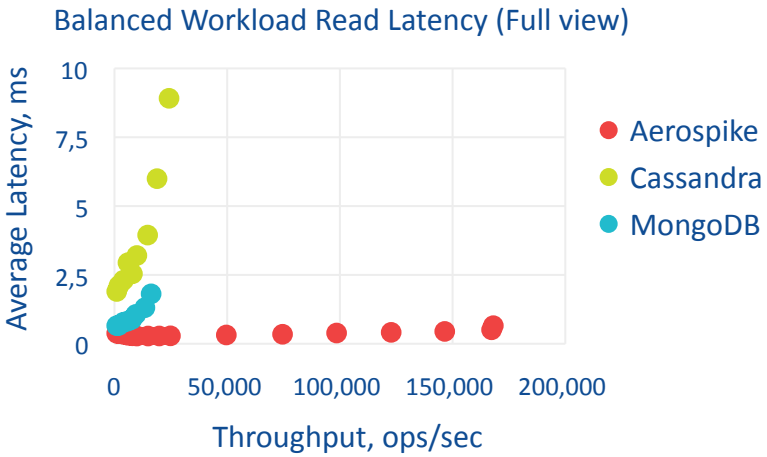
We then measured latency for various traffic levels for each database. We tracked read and update latencies separately for each workload. The graphs below show

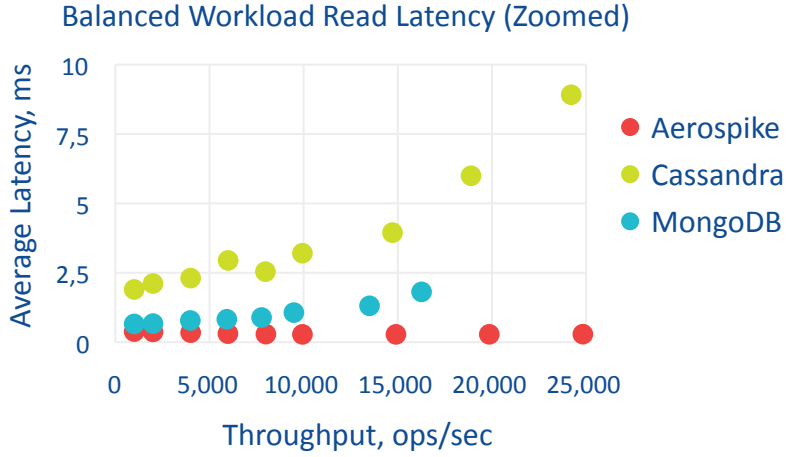
2. In the case of Cassandra, we wrote both copies of the data, and read one, which matched the durability guarantees we wanted. A similar test of writing one copy and reading both had worse performance. Abandoning the durability/consistency guarantee did not significantly affect the Read-Heavy results, but improved performance on the Balanced Workload to about 44,000 operations per second. For MongoDB, replication was synchronous but we left journaling as asynchronous.

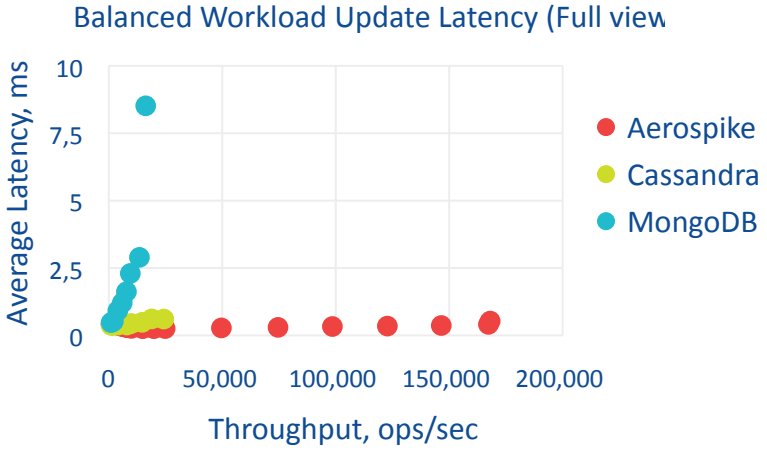
3. When run in asynchronous mode (with the 200 million row dataset), Couchbase showed performance of about 41,000 disk operations per second, suggesting an upper bound in the synchronous case.

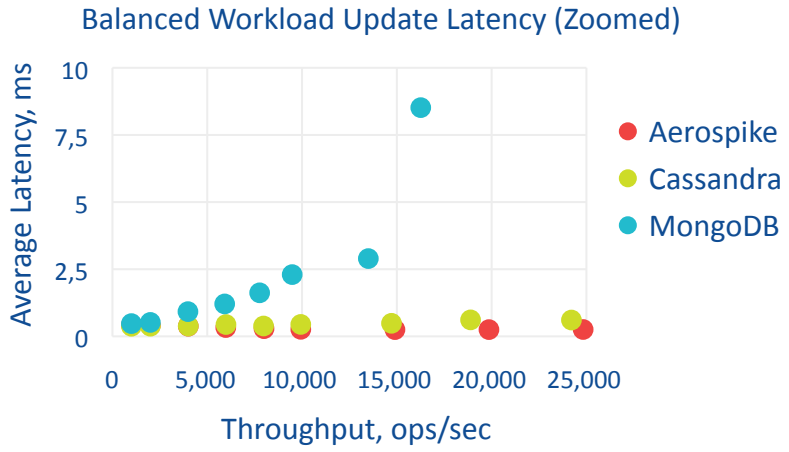
both the full set of results, plus a zoomed version to provide more clarity into cluster of data points at lower throughputs (lower is better).

