NoSQL Performance when Scaling by RAM

Benchmarking Cassandra, Couchbase, and MongoDB for RAM-heavy Workloads

Lineate

CONTENTS

OVERVIEW

In this study, we attempted to quantify how three of the most popular NoSQL databases scaled as more hardware was added to a cluster. The purpose was not to give an exhaustive answer on the scaling properties of different databases, but to address the primary reason why many companies will be choosing a NoSQL database: the ability to transparently and inexpensively scale up horizontally by adding hardware instead of vertically increasing processing power on a RDBMS.

We performed the studies using the YCSB benchmarking tool, which was developed originally by Yahoo to measure key-value storage performance and extended by Lineate to support added capacity and parallelization. We used this tool because most NoSQL benchmarks have been performed with it, and we wanted our results to be as comparable as possible to other studies.

NoSQL databases optimize performance by making different tradeoffs and assumptions about the workload. The configuration we chose to measure was the use case where the vast majority of data could fit into a server's RAM, but eventually persisted to disk. In our experience this is a common use case for companies servicing large request volumes, and we see it commonly used in things like user session storage, cookie matching, and application synchronization. There are other scaling strategies that can be better for other kinds of workloads or application needs, such as scaling by disk instead of RAM, scaling with strong durability guarantees, and others. Those are outside the scope of the research we performed.

This study was commissioned and subsidized by Couchbase. Lineate occasionally accepts outside funding to perform research activities that have substantial cost. In such cases, Lineate retains full independence in designing tests and interpreting results. When we perform such tests, we reach out to all the vendors involved in an attempt to present them as well as possible within the constraints of the test. Couchbase had no control over the test process other than offering guidance at maximizing Couchbase performance. At the same time, it is should also be noted that Couchbase would not have requested a study of a scaling strategy it would not do well on.

Rationale
Lineate has done a number of studies of NoSQL databases over the past two years, studying things like raw performance, failover behavior, cross datacenter replication, and effects of latency on eventual consistency. The purpose of these studies is to give insight into how these databases should be expected to perform by creating synthetic workloads to illustrate different properties. While no synthetic test can perfectly reproduce what happens in production, we feel that our tests have matched well with how we've seen these systems perform in the real world.

Database Categorization
We tried to measure here is how linearly the databases scaled. While not the only reason to use a NoSQL database (shortened development cycles is another major factor), it is the one that we at Lineate encounter the most. Our clients tend to jump to NoSQL when they've exceeded their ability to scale using more traditional technologies, and so our concern was measuring what this raw scaling behavior would mean (in the in-memory case). As such, we designed our tests purely around the Key-Value storage capabilities of each database instead of focusing on additional features such as document indexing or column features which are not directly comparable. Those features can be

TEST DESIGN & METHODOLOGY

Our hypothesis was that all three databases would scale at a near linear rate when operating under reasonably similar configurations. Since this was a test of raw key-value performance of primarily RAM-based datasets, we expected Couchbase to perform particularly well given its origins in Membase and memcached. Our goal was to measure how much traffic could be served at different hardware tiers while keeping latency at a reasonable level.

Databases Tested
Apache Cassandra 2.0.9, Couchbase 2.5.1, and MongoDB 2.6.4

Test Plan
It was important that the tests be transparent and externally reproducible, so we took a systematic approach towards defining the test steps.

Provision and Verify Clients and Servers

1. Configure the client and server hardware software according to vendor recommendations
2. Run a smoke test from a single client to determine baseline numbers
3. Increase traffic load
4. Make sure that network will not be the bottleneck for the cluster performance
5. Make sure that we measure maximum throughput
6. Make sure that client CPU or network is not artificially increasing latency

Run the Read-Heavy Workload on a 4-node Cluster

1. Reload database and populate a 20M record dataset by inserting one record at a time
2. Wait for the database to finish rebalancing
3. Clear the caches
4. Start all client YCSB nodes concurrently using the Read-Heavy Workload
5. Warm-up for 1 minute for the database to hit a steady state
6. Run the current load for 5 minutes under a consistent number of client threads
7. Incrementally increase the number of threads per client, running for 5 minutes per increment (steps 3 to 6)

Run the Balanced Workload on a 4-node Cluster
Do the same steps precisely, but for the Balanced Workload.

Repeat for Other Cluster Sizes
Run the same workloads (with appropriately scaled workloads) for other cluster sizes.

Repeat for Other Databases
Repeat all prior steps for each database.

Server Configurations
We initially planned to test both on bare metal and virtualized environments. The bare metal environment would obviously provide more consistent numbers, but the virtualized environment could show scaling behavior more linearly. However, we were unable to get consistent enough behavior to draw conclusions on either virtualized or collocated bare metal hardware. Therefore we measured performance on 4-, 6-, and 8-node clusters under our own full control, along with 32 client machines.

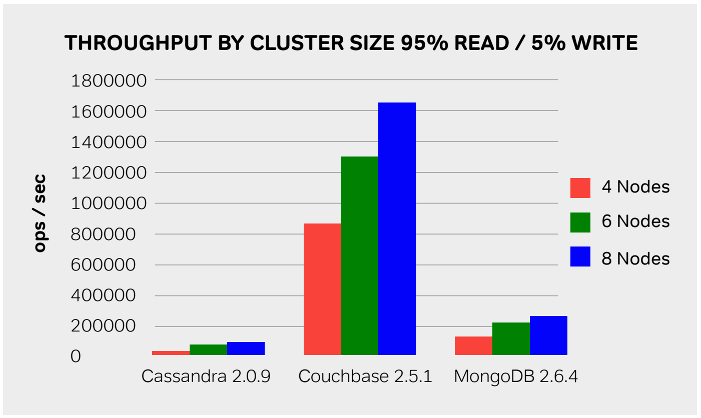
Database Configurations
A key consideration in benchmarking something as diverse as NoSQL databases is always to try to create as much of an apples-to-apples comparison as possible. This is easier said than done, since each NoSQL database makes different tradeoffs in its design considerations. Here are some of the assumptions we made.

