END POINT

Benchmarking Top NoSQL Databases

Apache Cassandra, Couchbase, HBase, and MongoDB

Originally Published: April 13, 2015
Revised: May 27, 2015
http://www.endpoint.com/

Table of Contents

Executive Summary

End Point performed a series of performance tests on Apache Cassandra, HBase, and MongoDB using the YCSB benchmark in February 2013. Because performance enhancements have been made to each database since that time, End Point was commissioned by DataStax to conduct a retest of each database, with Couchbase also included as it is sometimes used as a NoSQL caching layer for certain application workloads.

The overall conclusions of the performance tests closely mirrored the results of the previous benchmark, with Cassandra outperforming the other NoSQL databases in both throughput and latency. Depending on the test, HBase or Couchbase delivered the next best result, with MongoDB coming in last in most tests.

Overview

NoSQL databases are fast becoming the standard data platform for applications that make heavy use of telecommunication or Internet-enabled devices (i.e. browser-based, sensor-driven, or mobile) as a front end. While there are many NoSQL databases on the market, various industry trends suggest that the top three in use today are MongoDB, Apache Cassandra, and HBase.[1]

When it comes to performance, it should be noted that there is (to date) no single “winner takes all” among the top three NoSQL databases or any other NoSQL engine for that matter. Depending on the use case and deployment conditions, it is almost always possible for one NoSQL database to outperform another and yet lag its competitor when the rules of engagement change.

While it is always recommended that anyone assessing a database’s performance (NoSQL or otherwise) test the engine under the specific use case and deployment conditions intended for a particular production application, general competitive benchmarks of usual-and-customary application workloads can be useful to those evaluating NoSQL databases.

This paper documents an update to a benchmark published in February 2013 by End Point that compared the performance of Cassandra, HBase, and MongoDB. In this series of tests, Couchbase was also included.

Test Methodology and Configuration

End Point performed the benchmark on Amazon Web Services EC2 instances, which is an industry-standard platform for hosting horizontally scalable services such as the tested NoSQL databases. In order to minimize the effect of AWS CPU and I/O variability, we performed each test three times on three different days. New EC2 instances were used for each test run to further reduce the impact of any “lame instance” or “noisy neighbor” effect on any one test.

The tests ran on Amazon Web Services EC2 instances, using the latest instance generation. The tests used the i2.xlarge class of instances (30.5 GB RAM, 4 CPU cores, and a single volume of 800 GB of SSD local storage) for the database nodes. As client nodes, the c3.xlarge class of instances (7.5GB RAM, 4 CPU cores) were used to drive the test activity.

[1] As an example, see “MongoDB, Cassandra, and HBase -- the three NoSQL databases to watch” by Matt Asay, InfoWorld, November 2014: http://www.infoworld.com/article/2848722/nosql/mongodb-cassandra-hbase-three-nosql-databases-to-watch.html.

The instances used an Ubuntu 14.04 LTS AMI in HVM virtualization mode, customized with Oracle Java 7 and with the software for each database installed. On start up, each instance was provided with its configuration, and called back to a parent server to retrieve any further configuration information needed. A script was used to drive the benchmark process, including managing the start up, configuration, and termination of EC2 instances, calculation of workload parameters, and issuing the commands to the clients to run the tests.

Each test started with an empty database. The clients were instructed to load an initial set of randomly generated data. Once the data was loaded, each workload (described below) ran in sequence. All client instances were instructed to run the workloads in parallel, and then wait until all client instances completed the workload before continuing. In between each workload sometimes database health and readiness checks were performed. For example, tests for Cassandra and Couchbase checked for any ongoing compaction processes, and waited until those completed before continuing to the next workload.

The client software was a customized YCSB install, built from the open source repository at https://github.com/joshwilliams/YCSB. The customizations were primarily pulled from alternate repos where database-specific improvements had been made, as well as driver version upgrades, and code improvements to allow YCSB to run on the expanded dataset used for this test.

Cassandra Software

The Cassandra version used was 2.1.0, installed from source. It received its node ID from the parent server on start up, and then replaced tokens in a cassandra.yaml template as needed. The first node was declared the seed_provider in cassandra.yaml so that the subsequent hosts could find each other and build the cluster automatically.

Once the cluster was ready, the driver script connected to the first node to create the needed keyspace and column family, running the following commands via the cqlsh command line utility:

```
create keyspace ycsb
WITH REPLICATION = {'class' : 'SimpleStrategy', 'replication_factor': 1 };

create table ycsb.usertable
(y_id varchar primary key, field0 blob, field1 blob, field2 blob,
field3 blob, field4 blob, field5 blob, field6 blob, field7 blob, field8 blob,
field9 blob) with compression = {'sstable_compression': ''};
```

Couchbase Software

For Couchbase, the latest stable version of Couchbase Server Community Edition, 3.0.1, was installed into the AMI via packages provided at http://www.couchbase.com/nosql-databases/downloads#Couchbase_Server.