Figures 3 a-d: Latency/Throughput Results - Balanced Workload

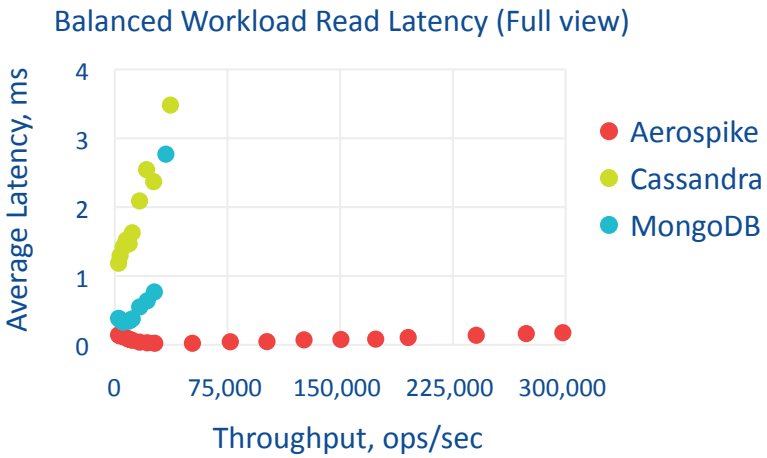


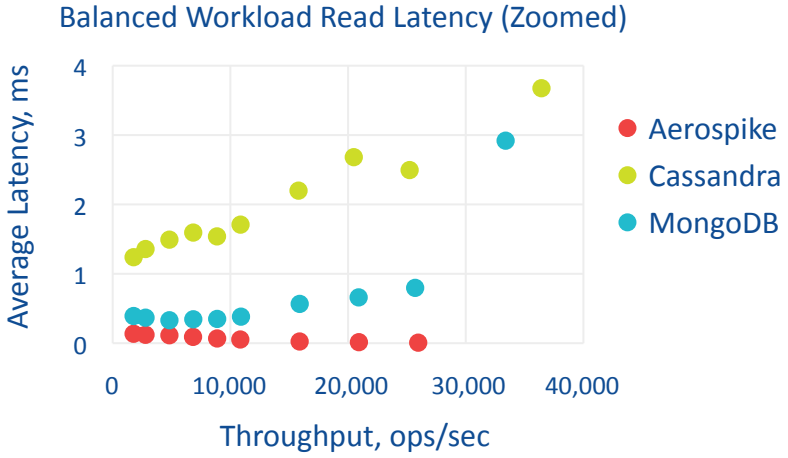


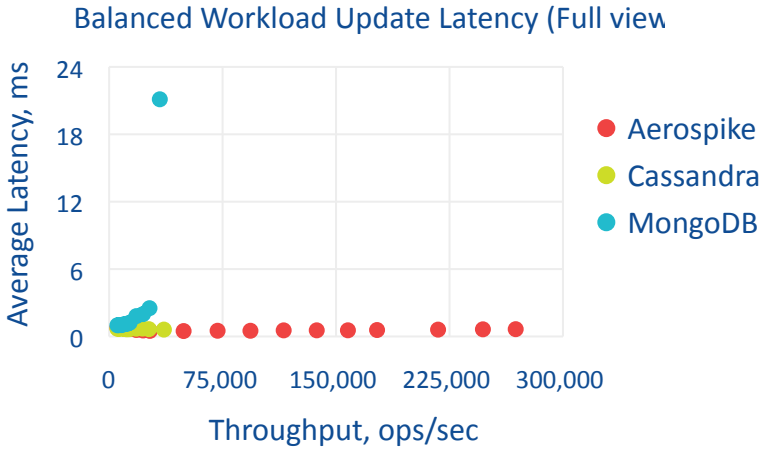


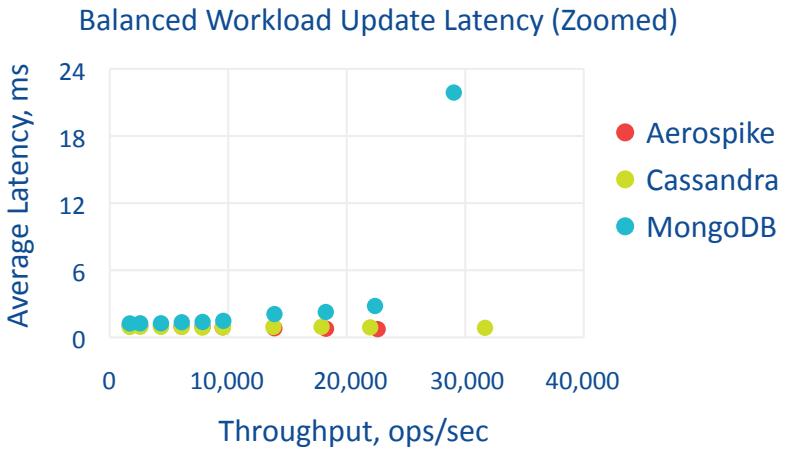


Figures 4a-d: Latency/Throughput Results - Read-Heavy Workload









Aerospike maintained sub-millisecond latencies up to its highest load in all cases. MongoDB also had very good read performance that trails off as it approaches maximum capacity, and Cassandra had consistent write latency while read latency increases linearly.

Both showed a significant drop in performance as they get saturated.

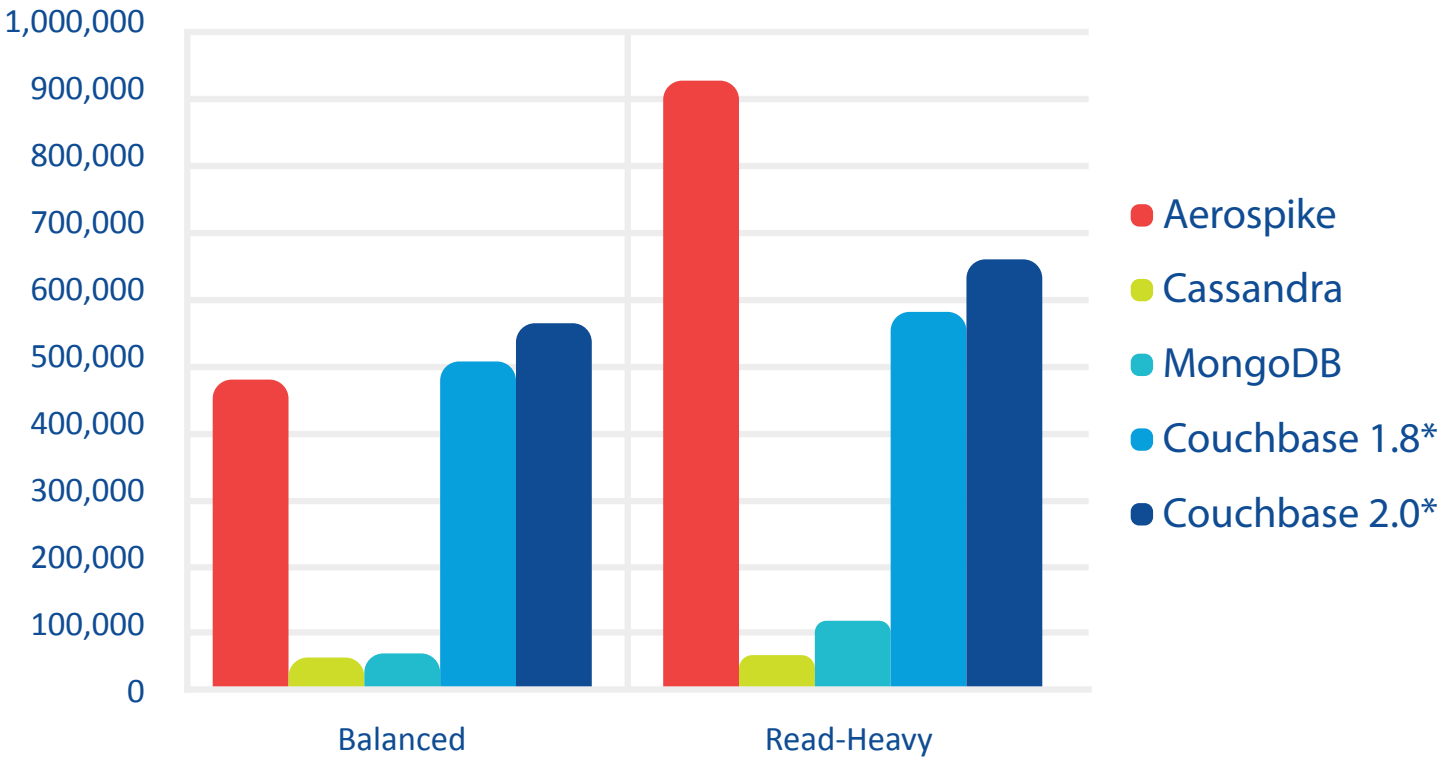
Again Couchbase numbers are excluded, but for smaller disk-bound data sets we witnessed linearly increasing latency for the balanced workload, and sub-millisecond performance for reads.

Test 3: Fast Scenario

After getting the durable numbers, we cleared the databases and ran the same tests again, this time with a dataset that was able to fit into RAM and with asynchronous replication. The goal of this test was to show the maximal performance that these databases could achieve, when liberated from the requirement of having durable data. All the databases in this test were configured to store the entire working set in memory and persist to disk and replicas as soon as it became available. We expected this to be a test where Couchbase would shine, since its default setup is designed to accept requests as quickly as memcached, and persist them asynchronously.

The client setups were identical to those in Test 2.

Figure 5: Maximum Throughput - In Memory Data Set

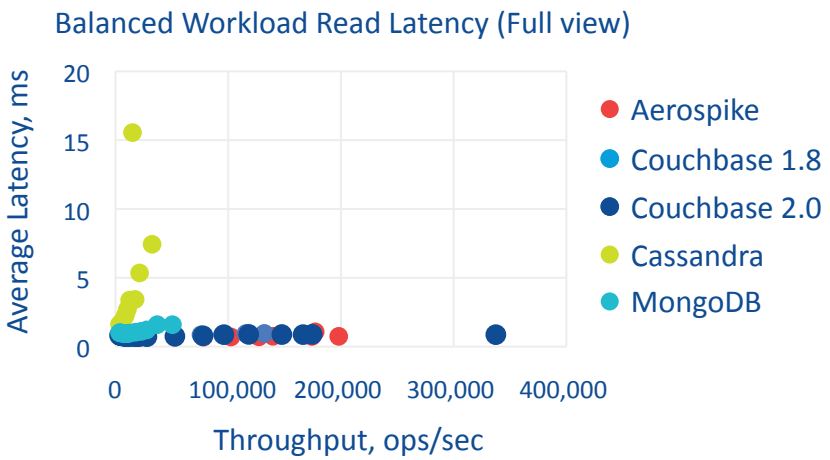


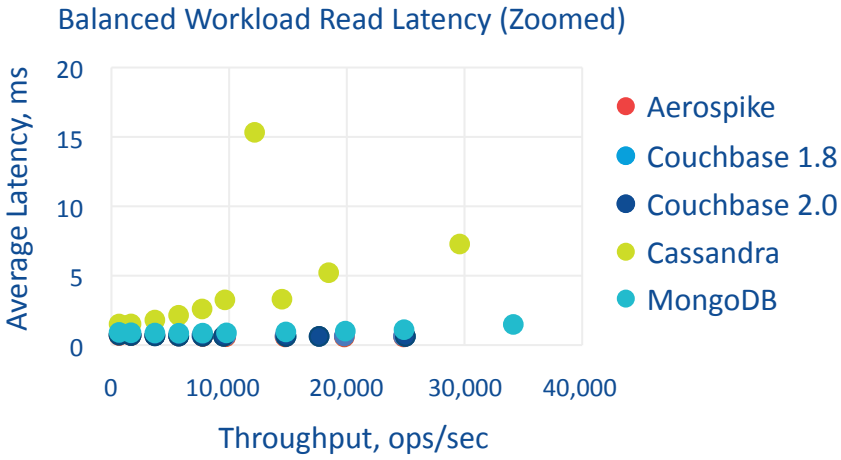
In this scenario, Couchbase was the fastest performer for the balanced workload by about 10%, whereas Aerospike outperformed Couchbase by about 40% on the read-heavy workload. Both of these systems demonstrated truly impressive performance, on the order of half a million to a million operations per second across the board.