- Both MongoDB and Couchbase trade durability for speed, while Cassandra trades consistency for speed. Since we were trying to measure maximum throughput, we ran Cassandra in Read.ONE / Write.ONE mode, which yields the highest numbers
    - Cassandra was thus on par with MongoDB and Couchbase in terms of replication of data (a single node failure results some data loss)
    - MongoDB and Couchbase offered immediate consistency in this configuration, whereas Cassandra offered eventual consistency

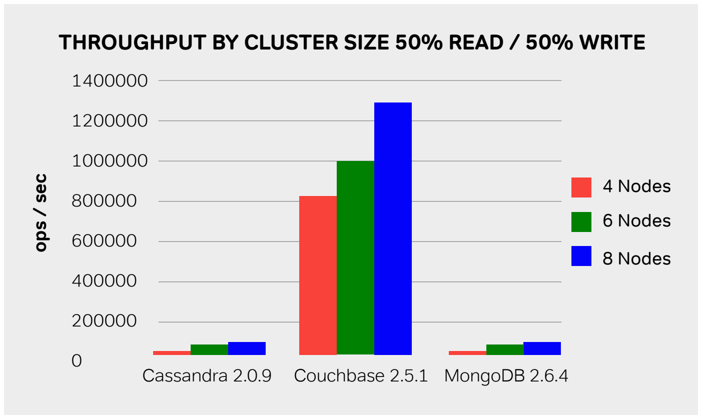
- We replicate each piece of data to two servers for each database
    - For Cassandra, a replication factor of 2 is not normally preferred, because it means there can be no quorum reads or writes. Since we were using only inconsistent reads and writes, we elected to for replication parity so that equivalent amounts of hardware could be used.
    - MongoDB is normally configured by replica sets, each of which has a Master and one or more Slaves. Since slaves don't participate in serving traffic, we configured each node (at MongoDB's suggestion) as both a Master for one replica set and a Slave for another, in order to make sure all nodes were serving traffic.
- MongoDB and Couchbase are faster at reads, and Cassandra is faster at writes, so we test two workloads:
    - 95% read / 5% write
    - 50% read / 50% write
- For Cassandra:
    - The data were collected using DataStax Cassandra Java Driver with TokenAware policy enabled.
    - DataStax recommends using the CQL3 driver instead of the Thrift driver. Our experience suggested that throughput was slightly higher but with increased latency when using CQL3, so we reported the (better) Thrift results. Further study is warranted.

RESULTS

Our results showed that Couchbase performed quite well, as we expected for this RAM-based use case.



Cluster size	Cassandra	Couchbase	MongoDB
4 Nodes	60,281	903,430	130,998
6 Nodes	72,983	1,284,336	194,831
8 Nodes	99,089	1,878,120	227,467



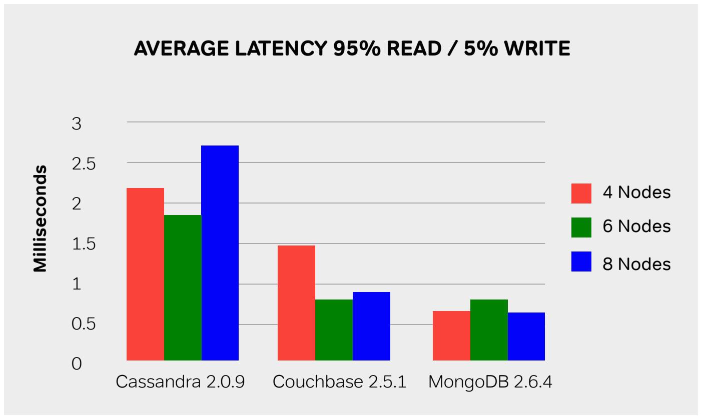
Cluster size	Cassandra	Couchbase	MongoDB
4 Nodes	53,896	800,660	50,120
6 Nodes	67,427	1,013,841	68,384
8 Nodes	89,728	1,246,967	85,928

Latency

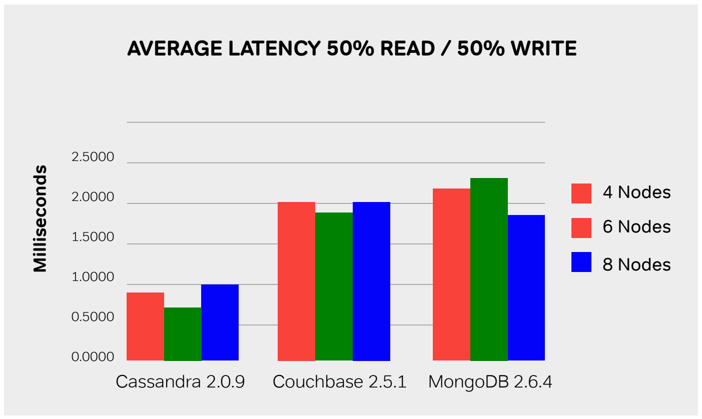
Latency is trickier to measure than throughput, because as load increases, latency tends to increase as well. Latency is best measured relative to throughput being achieved for the particular database. At the same time, we do want to compare results because latency is of critical importance when choosing a database.

Consider the case where a database has a fixed capacity of 100,000 operations per second. Doubling the number of threads connecting to this database will not increase throughput, so the result is that each thread will wait twice as long to service the same request. Often it's possible to squeeze out a few thousand more transactions per second at the cost of a large spike in latency. So which is the correct result to measure?

For our results, we chose the latency number that was in effective before it started to rise rapidly in order to squeeze in the last bit of throughput. This was an inherently subjective choice, but it's one that any user of these systems will make when performing capacity planning.



Cluster size	Cassandra	Couchbase	MongoDB
4 Nodes	2.173557066	1.477351476	0.781675911
6 Nodes	1.949606279	0.891571832	0.838519234
8 Nodes	2.656504164	0.900483548	0.743169516



Cluster size	Cassandra	Couchbase	MongoDB
4 Nodes	0.8879	1.9841	2.0985
6 Nodes	0.6793	1.9103	2.1711
8 Nodes	0.9940	1.9813	1.8555

Scalability
In a perfect world, doubling hardware will double throughput, but of course systems tend to scale less than perfectly linearly. We created a scalability score that represents how well a database scaled from 4 to 8 nodes. It was the average of the scaling score from 4 to 6 and from 4 to 8 nodes. The details are summarized below.