On startup, it received its node ID from the parent server. The first node performed a cluster-init command on itself via couchbase-cli and set its ramsize to 3/4 the system RAM, or about 23GB on the i2.xlarge class of instances. It then created the default bucket to use all of the allocated RAM and set the bucket-eviction-policy to fullEviction.[2] Subsequent nodes waited until the process completed, then joined the cluster by issuing a rebalance command to the first node and requesting a server-add of its own IP address. After this process the driver script didn’t need to perform any further action to configure the cluster.

[2] Because Couchbase was only able to process a little less than 60% of the data volume with the "value ejection" setting, "full ejection" was used instead.

HBase Software

The HBase install used the combination of the latest stable Hadoop (2.6.0) and the latest stable HBase (0.98.6-1) as of the benchmark's start date. It received its node ID from the head node on start up. The template files core-site.xml, hdfs-site.xml, and hbase-site.xml were read with tokens replaced to configure the cluster. The first node declared itself the master and ran as both the Hadoop namenode and the HBase master, and ran the Zookeeper process as well. All nodes then started as a datanode and regionserver, pointing at the master node's Zookeeper for coordination.

Once the cluster was ready, the driver script connected to the first node and created the table. To configure the cluster, we ran:

```
create 'usertable', {NAME => 'data'}, {SPLITS => [(split computation
here)]}(in contrast to the version described in the paper that had
COMPRESSION => 'GZ' included.)
```

MongoDB Software

The MongoDB installation used the latest preview release candidate (2.8.0-rc4, which eventually became MongoDB 3.0) as of benchmark's start date so that the new Wired Tiger storage engine could be used.

At the start of the test, the software was installed via the precompiled x86_64 tarball. It received its node ID from the head node on start up. The first node considered itself the configuration server and started a dedicated mongod instance for that. All nodes each then started the mongod process for holding the data, with a mongos process pointing at the configuration server, and then running "sh.addShard()" on itself with its own local IP address.

Once the cluster was ready, the driver script connected to the first node and marked the collection to enable sharding as:

```
sh.enableSharding("ycsb")
sh.shardCollection("ycsb.usertable", { "_id": 1})
```

Then, using the same numeric-based distribution calculated for HBase, the script pre-distributed the chunks, running the following for each shard:

```
db.runCommand({{split:"ycsb.usertable", middle: {{_id: "user(calculated
range)"}}}})
db.adminCommand({{moveChunk: "ycsb.usertable", find:{{_id: "user(calculated
range)"}}, to: "shard(node ID)"}})
```

Workload Selection

The overarching goals for the tests were to:

- Select workloads that are typical of today's modern applications
- Use data volumes that are representative of 'big data' datasets that exceed the RAM capacity for each node
- Ensure that all data written was done in a manner that allowed no data loss (i.e. durable writes), which is what most production environments require

Five workloads were selected for the benchmark, with the following distribution of operations:

- Read-mostly workload: based on YCSB's provided workload B: 95% read to 5% update ratio

- Read/write combination: based on YCSB's workload A: 50% read to 50% update ratio
- Read-modify-write: based on YCSB workload F: 50% read to 50% read-modify-write ratio
- Mixed operational and analytical: 60% read, 25% update, 10% insert, and 5% scan
- Insert-mostly combined with read: 90% insert to 10% read ratio

In all cases both fieldcount and fieldlength was set to 10 in order to produce 100 byte records. In all workloads except the request, distribution was set to "uniform". The workload that involved scans (Mixed) had a max scan length parameter limiting it to 100 records. To obtain a long-running sample of performance, the operation count was set to 9 million operations for each workload. However to ensure that the tests completed in a reasonable amount of time, the max execution time was limited to 1 hour if that operation count wasn't reached by then.

Other parameters were calculated at test run time by the driver script. The record count was calculated based on the number of requested data instances for the test to generate 500 million records per data node. Due to the smaller size of the records and the difference in overhead for each database, this resulted in a different amount of data per node for each type of test.

To ensure the clients didn't create a bottleneck, a client instance was started for the data instance. Each client instance also targeted 256 threads per data instance during the workloads, though to ensure an error-free dataset, the thread count was temporarily lowered to 40 threads per client during the load process.