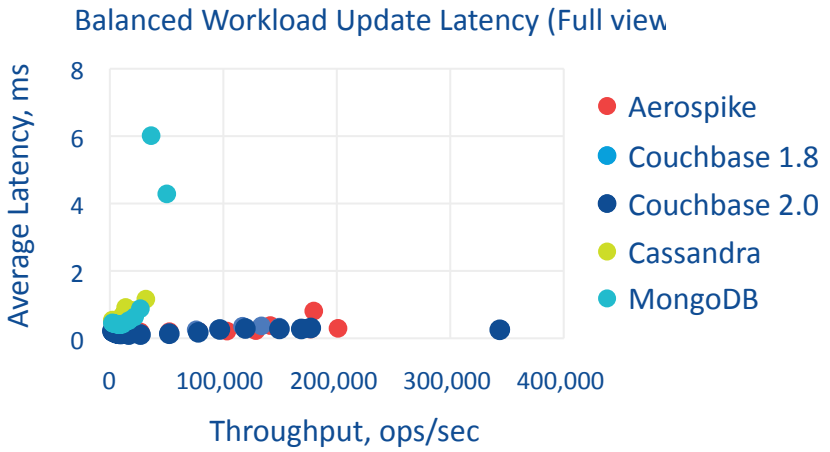
MongoDB and Cassandra were an order of magnitude slower - both were configured with enough RAM cache to contain the full working set. Cassandra's cache hit ratio was about 35% for the balanced workload and 70% for the read-heavy workload.

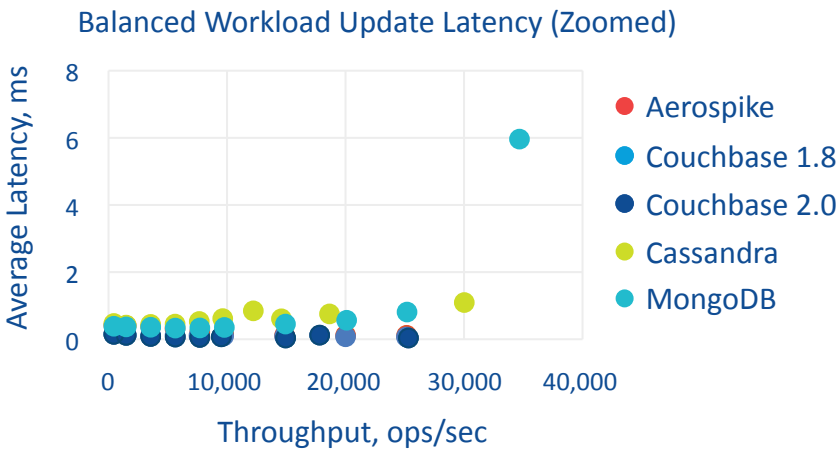
The latency tests were conducted in the same manner as before.

Figures 6a - 6d: Latency/Throughput Results (Balanced Workload)

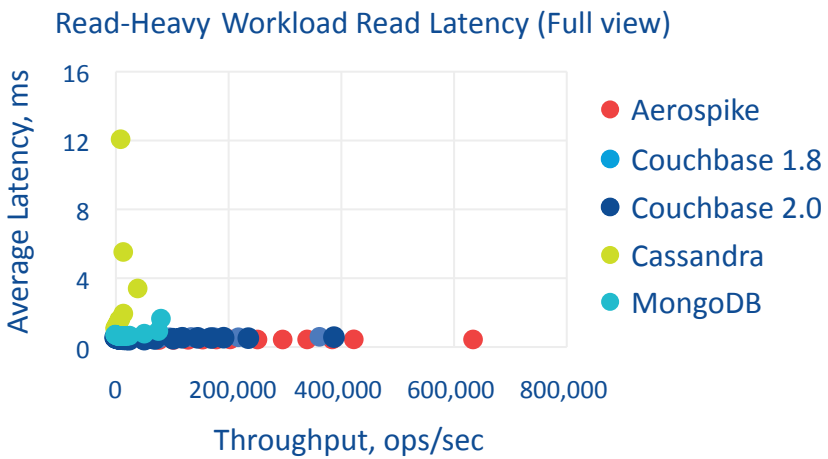


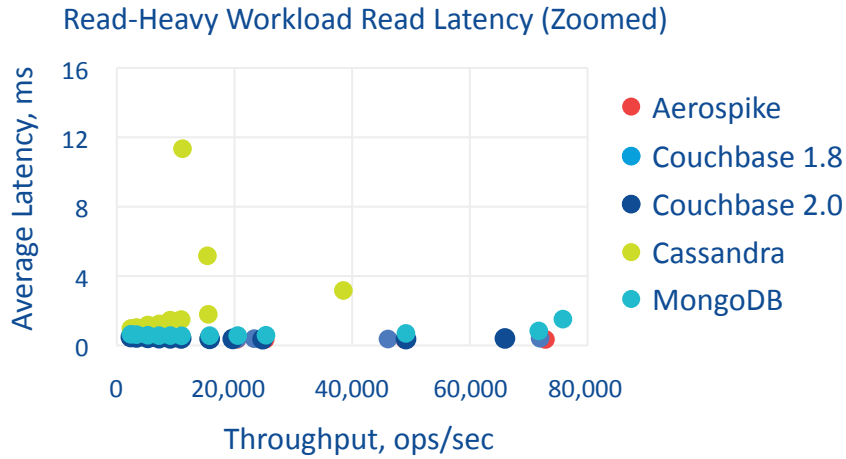


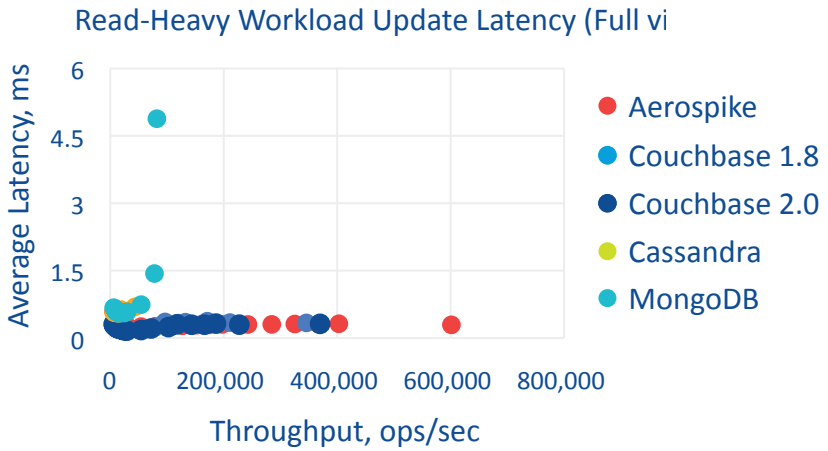


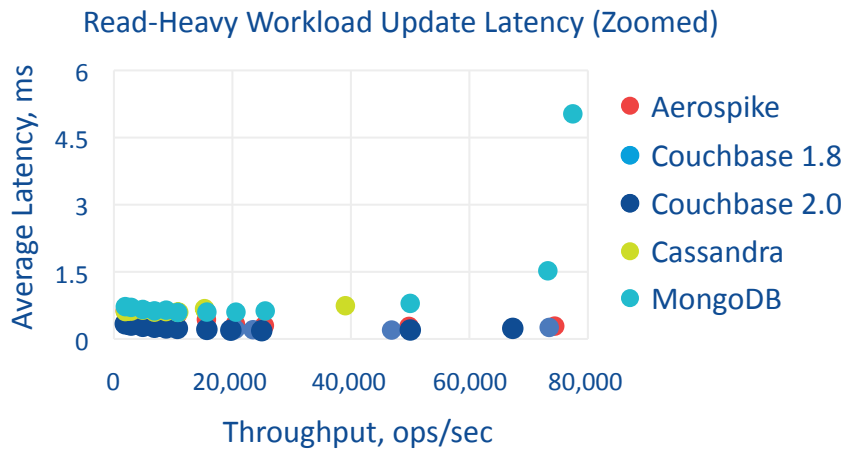


Figures 7a - 7d: Latency Throughput Results (Read Heavy Workload)









Both Couchbase and Aerospike maintained sub-millisecond latencies up to their highest performance. For MongoDB, latency degraded on writes but stayed consistent on reads, and the converse was true for Cassandra (which is optimized for writes).