Read Heavy Workload	Scaling 4 to 6	Scaling 4 to 8	Average Efficiency (out of 100%)*
Cassandra 2.0.9	21.07%	64.38%	53.26%
Couchbase 2.5.1	42.16%	89.57%	86.95%
MongoDB 2.6.4	48.73%	73.64%	85.55%

Balanced Workload	Scaling 4 to 6	Scaling 4 to 8	Average Efficiency (out of 100%)*
Cassandra 2.0.9	25.10%	66.48%	58.35%
Couchbase 2.5.1	26.63%	55.74%	54.50%
MongoDB 2.6.4	36.44%	71.44%	72.16%

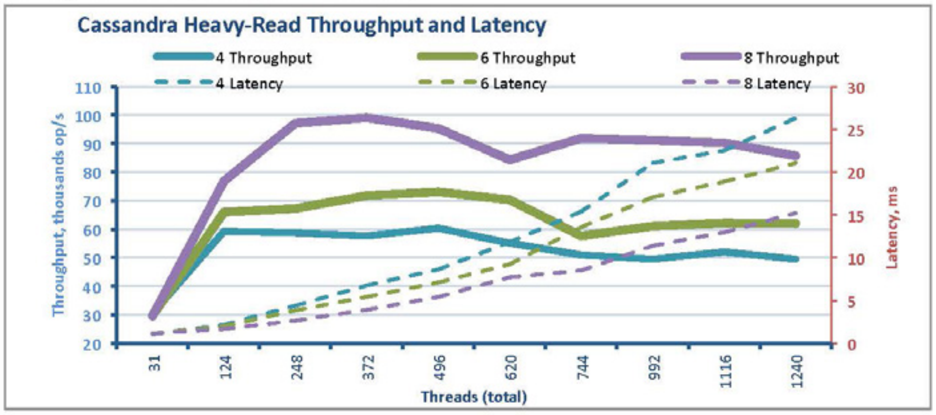
* The average of the two scores, normalized for 100% = purely linear scaling

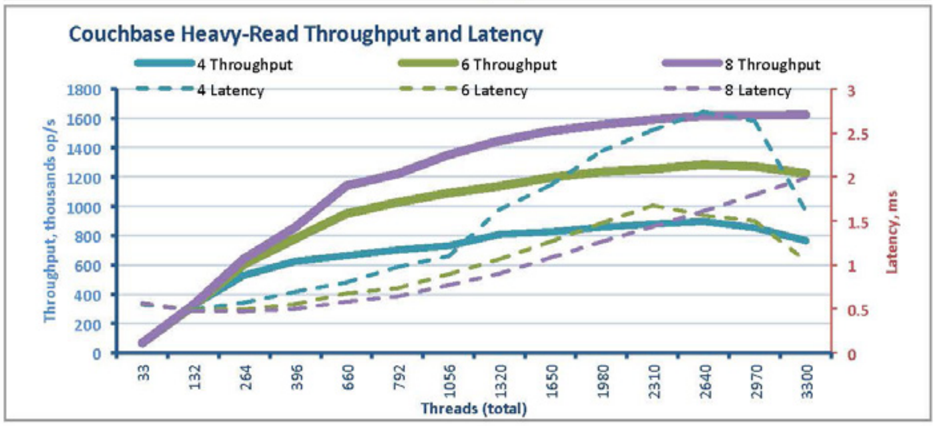
For heavy-read workloads (95% read, 5% write), both Couchbase and MongoDB scale close to linearly. For Cassandra, the answer is less clear, as the last two nodes had more impact than the first 2. A cluster size of 4 is not common for Cassandra installations.

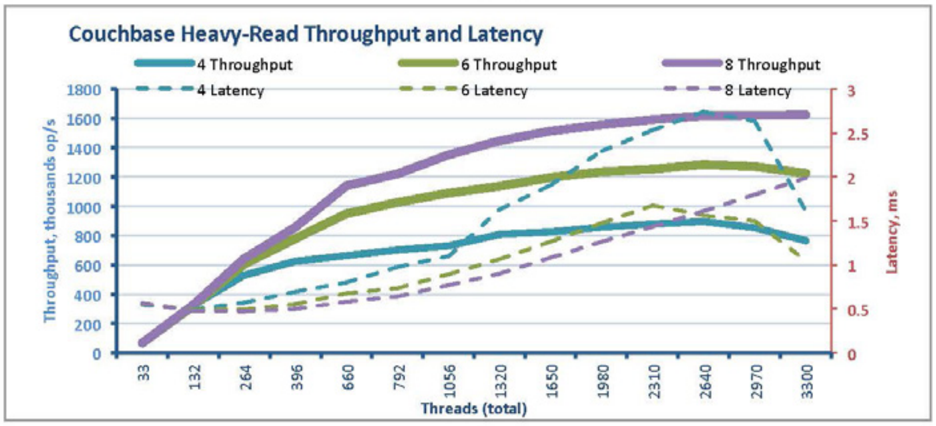
For balanced workloads (50% read, 50% write), MongoDB showed the best scaling behavior. Couchbase and Cassandra both show scaling at about half of linear capacity throughout. Cassandra actually scales slightly better for the balanced workload.

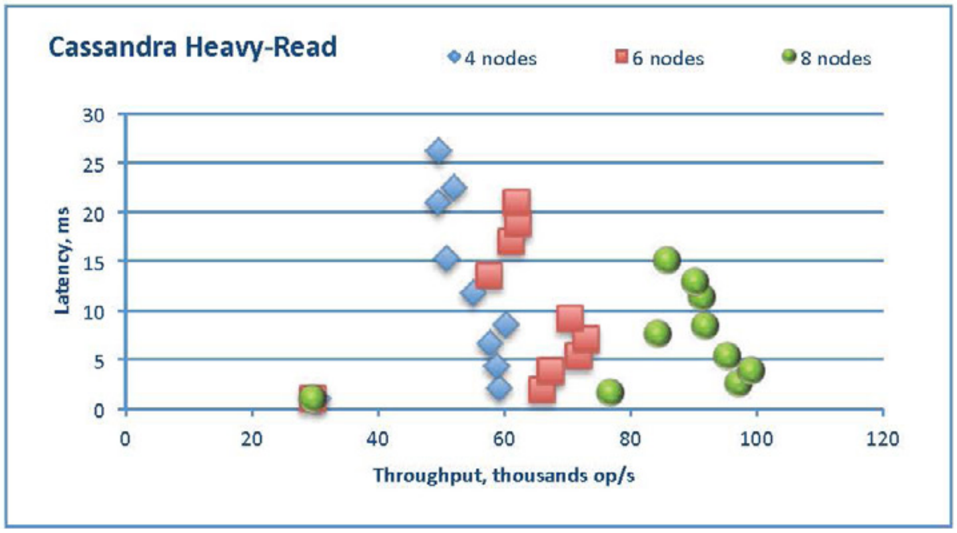
Test Repeatability
We repeated tests in the final configuration two times to ensure results are stable. Maximum difference of throughput for the same conditions we saw was 4.5%.