Result Collection and Aggregation

YCSB provides its results in a fairly structured text format for each workload. For each workload the overall throughput in operations/second and the total test runtime were gathered, along with the operation specifics in average latency. The return code, histogram data, and per-second throughput snapshot data was parsed and imported, but not used for any processing.

The results were aggregated as each client instance produced its own output. For the overall throughput, as each client tracked its own operations independently, the operations/sec across each client were summed in order to get an overall throughput per workload. Latency was aggregated as an average value of all client instances.

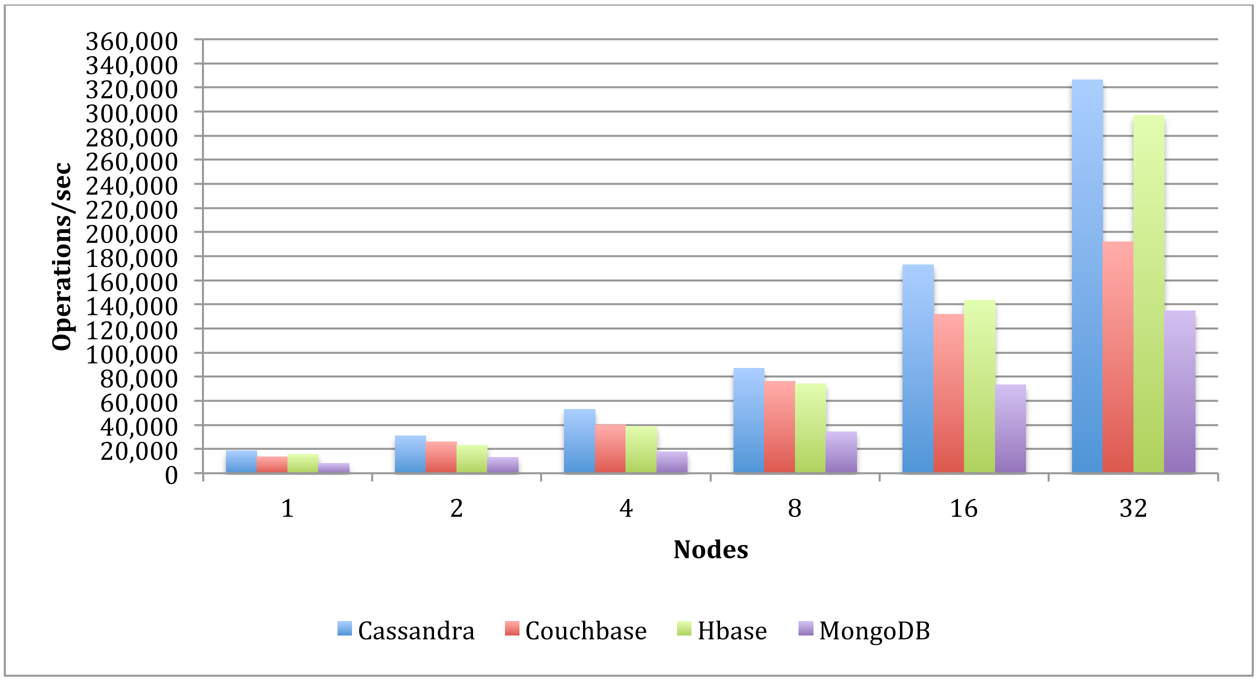
To work around instance performance variability each combination of database type and node count was run three times on different days. To combine the various test runs with the per-workload results above, the average was calculated for the three sets of results for each database type, node count, and workload.

Throughput by Workload

Each workload appears below with the throughput/operations-per-second (**more is better**) graphed vertically, the number of nodes used for the workload displayed horizontally, and a table with the result numbers following each graph.