Conclusion

The most striking result was the raw throughput number Aerospike was able to achieve even while committing to disk across multiple nodes. We expected it to shine in this case, but maintaining a speed of 200 thousand operations per second with these strong guarantees put it far ahead of its nearest competitor and at speeds we would not normally associate with ACID semantics.

When the entire data set fit into RAM and the durability guarantees are weakened, the results showed both Couchbase and Aerospike in a near-tie in terms of performance. Couchbase slightly outperformed Aerospike for the balanced read-write workload, and Aerospike somewhat more significantly outperformed Couchbase for the read-heavy workload. Both posted excellent numbers when durability is not a major concern. Of course, a lack of durability opens the question of how the cluster recovers in the event of a failure. We plan to answer this in our next report.

Both Cassandra and MongoDB lagged far behind the others, but it should be pointed out that both offer a significantly larger feature set than Aerospike and Couchbase 1.8. This test measured raw key-value performance, and the key-value stores were the ones that shined. Quantifying secondary indexes and other features will be done in a future report.

One thing to consider even in the asynchronous model is how an organization might scale out their data storage. We viewed the SSD based test as important since SSDs have higher densities and lower per-gigabyte costs than RAM. As such, it should be possible to scale our much larger data sets on fewer nodes, which is clearly valuable when talking about very large amounts of data. An alternative scaling model would be provisioning larger numbers of (possibly virtualized) RAM-based machines and ensuring that the working set will always fit into

memory. When talking about data sets of 6GB or 60GB as we do in this report, this is clearly a viable scaling option, and one that can offer higher throughput. At truly large data sets, this scaling approach becomes more questionable.

When scaling while using more instances of RAM-backed storage, one should consider the recovery aspect of the solution. More nodes implies a higher rate of node failures, so recovery becomes more important. If recovery can be managed effectively, a RAM-based cloud-oriented approach to storage might be a viable way to scale. However, if writes to disks and replicas are done asynchronously, recoverability can be problematic. A future report will compare database recoverability and virtualized performance.

Many other considerations besides raw performance have been swept under the rug in this study. For example, all the databases except Couchbase 1.8 offer cross datacenter replication, which is likely a consideration when deciding between consistency, response time, and durability

As with any benchmarking tests, the results should be taken with the usual caveats. Running application loads in bulk against a database is merely a proxy for how they will be used in real life.

Is this a fair test?

It can certainly be argued the test was set up to achieve certain results. Since Aerospike was specifically written to be explicitly optimized for SSDs, it is not surprising that a test against raw SSD hardware would give it the best results. However, the need for a high-performance key value store is a real one, and the assumption that companies looking for maximal performance in such scenarios would invest in servers with solid-state storage seems sound. For applications in the space we are considering, we believe running the tests on such hardware is reasonable, and the test is a rough approximation of similar applications we've built. It is also important to point out that MongoDB, Cassandra, and Couchbase 2.0 have much broader functionality than what we test. In particular, they have secondary indices that allow querying of data by field instead of simply by key, and allow the databases to be used in many kinds of use cases that are

currently not viable for Aerospike. A raw benchmark test here is not the best way to illustrate their functionality. Each of these databases has clear use cases that are not illustrated at all by a raw key-value performance test. In particular, 10gen explicitly warns against benchmark tests as a way to describe MongoDB, and we have used MongoDB in production with very good results for many projects that are more document-oriented and less geared towards maximal transaction volumes.

Future tests

This benchmark was the first of several tests we plan to run to more accurately categorize the NoSQL landscape. We started with raw key-value performance because it was relatively easy to measure and is a very real use case that we regularly encounter. However, there are several other dimensions we plan to analyze in the upcoming weeks and months, including:

1. Recovery - These databases vary considerably in how well they tolerate and recover from node failures. We are in the process of quantifying this for future publication.

2 Burstability - How the databases perform when data starts to exceed capacity planning. For example, a RAM-based cluster might perform extremely well, but when the working set grows too large suddenly, how will performance be impacted? Different databases have different approaches to handling this (evicting records, slowing performance, etc.)
and quantifying this would shed light on expected system stability.

3. Cloud support - A major reason to choose a NoSQL solution is to offer horizontal scalability, and the cloud is a natural choice for this. We elected to start with bare metal hardware because such hardware can be expected to deliver the highest raw
performance. However, a test using High IO cloud instances would be a valuable follow up.

4. Secondary indexes - Such a test can be quite complex, as queries across secondary indexes will hit every node in the cluster and may result in complex query execution
paths. However, a large number of applications need more than key-value storage, and a way to quantify secondary index capabilities and performance can

provide insight into entirely new classes of applications.

5. Other workload distributions - YCSB comes with a number of data set distributions, and we weren't particularly happy with any of them. The Zipfian distribution seems to be very heavily weighted to a few keys, but the random distribution doesn't seem like it would model real-world usage. We preferred to conduct in-depth tests around a single distribution instead of adding more and more variables, potentially confusing the results. In the future, we plan to measure the impact of data distributions more carefully

Appendix A: Hardware

Database Servers

We ran the tests on four server machines. Each machine had the following specs:

CPU: 8 x Intel(R) Xeon(R) CPUE5-2665 0 @ 2.40GHz

RAM: 31 GB

SSD: 4 x INTEL SSDSA2CW120G3, 120 GB full capacity, 94 GB over-provisioned size

HDD: ST500NM0011, 500 GB, SATA III, 7200 RPM

Each server had all five databases installed, but only one running at any given time. Each database used four SSDs simultaneously to store its data. Aerospike accessed the disk directly; the others accessed the data via software RAID0. For all databases, the data were distributed across all four SSDs. Other disks were used to some small capacity on individual tests, but only in ancillary roles. For example, we tested three ways of storing Cassandra's commit log, as described in Appendix C. The SSDs for Cassandra, both Couchbase versions and MongoDB were formatted as ext4. The file systems are mounted with the "noatime" option (the inode access times are not updated). While the RAID chunk size was left at the default (521k), the "readahead" parameter was changed to 32 sectors for the whole RAID device and to 8 sectors for each SSD. Aerospike was tested in two different configurations: by using the disk as a block device, and by storing the data in RAM. All the disks were overprovisioned, leaving 21% of disk space empty. This is consistent with recommendations made by 10gen and Aerospike (the other vendors had no documentation around this). Overprovisioning can be done simply by leaving 21% of the disk unpartitioned. In our case, we used the hdparm command as described in Aerospike guide.

Disks were initialized by wiping out the data directory (for Cassandra, MongoDB, Couchbase) or by zeroing out the block from /dev/zero (for Aerospike). The system I/O scheduler was set to be NOOP.

The rotational drive was used to store logs and binaries. Network was 1Gbps Ethernet. Higher network bandwidth would be particularly valuable in the case of Couchbase.

Client Machines

We used up to ten client machines to generate load to the database with YCSB. Each had the following specs:

CPU: 4 x Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz
RAM: 3.7 GB
HDD: ST500DM002-1BD142, 500 GB, SATA III, 7200 RPM

Appendix B: Installed Software

Database Servers

Each of the database server machines had the following software configured:

1. OS: Ubuntu Server 12.04.1 64-bit (Linux kernel v.3.2.0)
2. JDK: Oracle JDK 7u9
3. Databases:
- Aerospike 2.1.2-100-g35e99a9
- Couchbase 1.8.1
- Couchbase 2.0.0
- Cassandra 1.1.7
- MongoDB 2.2.2

Client Machines

1. OS: Ubuntu Server 12.04.1 64-bit (Linux kernel v.3.2.0)
2. JDK: OpenJDK 6.0u24
3. Customized YCSB forked from the commit on September 10, 2012 (Aerospike guide)
- Couchbase plugin downloaded from Aerospike guide.
- Aerospike plugin provided by Aerospike
- Cassandra plugin from YCSB distribution
- MongoDB plugin from YCSB distribution, but with the following customizations:
- Upgraded to use MongoDB 2.10.1 drivers
- Additional code to enable routing read requests to secondary nodes
- See Appendix E for information on the changes we made as well as the location of the source code.