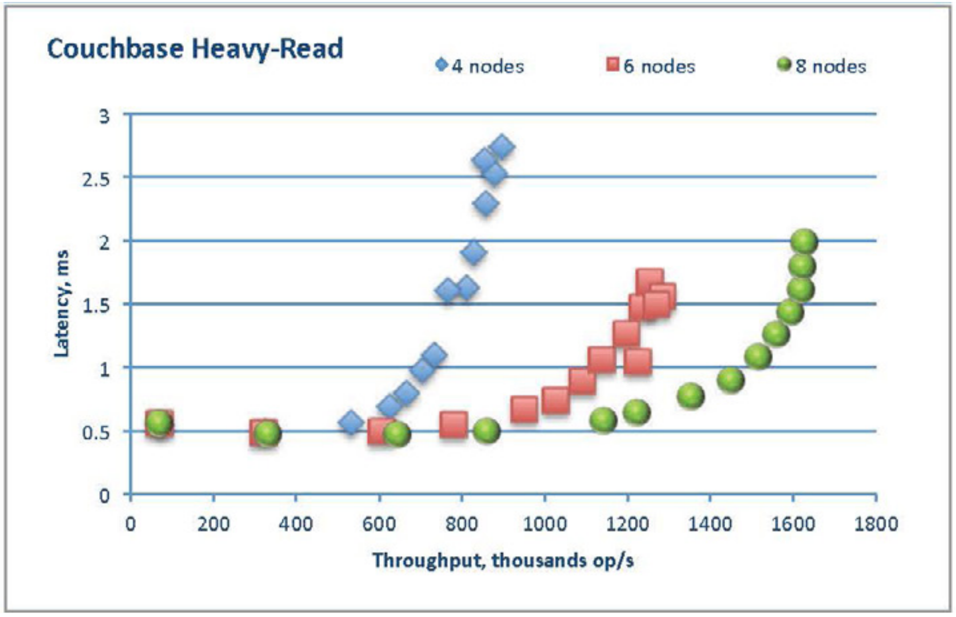
Heavy Read Workload — Details

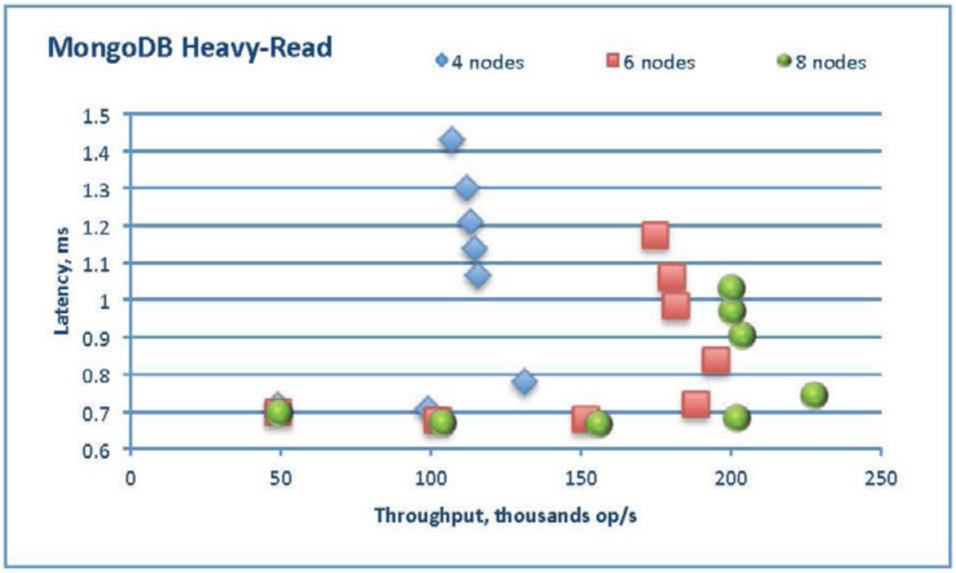




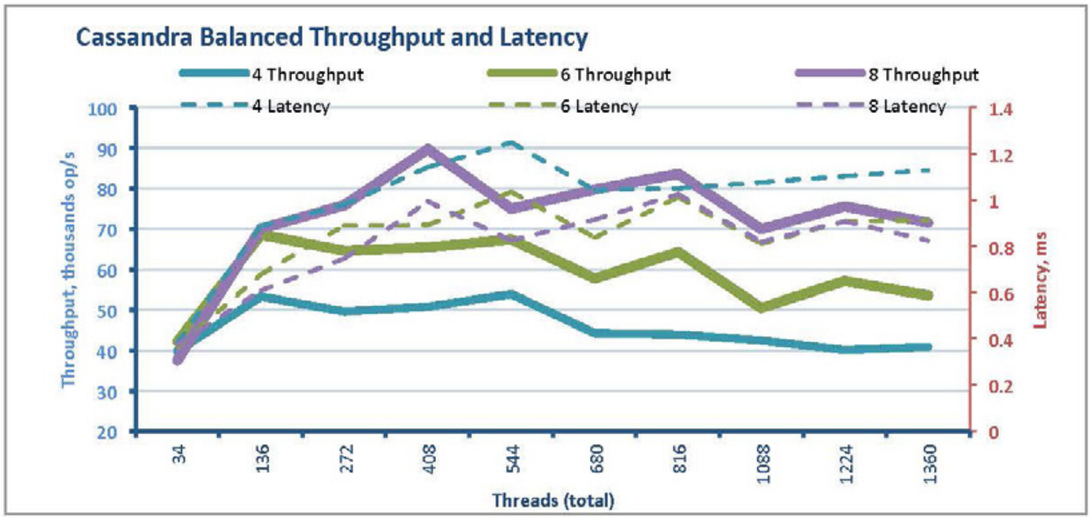


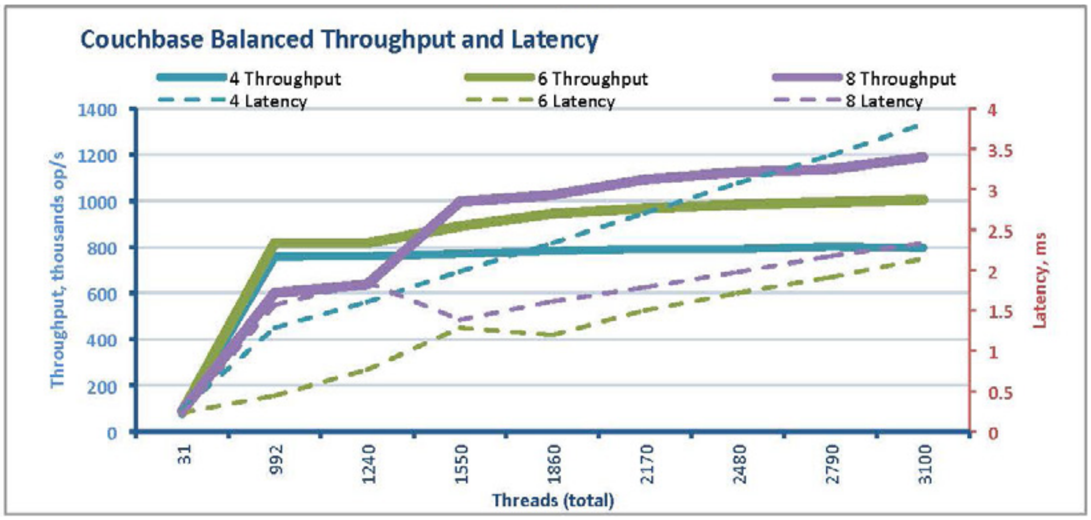

Cassandra Heavy-Read
4 nodes
6 nodes
8 nodes
Latency, ms
Throughput, thousands op/s
0
5
10
15
20
25
30
0
20
40
60
80
100
120