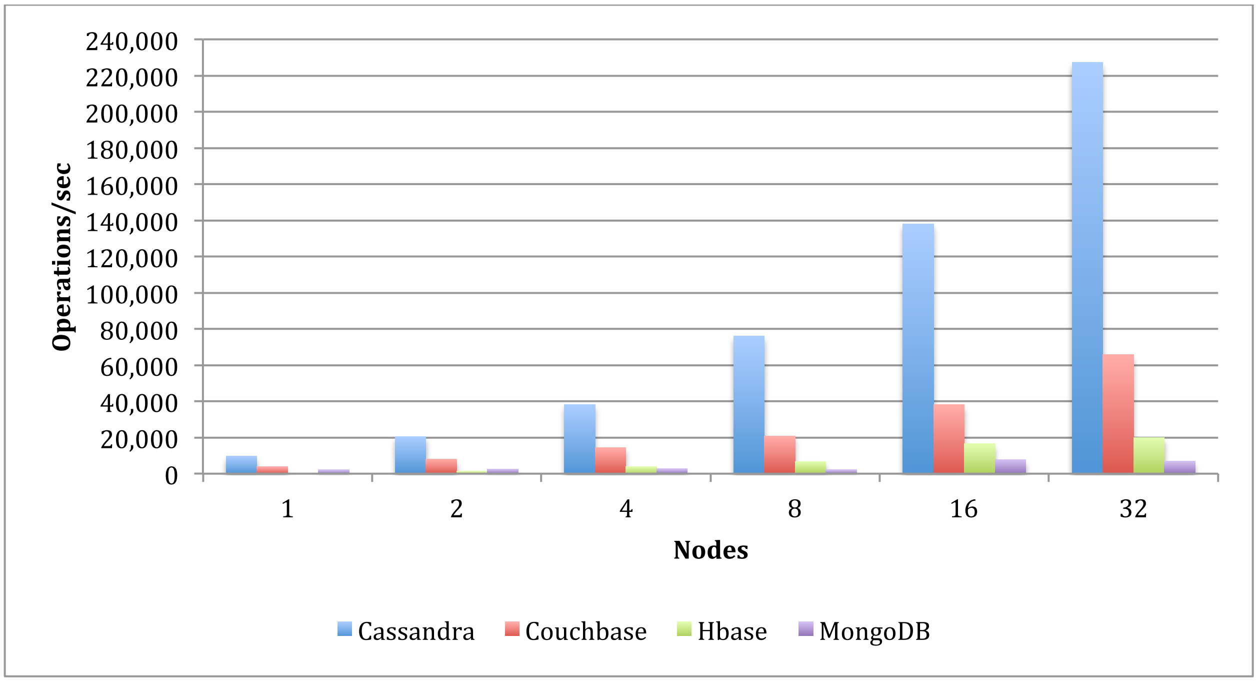
Load process

Bulk load was done ahead of each workload. We allowed each database to perform non-durable writes for this stage only to ingest as fast as possible.



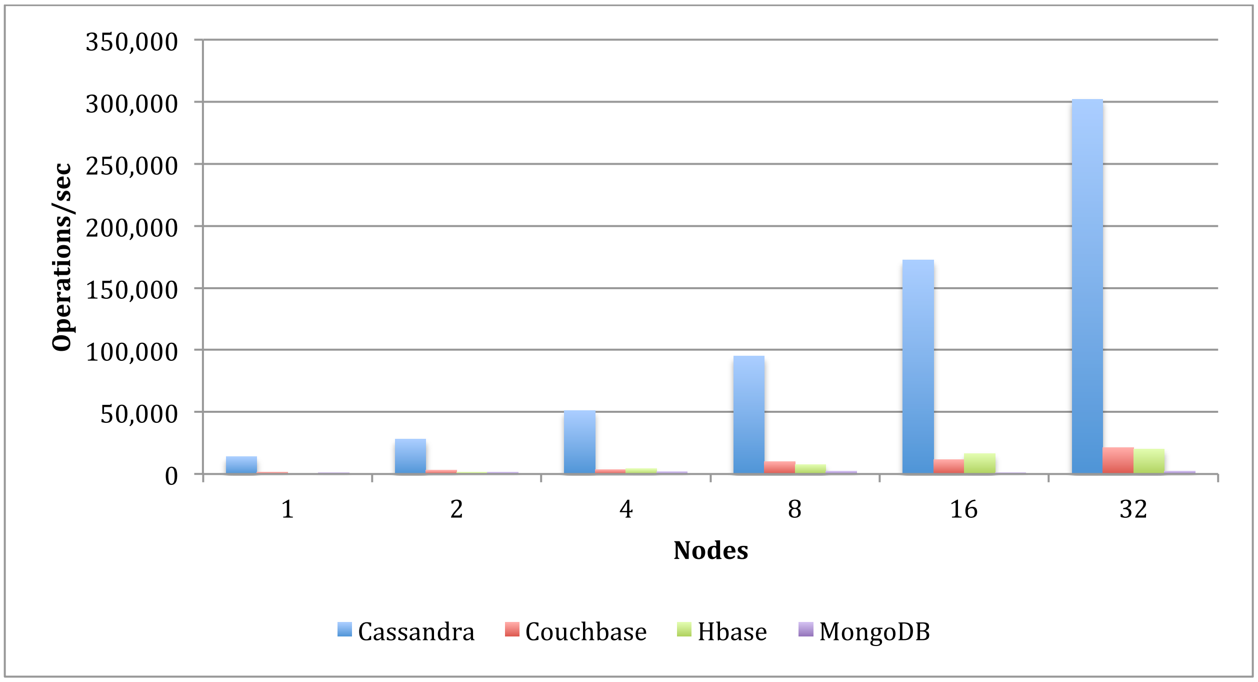
Nodes	Cassandra	Couchbase	HBase	MongoDB
1	18,683.43	13,761.12	15,617.98	8,368.44
2	31,144.24	26,140.82	23,373.93	13,462.51
4	53,067.62	40,063.34	38,991.82	18,038.49
8	86,924.94	76,504.40	74,405.64	34,305.30
16	173,001.20	131,887.99	143,553.41	73,335.62
32	326,427.07	192,204.94	296,857.36	134,968.87