Appendix C: Database Configuration

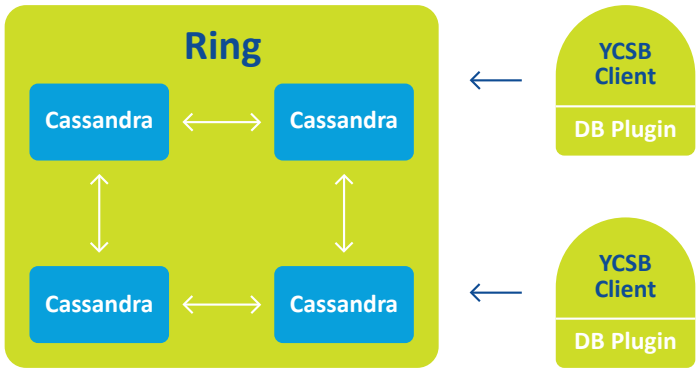
Other settings

1. ntpd was running on all machines.
2. Swap space was set to the size of main memory and left on the rotational disk
3. The maximum number of open files for all users was increased from 1024 to 16384 (ulimit nofiles)
4. Network cards with multiqueue support were installed: Intel Corporation I350 Gigabit Network Connection, and the igb network driver was used

Cassandra

Cassandra 1.1.7 was installed as a four-node cluster. We ran the tests using 3 different disc configurations. In the first variant, three dedicated SSD disks on each machine were formatted as ext4 and used to store the data. Commit logs were written to a separate SSD disk for performance reasons. (Since Cassandra is optimized for writes, the first thing it will do is append writes to a commit log, persisting data to the main disk in batches. We placed these on separate disks to reduce possible contention between these operations.). In the second variant, four SSD disks were used for data and one HDD disk was used for commit logs. In the third variant, four SSD disks were used both for data and for the commit log. The third configuration gave the best results.

Unlike the other databases tested, Cassandra uses a ring topology in its configuration and the nodes need to be made aware of “seed” nodes (who help them join the ring) explicitly. At configuration time, it was necessary to specify which tokens map to which instance.



We used the token generation tool available at https://raw.github.com/riptano/ComboAMI/2.2/tokentoolv2.py to create the node configuration:

```
$ ./tokentoolv2.py 4
{
    "0": {
        "0": 0,
        "1": 42535295865117307932921825928971026432,
        "2": 85070591730234615865843651857942052864,
        "3": 127605887595351923798765477786913079296
    }
}
```

Cassandra has tunable consistency levels. Each read or write can explicitly state what level of database consistency is needed for that operation. Since this was a benchmark project, we used the weakest and fastest consistency level (ONE) for both reads and writes[4].

The JVM used to run Cassandra was initialized with the following settings (from conf/cassandra-env.sh):

```
MAX_HEAP_SIZE="15G"
HEAP_NEWSIZE="800M"
```

By default, the row cache for Cassandra is disabled; we enabled it and set it to its max size: 10GB per node. The cache is not in the Java heap space. Enabling the cache dramatically increased the read throughput – but did not have a significant effect on latency. Even on inmemory tests, the cache hits did not exceed 70%.

---

4. For more information on consistency levels, see http://www.datastax.com/docs/1.1/dml/data_consistency#tunable-consistency

As with all the databases, we used a replication factor of two. Other major settings used were:

Partitioner: RandomPartitioner
Initial token space: 2^127 / 4
Memtable space: 4Gb
Concurrent reads: 64
Concurrent writes: 64
Compression: SnappyCompressor
Commit log sync: 10,000 ms

The disc configurations for each variant follow:

Variant 1

3 SSD for data + 1 SSD for commitlog

```
mdadm --create --verbose /dev/md0 --level=0 --raid-devices=3 /dev/sdb /dev/sdc /dev/sdd
mkfs.ext4 /dev/md0
mkdir /mnt/raid
mount /dev/md0 /mnt/raid
mkfs.ext4 /dev/sde
mkdir /mnt/log
mount /dev/sde /mnt/log
```

Inserts: 72k ops/sec
Heavy Update: 46k ops/sec
Mostly Read: 34k ops/sec

Variant 2

4 SSDs for data + 1 HDD for commitlog

```
mdadm --create --verbose /dev/md0 --level=0 --raid-devices=4 /dev/sdb /dev/sdc /dev/sdd/dev/sde
mkfs.ext4 /dev/md0
mkdir /mnt/raid
mount /dev/md0 /mnt/raid
mkdir /mnt/log
```

Inserts: 69k ops/sec
Heavy Update: 46k ops/sec
Mostly Read: 30k ops/sec

Variant 3

4 SSDs for data and commitlog

Inserts: 78k ops/sec

Heavy Update: 45k ops/sec

Mostly Read: 32k ops/sec

And below are the settings for **conf/cassandra.yaml:**

```
cluster_name: 'Test Cluster'
initial_token: 0
hinted_handoff_enabled: true
max_hint_window_in_ms: 3600000 # one hour
hinted_handoff_throttle_delay_in_ms: 1
authenticator: org.apache.cassandra.auth.AllowAllAuthenticator
authority: org.apache.cassandra.auth.AllowAllAuthority
partitioner: org.apache.cassandra.dht.RandomPartitioner
data_file_directories:
- /mnt/raid/cassandra/data
commitlog_directory: /mnt/raid/cassandra/commitlog
key_cache_size_in_mb:
key_cache_save_period: 14400
row_cache_size_in_mb: 10240
row_cache_save_period: 0
row_cache_provider: SerializingCacheProvider
saved_caches_directory: /var/lib/cassandra/saved_caches
commitlog_sync: periodic
commitlog_sync_period_in_ms: 10000
commitlog_segment_size_in_mb: 32
seed_provider:
    - class_name: org.apache.cassandra.loca
    tor.SimpleSeedProvider
    parameters:
        - seeds: "e1.citrusleaf.local"
flush_largest_memtables_at: 0.75
reduce_cache_sizes_at: 0.85
reduce_cache_capacity_to: 0.6
concurrent_reads: 64
concurrent_writes: 64
memtable_flush_queue_size: 4
trickle_fsync: false
trickle_fsync_interval_in_kb: 10240
storage_port: 7000
```

```
  ssl_storage_port: 7001
  listen_address: e1.citrusleaf.local
  rpc_address: e1.citrusleaf.local
  rpc_port: 9160
  rpc_keepalive: true
  rpc_server_type: sync
  thrift_framed_transport_size_in_mb: 15
  thrift_max_message_length_in_mb: 16
  incremental_backups: false
  snapshot_before_compaction: false
  auto_snapshot: true
  column_index_size_in_kb: 64
  in_memory_compaction_limit_in_mb: 64
  multithreaded_compaction: false
```

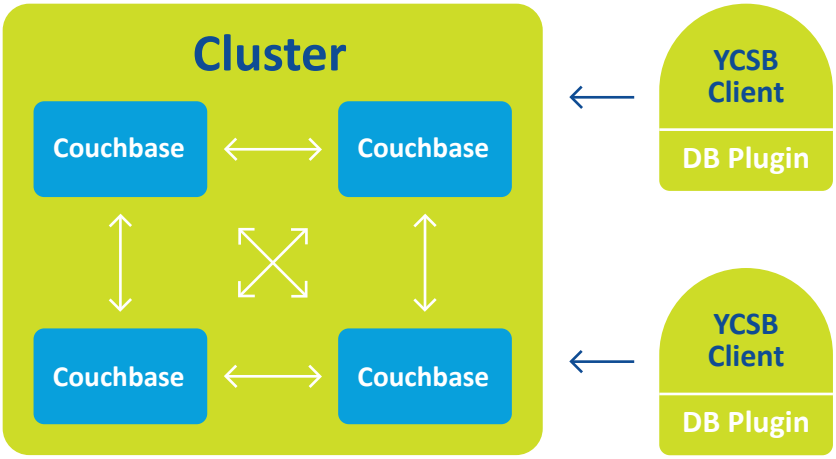
The database was initialized using the following commands:

```
  CREATE KEYSPACE usertable
  WITH placement_strategy = 'org.apache.cas
sandra.locator.SimpleStrategy'
  AND strategy_options = {replication_factor:2};
  use usertable;
  CREATE COLUMN FAMILY data
  WITH comparator = UTF8Type
  AND key_validation_class = UTF8Type
  AND caching = all;
```

Couchbase 1.8

Couchbase 1.8.1 was installed on four server nodes are configured as a cluster. Four SSDs accessible via software RAID0 formatted as ext4 were used to store the data.

One thing to note is that Couchbase defines its replication factor as “Number of replica (backup) copies)”, so a setting of 1 corresponds to our replication factor of 2 for other databases.