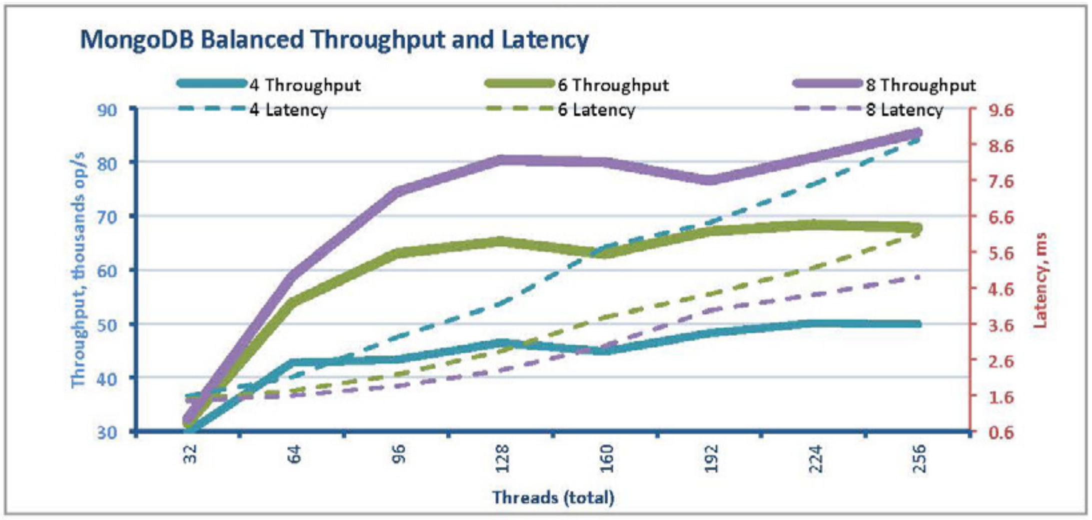

Couchbase Heavy-Read
4 nodes
6 nodes
8 nodes
Latency, ms
Throughput, thousands op/s
0
0.5
1
1.5
2
2.5
3
0
200
400
600
800
1000
1200
1400
1600
1800

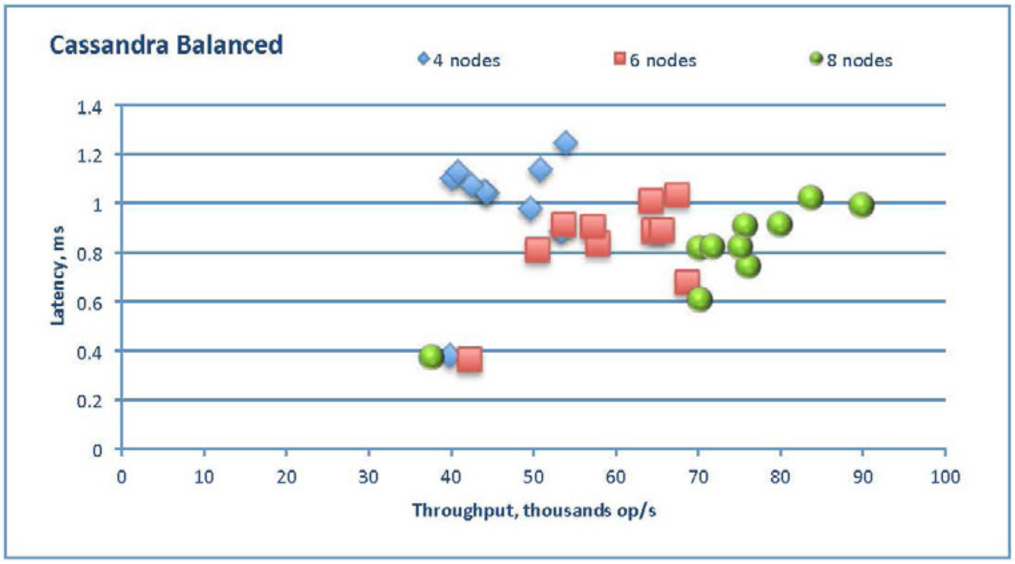

MongoDB Heavy-Read
4 nodes
6 nodes
8 nodes
Latency, ms
Throughput, thousands op/s
0.6
0.7
0.8
0.9
1
1.1
1.2
1.3
1.4
1.5
0
50
100
150
200
250

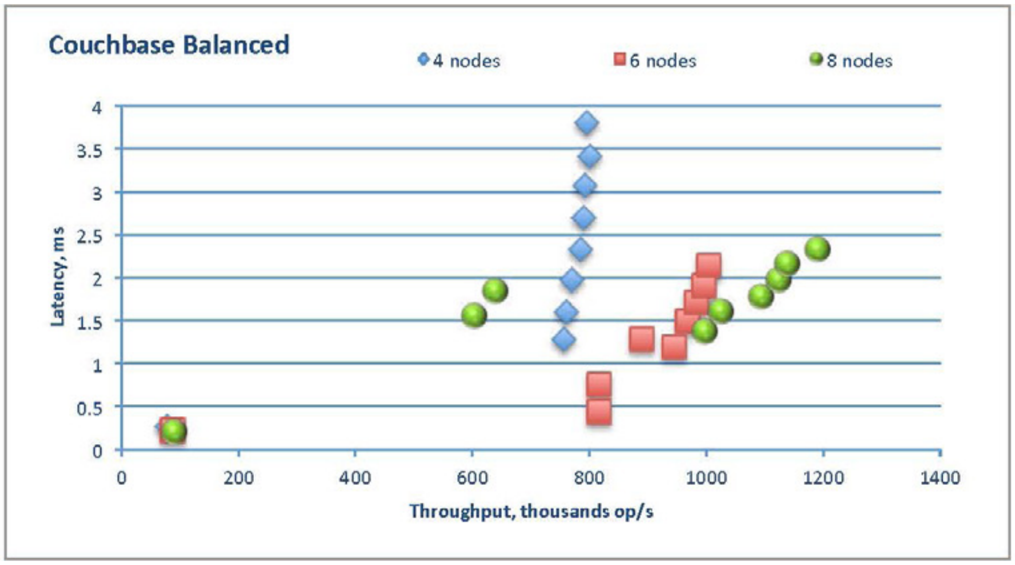
Balanced Workload — Details

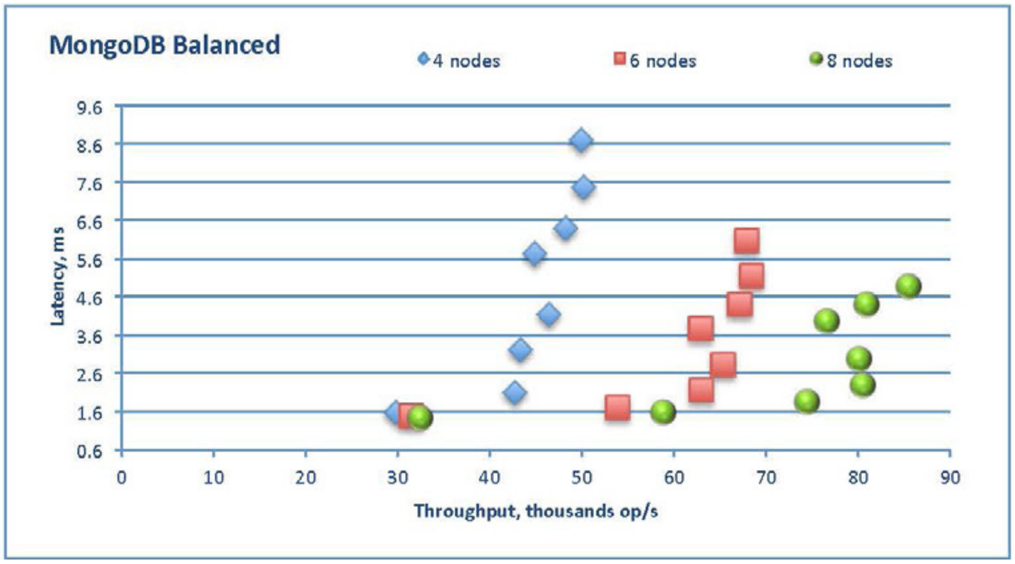







Cassandra Balanced
4 nodes
6 nodes
8 nodes
Latency, ms
Throughput, thousands op/s
0
0.2
0.4
0.6
0.8
1
1.2
1.4
0
10
20
30
40
50
60
70
80
90
100


Couchbase Balanced
4 nodes
6 nodes
8 nodes
Latency, ms
Throughput, thousands op/s
0
0.5
1
1.5
2
2.5
3
3.5
4
0
200
400
600
800
1000
1200
1400


MongoDB Balanced
4 nodes
6 nodes
8 nodes
Latency, ms
Throughput, thousands op/s
0.6
1.6
2.6
3.6
4.6
5.6
6.6
7.6
8.6
9.6
0
10
20
30
40
50
60
70
80
90

CONCLUSION

All three databases were relatively simple to scale by adding nodes to the cluster (though MongoDB took a while to replicate.) As we expected, none of the databases scaled completely linearly, though Couchbase and MongoDB both came quite close in the Read Heavy case. Cassandra showed the most consistency across workloads, which was not unexpected given the write-optimized nature of Cassandra.

Couchbase dominated in terms of raw performance, which was consistent with our prior tests and with our expectations for RAM-heavy configurations. This is Couchbase's sweet spot and it performed as both Couchbase and Lineate expected it would.

Scaling is of course a complicated topic, and we measured only 4, 6, and 8 node clusters. It would be very interesting to examine scaling behavior to significantly larger cluster sizes, and would dramatically strengthen our conclusions on linear scalability. As discussed earlier, such a test is difficult to do using virtualized hardware, and given that this existing test used 40 or more dedicated machines, scaling much further would be impractical from a cost perspective.

There are several important dimensions that have not been explored in this study. A major one is examining the strategy of scaling data by disk instead of RAM. Configurations where the amount of data stored greatly exceeds RAM and where the cache hit ratio is lower is a sweet spot for Cassandra, and we report no data on this case. It is certainly more cost effective in general to scale with disk instead of RAM, and a study to explore how such clusters would behave would be incredibly valuable.

Another thing not explored in this study are the tradeoffs involved regarding consistency, durability, and other reliability parameters. Some of our other reports explore this in more depth, and we encourage people to examine those in conjunction with this study. The fact of the matter is that different databases achieve higher performance by sacrificing different things, and it is critical to understand the various tradeoffs before making broad conclusions.

These caveats aside, however, we find the use case where the working copy of data fits primarily into RAM to be a common one for our clients, and hope these results provide valuable insight to those looking into solutions.

Appendix A: Hardware Specifications

8 Servers

- CPU: 2 x Intel Xeon E5-2620 V2 2.1GHz (6 cores)
- RAM: 4 x 16Gb Kingston ValueRAM KVR13R9D4/16 (64GB total)
- MB: SuperMicro 1U 6017R-WRF
- SSD: 2 x 100Gb SATA 6Gb/s intel DC S3700 - for data (200GB total, 70GB RAID-1)
- 80Gb SATA 6Gb/s Intel DC S3500 - for OS
- Network: Intel I350T2BLK PCI-E x4 (1000Mbits)

Note: *MongoDB recommended using RAID-10 for maximal performance. Our verification tests showed minimal performance improvement in this configuration, so we fell back to RAID-1.*

32 Clients

- CPU: 4 Core Intel(R) Core(TM) i5-4440 CPU @ 3.10GHz 3GHz
- RAM: 8Gb
- Network: Realtek Semiconductor Co., Ltd. RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller

One 1Gbit switch

- 96Gbit/second communication matrix
- 48 ports
- all Clients and Servers are directly connected through it

One Orchestration Server
This server was simply used to gather and aggregate test data from the clients. The configuration of this server is unimportant to the results.

Appendix B: Dataset

Dataset
For each database, depending on the cluster size we used 3 different datasets:

	4 Node cluster	6 Node cluster	6 Node cluster
Number of records	20M	30M	40M
Number of fields per record	10	10	10
Field size	150 Bytes	150 Bytes	150 Bytes
Total record size	1,500 Bytes	1,500 Bytes	1,500 Bytes
Total dataset size	27.9 GB	41.9 GB	55.8 GB

Notes:

- We are aware that a dataset of 20M records is small
    - If we modestly increased the size of the dataset, different databases started keeping different amounts of data in RAM, creating results that were not easily comparable
    - The scaling behaviors within each database were consistent at any record size
    - We elected to report on this small dataset size as the "all-RAM" number

Appendix C: OS Configuration

In order to run the test correctly we asked Database vendors to recommend system-level parameters.