Below are the settings for couchbase-cli server-info:


```
{
  "availableStorage": {
    "hdd": [
        {
          "path": "/",
          "sizeKBytes": 447427440,
          "usagePercent": 6
        },
        {
          "path": "/dev",
          "sizeKBytes": 16451884,
          "usagePercent": 1
        },

        {
          "path": "/run",
          "sizeKBytes": 6584384,
          "usagePercent": 1
        },
        {
          "path": "/run/lock",
          "sizeKBytes": 5120,
          "usagePercent": 0
        },
        {
          "path": "/run/shm",
          "sizeKBytes": 16460960,
          "usagePercent": 0
        },
```

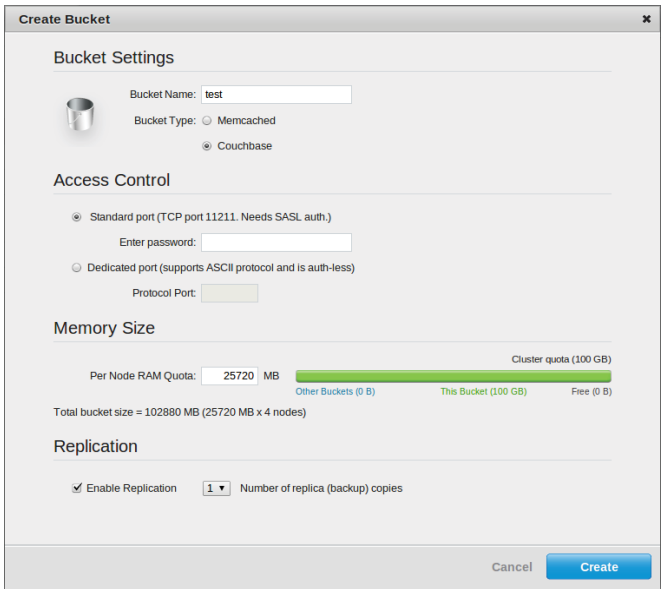
```
      {
        "path": "/boot",
        "sizeKBytes": 233191,
        "usagePercent": 33
      },
      {
        "path": "/mnt/raid",
        "sizeKBytes": 364603720,
        "usagePercent": 1
      }
    ]
},
"clusterCompatibility": 1,
"clusterMembership": "active",
"hostname": "192.168.109.168:8091",
"interestingStats": {
  "curr_items": 0,
  "curr_items_tot": 0,
  "vb_replica_curr_items": 0
},
"mcdMemoryAllocated": 25720,
"mcdMemoryReserved": 25720,
"memoryFree": 22558875648.0,
"memoryQuota": 25720,
"memoryTotal": 33712046080.0,
"os": "x86_64
-unknown
-linux
-gnu",
"otpCookie": "opztzvwywxeczcji",
"otpNode": "ns_1@192.168.109.168",
"ports": {
  "direct": 11210,
  "proxy": 11211
},
"status": "healthy",
"storage": {
  "hdd": [
    {
      "path": "/mnt/raid/couchbase",
      "quotaMb": "none",
      "state": "ok"
```

```
        }
      ],
      "ssd": []
    },
    "storageTotals": {
      "hdd": {
          "free": 369620667188.0,
          "quotaTotal": 373354209280.0,
          "total": 373354209280.0,
          "used": 3733542092.0,
          "usedByData": 7281024
    },
    "ram": {
      "quotaTotal": 26969374720.0,
      "total": 33712046080.0,
      "used": 11153170432.0,
      "usedByData": 55127464
    }
    },
   "systemStats": {
      "cpu_utilization_rate": 0.4987531172069 8257,
      "swap_total": 34326179840.0,
      "swap_used": 6639616
    },
    "uptime": "1782",
    "version": "1.8.1-937-rel-community"
}
```


The couchbase-cli bucket-list was left at the default settings:


```
    test
        bucketType: membase
        authType: sasl
        saslPassword:
        numReplicas: 1
        ramQuota: 1.0762584064e+11
        ramUsed: 53101028392.0
```


Here is the configuration used to set up the db schema.



Couchbase 1.8

Couchbase 2.0 was officially released
on December 12, 2012. As with Couchbase 1.8, it was installed on four server nodes are configured as a cluster, and four SSDs accessible via software RAID0 formatted as ext4 were used to store the data.

Below are the settings for couchbase-cli server-info:

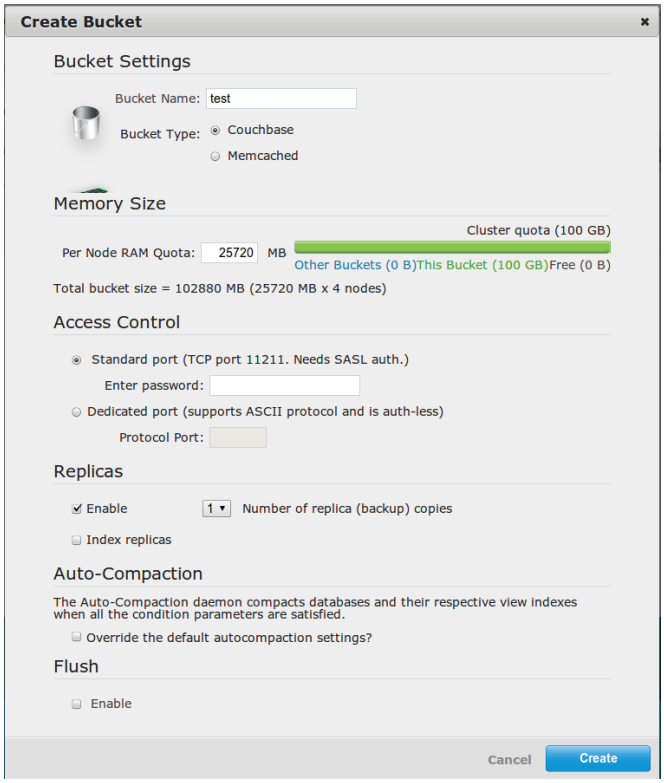

```
 {
   "availableStorage": {
     "hdd": [
       {
         "path": "/",
         "sizeKBytes": 447427440,
         "usagePercent": 9
       },
       {
         "path": "/dev",
         "sizeKBytes": 16451884,
         "usagePercent": 1
       },
       {
         "path": "/run",
         "sizeKBytes": 6584384,
         "usagePercent": 1
       },
       {
```

```
        "path": "/run/lock",
        "sizeKBytes": 5120,
        "usagePercent": 0
      },
      {
        "path": "/run/shm",
        "sizeKBytes": 16460960,
        "usagePercent": 0
      },
      {
        "path": "/boot",
        "sizeKBytes": 233191,
        "usagePercent": 33
      },
      {
        "path": "/mnt/raid",
        "sizeKBytes": 364603720,
        "usagePercent": 52
      }
    ]
},
"clusterCompatibility": 131072,
"clusterMembership": "active",
"couchApiBase":
"http://192.168.109.168:8092/",
"hostname": "192.168.109.168:8091",
"interestingStats": {
    "couch_docs_actual_disk_size": 9205562348,
    "couch_docs_data_size": 6887104963,
    "couch_views_actual_disk_size": 0,
    "couch_views_data_size": 0,
    "curr_items": 12506364,
    "curr_items_tot": 25003344,
    "mem_used": 7481749880,
    "vb_replica_curr_items": 12496980
},
"mcdMemoryAllocated": 25720,
"mcdMemoryReserved": 25720,
"memoryFree": 566579200,
"memoryQuota": 25720,
"memoryTotal": 33712046080,
"os": "x86_64-unknown-linux-gnu",
```

```
"otpCookie": "qdfyjoucywviqpah",
"otpNode": "ns_1@192.168.109.168",
"ports": {
  "direct": 11210,
  "proxy": 11211
},
"status": "healthy",
"storage": {
  "hdd": [
    {
      "index_path": "/mnt/raid/couchbase/data",
      "path": "/mnt/raid/couchbase/data",
      "quotaMb": "none",
      "state": "ok"
    }
  ],
  "ssd": []
},
"storageTotals": {
  "hdd": {
    "free": 179210020455,
    "quotaTotal": 373354209280,
    "total": 373354209280,
    "used": 194144188825,
    "usedByData": 9205562348
  },
  "ram": {
    "quotaTotal": 26969374720,
    "quotaUsed": 26969374720,
    "total": 33712046080,
    "used": 33145466880,
    "usedByData": 7481749880
  }
},
"systemStats": {
  "cpu_utilization_rate": 0.625782227784 7309,
  "swap_total": 34326179840,
  "swap_used": 15790080
},
"thisNode": true,
"uptime": "15975",
"version": "2.0.0-1976-rel-enterprise"
```

```
}
```

Here is the configuration used to set up the db schema.



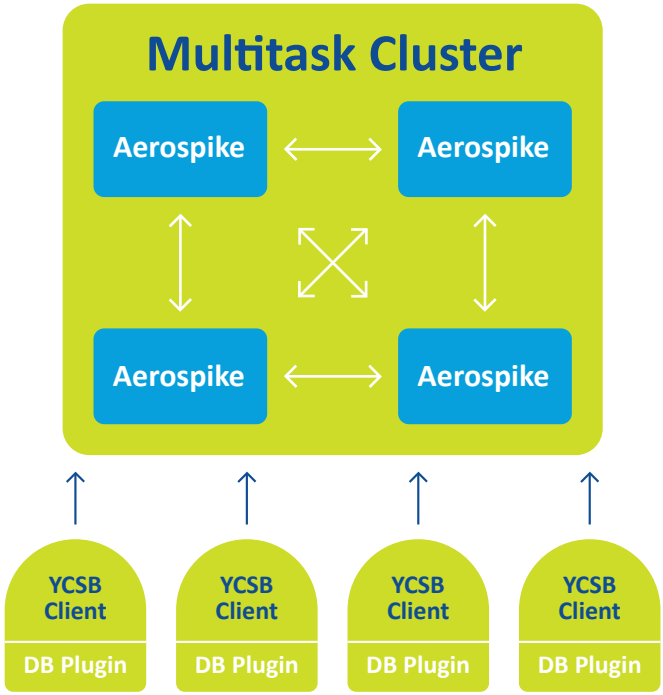
Aerospike

Aerospike 2.1.2-100-g35e99a9 was used in a four node cluster. The free trial is a fully functional but time-limited database.