General Parameters

1. Setup NOOP scheduler for SSD
   # echo noop > /sys/block/sda/queue/scheduler
2. Decrease read-ahead value to 8
   # blockdev --setra 8 /dev/sda

3. Increase limit of number of simultaneously opened files and number of processes.

```
# vim /etc/security/limits.conf
*- nofile 100000
*- nproc 32768
```

4. Synchronize time using ntpd

```
# chkconfig --level 0123456 ntpd on
# /etc/init.d/ntpd start
```

5. Set up network card interrupts

```
# cat /proc/interrupts | grep em1
95:  1  0  0  0  0  0  0  0  0  0  0  0 IR-PCI-MSI-edge em1
96: 25344  0  0  0  0  0  0  0  0  0 337041437  0 IR-PCI-MSI-edge em1-TxRx-0
97: 5511  0  0  0  0  0  0  0  0 367989305  0  0 IR-PCI-MSI-edge em1-TxRx-1
98: 12685  0  0  0  0  0  0  0 300775727  0  0  0 IR-PCI-MSI-edge em1-TxRx-2
99: 9585  0  0  0  0  0  0 327067958  0  0  0  0 IR-PCI-MSI-edge em1-TxRx-3
100: 39700  0  0  0  0  0 246153034  0  0  0  0  0 IR-PCI-MSI-edge em1-TxRx-4
101: 5726  0  0  0  0 331411642  0  0  0  0  0  0 IR-PCI-MSI-edge em1-TxRx-5
102: 28678  0  0  0 301331698  0  0  0  0  0  0  0 IR-PCI-MSI-edge em1-TxRx-6
103: 28813  0  0 329040811  0  0  0  0  0  0  0  0 IR-PCI-MSI-edge em1-TxRx-7
```

```
# cat /proc/irq/56/smp_affinity
```

The value stored in this file is a hexadecimal bit-mask representing all CPU cores in the sys tem. If the machine has 4 cores, then the value should be 15 (1111). In this case all cores will use for handling requests for interrupts from network devices.

6. Disable access time and disable barrier

```
# vim /etc/fstab
```

add noatime,nodiratime,barrier=0 option to the / and /data file systems.

```
UUID=...     /              ext4    defaults,noatime,nodiratime,barrier=0   11
UUID=...       /data        ext4    defaults,noatime,nodiratime,barrier=0   11
```

then remount

```
# mount -o remount /data
```

and check if the partition really got mounted with noatime

```
# mount
```

7. Overprovisioning
   Left 20% of free space on SSD drive for overprovisioning.

8. Disabling THP (transparent hugepages)
   Add the following to /etc/rc.local

```
# vim /etc/rc.local:
for i in /sys/kernel/mm/*transparent_hugepage/enabled; do echo never > $i; done for i in /
sys/kernel/mm/*transparent_hugepage/defrag; do echo never > $i; done
```

9. Turn Swappiness OFF
   # sysctl vm.swappiness=

Cassandra-specific Parameters

1. Update /etc/security/limits.d/cassandra.conf with following
   cassandra - memlock unlimited
   cassandra - nofile 100000
   cassandra - nproc 32768
   cassandra - as unlimited

2. Update /etc/security/limits.d/90-nproc.conf with following
   *- nproc 32768

3. Add following line in /etc/sysctl.conf
   vm.max_map_count = 131072
   then run the following command
   # sysctl -p

4. Disable swap entirely
   Included in common recommendations.

5. Per Matt Kennedy recommendations for SSD:
   Put the following into /etc/rc.local
   # vim /etc/rc.local
   # echo deadline > /sys/block/<sda>/queue/scheduler
   # echo 0 > /sys/block/<sda>/queue/rotational
   # setra 0 /dev/<sda>

6. Cassandra.YAML tuning
   memtable_flush_writers: 8
   trickle_fsync: true
   compaction_throughput_mb_per_sec up: 128

7. Disable the commit log.
   see http://www.datastax.com/docs/1.1/references/cql/CREATE_KEYSPACE

8. Use current driver as token aware is now the default policy
   see_http://www.datastax.com/doc-source/developer/java-apidocs/com/datastax/driver/core/policies/TokenAwarePolicy.html

Couchbase-specific Parameters

1. Disable NUMA
   add at the start of command =”$COUCHDB -b” in /opt/couchbase/etc/init.d/couchdb # vim /opt/couchbase/etc/init.d/couchdb
   numactl --interleave=all
   # echo 0 > /proc/sys/vm/zone_reclaim_mode

2. Mount with BARRIER turned off
   Add barrier=0 to file system options in /etc/fstab and re-mount device.

3. Increase memcached threads to 8
   # curl --data “[ns_config:update_key({node, N, memcached}, fun (PList) -> lists:keystore(verbosity, 1, PList, {verbosity, ‘-t 8’}) end) || N <- ns_node_disco:nodes_wanted()].” -u Administrator:<password> http://<ip>:8091/diag/eval
   You only need to run this once against a cluster, but it will restart all the memcached process es at once.

4. Ensure compaction is turned off.
   check - Override the default autocompaction settings
   uncheck all other options to run compaction.

5. Increase bucket reader/writers to 8
   this is done through the IO on the bucket settings page.

6. Optimize Workload B (95/5% read/write)
   # wget -O- --user=Administrator --password=password --post-data=’ns_bucket:update_bucket_props(“default”, [{extra_config_string, “workload_optimization=read”}]).’ http://localhost:8091/diag/eval

MongoDB-specific Parameters

1. Place journal, log and data files on separate storage devices.

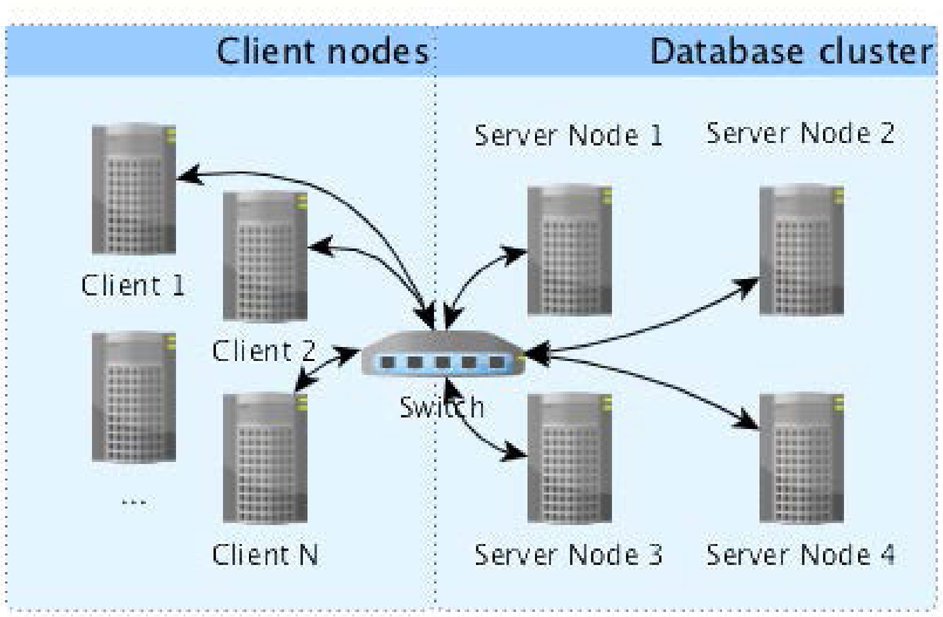
2. Filesystem: Use EXT4 or XFS filesystems; avoid EXT3.
   EXT4 filesystem is used.