Unlike the other three databases, Aerospike uses the SSD natively as a block device when storing the data on discs. As with the other databases, we used four SSDs accessible via software RAID0 formatted as ext4, only instead of formatting it as ext4 we simply initialized the drives to all zeros:

```
dd if=/dev/zero of=/dev/sdb bs=128k
```

The cluster heartbeat was configured to use multicast, not mesh.



For the SSD tests, the **/etc/citrus**

leaf/citrusleaf.conf we used is as follows:

```
service {
        user root
        group root
        run-as-daemon
        transaction-queues 8
        transaction-threads-per-queue 3
        service-threads 8
        fabric-workers 24
        migrate-threads 1
        migrate-xmit-hwm 6
        migrate-xmit-lwm 1
        transaction-retry-ms 1000
        transaction-max-ms 1000
        transaction-pending-limit 200 # Max # of same-key transactions on queue
        ticker-interval 10
        nsup-period 120
        nsup-max-deletes 25000
        nsup-queue-hwm 2
        nsup-queue-lwm 1
        nsup-startup-evict true
        defrag-queue-hwm 20
        defrag-queue-lwm 5
```

```
        defrag-queue-escape 10
        defrag-queue-priority 10
        proto-fd-max 15000
# Keep this less than 1024 so the server starts up even on low-end machines.
        paxos-single-replica-limit 1
 # Number of nodes where the replica count is
automatically reduced to 1.
        transaction-repeatable-read false
        pidfile /var/run/cld.pid
        trial-account-key P3NqitOnyXBfCb0Xd3vqmPwXj2M60TnanXtre3OEY3g
}
# Log configuration. Log to stderr by default. Log file must be an absolute path.
  logging {
        file /var/log/citrusleaf.log {
            context any info
  #         context batch debug
  #         context rw detail
        }

  #     console {
  #       context any info
  #     }
  #
  }

  network {
        service {
          address any
          port 3000
          reuse-address
        }

        heartbeat {
          address 239.1.99.223
          mode multicast
          port 9918
          interval 150
          timeout 15
        }

        fabric {
          address any
```

```
            port 3001
        }
        info {
            address any
            port 3003
        }
    }
    #namespace test {
    #        replication-factor 2
    #        storage-engine memory
    #}

    namespace test {
            replication-factor 2
            high-water-memory-pct 60
            high-water-disk-pct 50
            stop-writes-pct 90
            memory-size 32212254720
# 30G
            default-ttl 2592000
# default 30 days expiration

# Warning - legacy data in defined raw partition devices will be erased.
# These partitions must not be mounted by the filesystem.
    storage-engine device {
            scheduler-mode noop
# for SSD
            device /dev/sdb
# uncomment this line when correct device is used.
            device /dev/sdc
            device /dev/sdd
            device /dev/sde
            load-at-startup true
            write-block-size 131072
            defrag-period 120
            defrag-lwm-pct 50
            defrag-max-blocks 4000
            defrag-startup-minimum 10
    }
}
```

These are mostly default values. The most important thing to consider here is the

proper settings for high-water-disk-pct. Like Cassandra, Aerospike optimizes writes by streaming them sequentially. The number of records was modified to fit into the watermark and avoid object evictions. The index fits into memory.

Also, we tested Aerospike storing the data in RAM. For the RAM tests, both the data and index were modified to fit into memory. We used the following configuration file:

```
  service {
            user root
            group root
            run-as-daemon
            transaction-queues 8
            transaction-threads-per-queue 3
            service-threads 8
            fabric-workers 24
            migrate-threads 1
            migrate-xmit-hwm 6
            migrate-xmit-lwm 1
            transaction-retry-ms 1000
            transaction-max-ms 1000
            transaction-pending-limit 200
# Max # of same-key transactions on queue
            ticker-interval 10
            nsup-period 120
            nsup-max-deletes 25000
            nsup-queue-hwm 2
            nsup-queue-lwm 1
            nsup-startup-evict true
            defrag-queue-hwm 20
            defrag-queue-lwm 5
            defrag-queue-escape 10
            defrag-queue-priority 10
            proto-fd-max 15000
# Keep this less than 1024 so the server starts up even on low-end machines.
            paxos-single-replica-limit 1
# Number of nodes where the replica count is
automatically reduced to 1.
            transaction-repeatable-read false
            pidfile /var/run/cld.pid
            trial-account-key P3NqitOnyXBfCb0Xd3vqmPwXj2M60TnanXtre3OEY3g
}
```

```
# Log configuration. Log to stderr by default. Log file must be an absolute path.
logging {
        file /var/log/citrusleaf.log {
                context any info
#               context batch debug
#               context rw detail
        }

#       console {
#               context any info
#       }
#
}

network {
        service {
                address any
                port 3000
                reuse-address
        }

        heartbeat {
                address 239.1.99.223
                mode multicast
                port 9918
                interval 150
                timeout 15
        }

        fabric {
                address any
                port 3001
        }

        info {
                address any
                port 3003
        }
}
#namespace test {
#       replication-factor 2
#       storage-engine memory
```

```
#}
namespace test {
        replication-factor 2
        high-water-memory-pct 60
        high-water-disk-pct 50
        stop-writes-pct 70
        memory-size 32212254720
# 30G
        default-ttl 2592000
# default 30 days expiration

# Warning - legacy data in defined raw partition devices will be erased.
# These partitions must not be mounted by the filesystem.
storage-engine device {
                file /var/data/citrusleaf/test.data
# data file name on rotational disk
                filesize 137438953472
# 128G - use disk file up to 128G for
this namespace
                data-in-memory true
# keep a copy of all data in memory always
                defrag-period 120
# run defrag every 120 seconds
                defrag-lwm-pct 45
# reclaim blocks that are less than 45% full
                defrag-max-blocks 4000
# defragment at most 4000 disk blocks in each run
                defrag-startup-minimum 10
# server needs at least 10% free space at startup
        }

}
```

MongoDB

We used MongoDB 2.2.2 to perform the tests. Four SSDs accessible via software RAID0 formatted as ext4 were used to store the data.

MongoDB has a different approach to clustering than the other databases. Instead of one monolithic cluster, MongoDB uses shards of replica sets. Each replica set is responsible for a set of keys, and contains a Primary which by default

handles all read and write requests, and one or more Secondaries which are used in recovery. In our setup, we divided the four node cluster into two replica sets, each with a primary and secondary node. This is not a productionrecommended setup, but seemed suitable for a benchmarking test.

In situations where an even number of nodes exist in each replica set, a third Arbiter node is necessary to participate in elections in case a node goes down. In our setup, we placed the Arbiter on one of the client nodes, which is a poor decision in a production environment but in our controlled environment was adequate to perform our failover tests. There is also a mongos process which lived on the client machines and was responsible for routing queries to the appropriate shards (replica sets).

The resulting cluster configuration:

Server Node 1: mondod as Primary of Shard 1
Server Node 2: mongod as Secondary of Shard 1
Server Node 3: mongod as Primary of Shard 2
Server Node 4: mongod as Secondary of Shard 2, mongod as Arbiter of Shard 2
Client Nodes 1-8: YCSB, mongos
Client Node 1-2: mongod as Arbiter of Shard 1, Shard 2 respectively
Client Node 3: mongod as Config server [5]

The sharding key was the id generated by YCSB. Care must be taken when generating sharding keys to ensure that load is distributed properly across the shards. The zipfian distribution we used generated stable load across both replica sets.