3. UNIX ulimit settings
   Set limits as recommended in the MongoDB documentation:
   file size:
   # ulimit -f unlimited
   cpu time:
   # ulimit -t unlimited
   virtual memory:
   # ulimit -v unlimited
   open files:

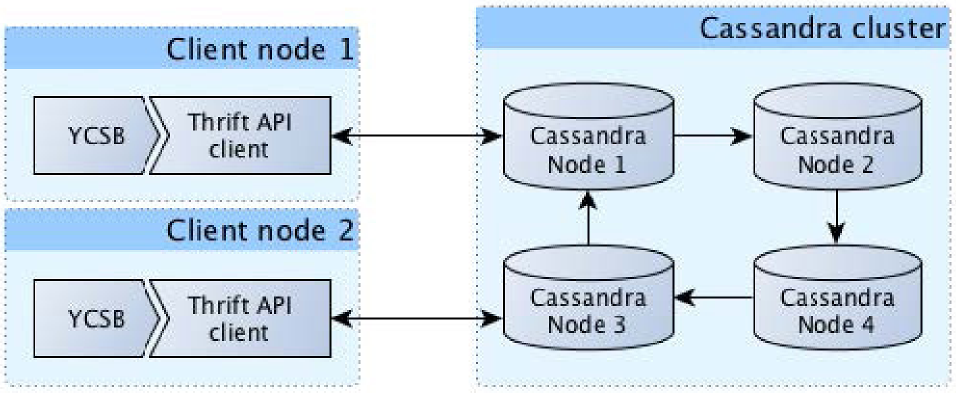
# ulimit -n 64000
memory size:
# ulimit -m unlimited
processes/threads:
# ulimit -u 64000
Always remember to restart your mongod and mongos instances after changing the ulimit settings to make sure that the settings change takes effect.

4. Most MongoDB deployments should use disks backed by RAID-10.
   2 SSD disks joined into RAID1 array were used instead.

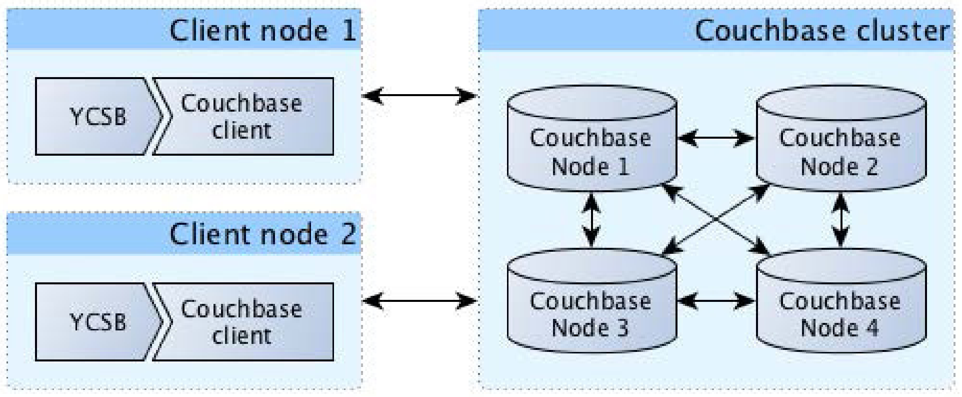
Appendix D: Database Topology
Network topology



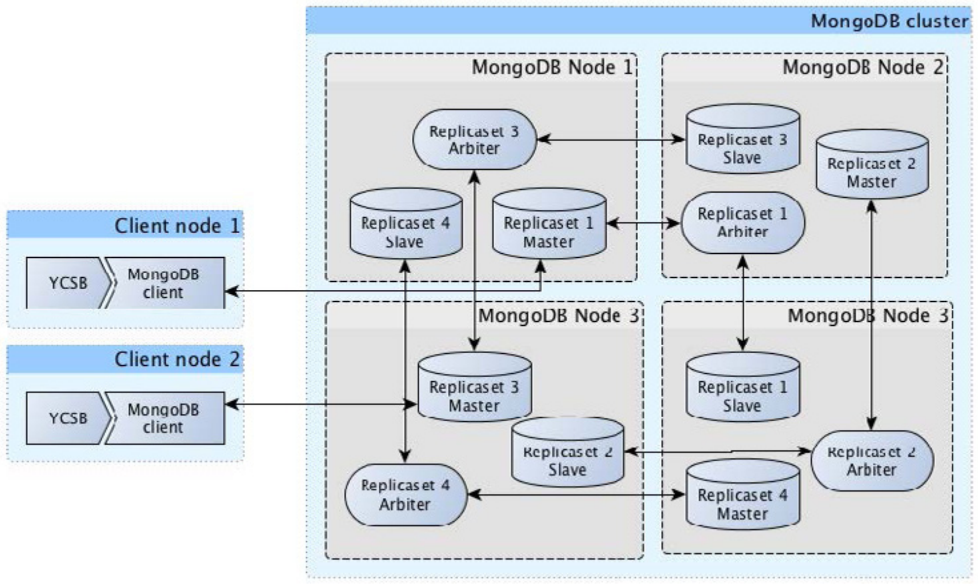
Cassandra
A Cassandra cluster uses a ring topology. Our test used the Thrift client through YCSB.
YCSB -> CassandraClient -> Thrift -> Cassandra Cluster



Couchbase
Couchbase uses a star topology. Each server talks with all others.



MongoDB



The following topology was recommended by MongoDB: master, slave and arbiter installed on each physical server, where master and slave store the data and the arbiter simply participates in elections. The main point was that every server is a primary node for particular shard.

server-1: {master A, slave B, arbiter C}

server-2: {master B, slave C, arbiter D}

server-3: {master C, slave D, arbiter A}

server-4: {master D, slave A, arbiter B}

THANK YOU

CONTACT US