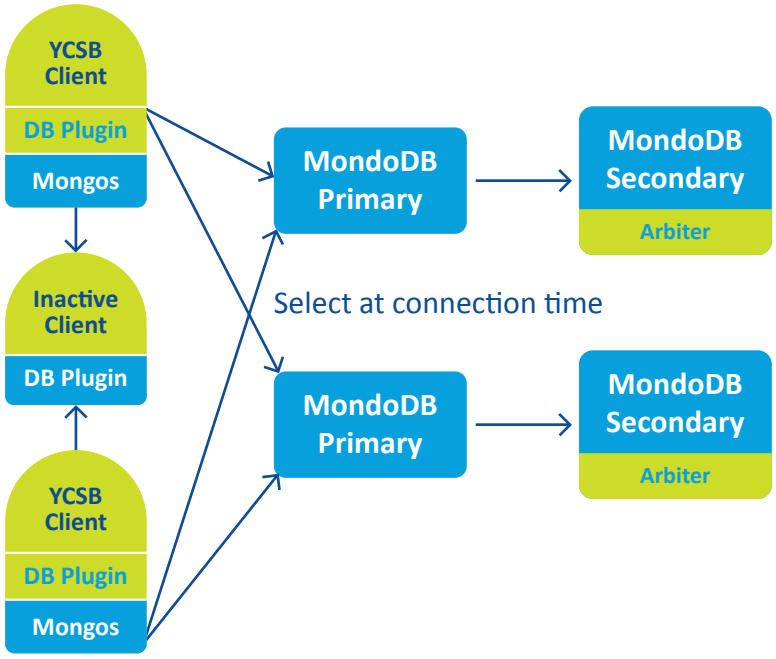
All write requests to MongoDB were done with writeConcern=normal. This caused writes to return successfully as soon as they are sent to the server (barring network failures), but before they are written to disk or even acknowledged as successfully committed to memory. This setting provides very weak durability guarantees, but we used it to maximize benchmark performance. All writes were done to the Primary nodes of replica-sets.

---

5. This is also non-production configuration. Production requires a replica-set of config servers, with at least three members. The cluster will not function if the config database is lost.

All read requests to MongoDB were done with readPreference=primaryPreferred. This meant that all read requests are also routed to the Primary. This may seem counterintuitive, but in our tests performed better than routing reads to the secondary. The large amount of replication traffic being processed by the Secondary actually made it slower than the Primary in servicing read requests.

Note: The ability to set readPreference was not part of YCSB. We upgraded the MongoDB driver from version 2.8.0 (from before the Mongo 2.2 release) to version 2.10.1 and allowed the readPreference to be set as a configuration. The new code has been submitted for approval to the YCSB master.
NUMA note: these servers did not use NUMA.
Journals were located on the same RAID0 of SSDs as the data.



The following steps were used to set up the cluster:

Initialization of Primary:
/opt/mongodb/bin/mongod --dbpath /mnt/raid/mongodb/data --replSet shard1 --logpath mongod.log --logappend --quiet --fork
rs.initiate({ _id: "shard1", members: [{ _id: 0, host : "e1.citrusleaf.local", priority: 2 }] })

Initialization of Secondary:

```
/opt/mongodb/bin/mongod --dbpath /mnt/raid/mongodb/data --replSet shard1 --logpath
mongod.log --logappend --quiet --fork
rs.add("e2.citrusleaf.local") # run on primary
```

Initialization of Arbiter:

```
/opt/mongodb/bin/mongod --dbpath /mnt/raid/mongodb/data --replSet shard1 --logpath mon-
god.log --logappend --quiet --fork rs.addArb("r2.citrusleaf.local")
```

Start config server:

```
/opt/mongodb/bin/mongod --dbpath /mnt/mongodb/data --configsvr --logpath mongod.log --logap-
pend --quiet --fork
```

Start mongos:

```
/opt/mongodb/bin/mongos --configdb r5.citrusleaf.local --logpath mongos.log --logappend --quiet
--fork
```

Initialize sharding:

```
sh.addShard("shard1/e1.citrusleaf.local")
sh.addShard("shard2/e3.citrusleaf.local")
sh.enableSharding("ycsb")
sh.shardCollection("ycsb.usertable", { "_id": 1 } )
```

Note: Dropping the collections causes the sharding data (saved on the config server) to be lost.

Appendix D: Test List

Load 50 Million (or 500 Million) records to 4 node cluster

Load the complete dataset into each database. This was done once and then reused for each of the following tests.

Metrics: throughput, average latency for insert operations.

Find maximum performance, Workload A

Run YCSB Workload A on the cluster without limiting the throughput artificially. A warm-up period of 10 minutes was used to prime the cache before metrics were gathered.

Metrics: throughput, average latency for read and update operations, cache hit ratio.

Find maximum performance, Workload B

Same as previous, but with workload B.

Find relationship of latency on throughput, Workload A

Run YCSB Workload A on the cluster while throttling throughput from YCSB. The throughput was increased until the it reaches the maximum found in the prior tests. The throughput levels tested were: 1k, 2k, 4k, 6k, 8k, 10k, 15k, 20k, 25k, 30k, 35k, 40k, 45k, 50k, 75k, 100k, 125k, 150k, 175k, 200k, 250k, 300k, 350k, 400k, 450k, but in some cases not every point was plotted to increase clarity.

Number of operations for each throughput level: 10,000,000

Metrics: graphs of average latency vs. throughput for read and update operations.

Find relationship of latency on throughput, Workload B

Same as previous, but with workload B.

Metrics: graphs of average latency vs. throughput for read and update operations.

Appendix E: YCSB Customizations

Lineate modified YCSB to provide multi-client automation as well as a variety of enhancements to increase load, improve stability, and test consistency and durability models. The customized code can be found at https://github.com/thumbtack-technology/ycsb. (It was taken from the sources committed to https://github.com/thumbtack-technology/ycsb on September 10, 2012.)

YCSB includes clients for MongoDB and Cassandra by default. We modified the MongoDB driver to support different read preferences.

The Couchbase client code was written by Couchbase and was taken from https://github.com/thumbtack-technology/ycsb. We modified it to support synchronous replication as a configuration option.

The Aerospike client code was provided by Aerospike. We worked with Aerospike

Configuration

YCSB runs in 32 threads on each client machine. We found this number to be optimal. Four or eight client machines were run simultaneously for most of tests.

Java was run using default settings.

Automation

A set of Fabric commands were added to the base install to provide automation across multiple client machines to perform tasks such as:

- Data loading

- Running workloads
- Checking status
- Aggregating logs
- Startup / shutdown

Instructions how to use these automation tasks can be found in the source.

List of Changes to Core YCSB

Upgrade of MongoDB client

We upgraded the MongoDB driver from version 2.8.0 (appeared before Mongo 2.2 release) to version 2.10.1 and allow the readPreference to be set as a configuration.

Also, now all the write errors are printed to stderr.

New configuration properties

mongodb.readPreference = primary|primaryPreferred|secondary|secondaryPreferred

Improvements of Aerospike client

We added the ability to display operations' result codes in more detail.

Throttling improvements

YCSB has features to limit throughput, but uses the average throughput for the whole experiment. This causes peaks after node failures in failover tests. We modified YCSB to keep the desired throughput on the same level, without peaks, by throttling based on the average throughput over the last 100 ms.

Output improvements

- Print current statistics to stderr every 2 secs instead of 10 secs
- Print intermediate statistics (identical to final) to stdout in every configured

time interval in order to avoid losing data on YCSB hangs or crashes
- Print final statistics on YCSB process shutdown

New configuration properties

exportmeasurementsinterval: interval time for exporting measurements in out stream in milliseconds (default: 1000)

Warm-up

This change forces YCSB to do some read operations before gathering statistics, in order to ensure the database is in a steady state. The length of the warm-up can be limited by number of operations or by time period. Note: This appears to cause some problems with Couchbase, so for the tests described in this report the warm-ups were done manually.

New configuration properties

warmupoperationcount: number of operations in warmup phase, if zero then don't warmup (default: 0)
warmupexecutiontime: execution time of warmup phase in milliseconds, if zero then don't warmup (default: 0)

Field name

By default YCSB names the database record fields as “field” + a number. This new configuration option allows replacing the “field” prefix with something shorter. This was critical for producing record sizes small enough not to saturate network bandwidth at very high throughput levels.

New configuration properties

fieldnameprefix: string prefix for the field name (default: “field”)

Retries

Added the ability to retry failed operations. These retries are done within the same operation, so they don't affect the number of operations reported (but do increase the reported latency of the operation, which we feel is fair).

The original YCSB stops if it encounters an error on insert; this setting allows retries on insert as well.

New configuration properties

readretrycount: number of retries if read fails, if zero then don't retry (default: 0)
updateretrycount: number of retries if update fails, if zero then don't retry (default: 0)
insertretrycount: number of retries if insert fails, if zero then don't retry (default: 0)
retrydelay: delay between retries in milliseconds (default: 0)

Reconnections

If YCSB stops operations for some reason (e.g. cluster reconfiguration or other issues which causes the working threads to be blocked) we force it to reconnect to the DB (reinitializing the DB client). This prevents many kinds of YCSB-related problems with failover tests.

New configuration properties

reconnectiontarget: the throughput value threshold when to do reconnect

Inserts with errors

The new configuration option was added to allow errors on inserts.

Usually YCSB stops when any operation fails on load phase. This setting makes it possible to ignore such errors and continue inserting.

New configuration properties

ignoreinserterrors: set to true to activate the new feature

THANKS FOR READING

CONTACT US