Read-mostly Workload



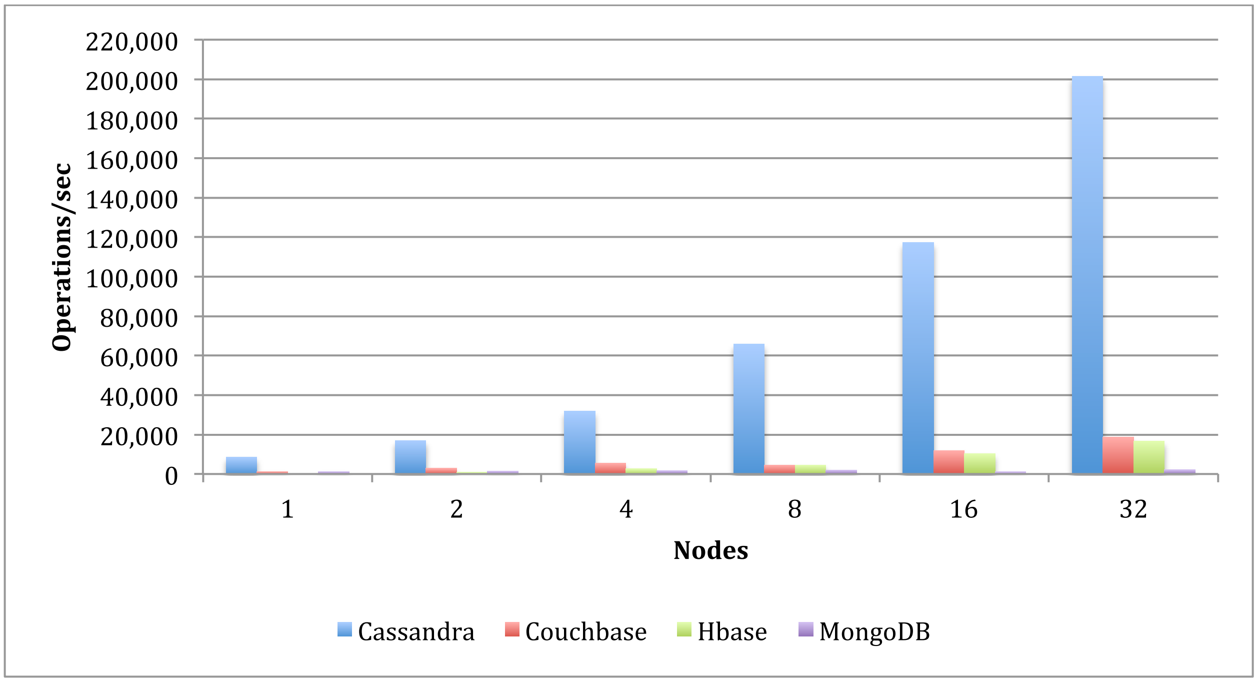
Nodes	Cassandra	Couchbase	HBase	MongoDB
1	9,768.76	3,950.90	428.12	2,149.08
2	20,511.51	8,147.78	1,381.06	2,588.04
4	38,235.67	14,503.08	3,955.03	2,752.40
8	76,169.72	20,904.02	6,817.14	2,165.17
16	138,141.79	38,084.77	16,542.11	7,782.36
32	227,292.80	65,978.17	20,020.73	6,983.82

Balanced Read/Write Mix



Nodes	Cassandra	Couchbase	HBase	MongoDB
1	13,929.58	1,554.14	527.47	1,278.81
2	28,078.26	2,985.28	1,503.09	1,441.32
4	51,111.84	3,755.28	4,175.8	1,801.06
8	95,005.27	10,138.80	7,725.94	2,195.92
16	172,668.48	11,761.31	16,381.78	1,230.96
32	302,181.72	21,375.02	20,177.71	2,335.14

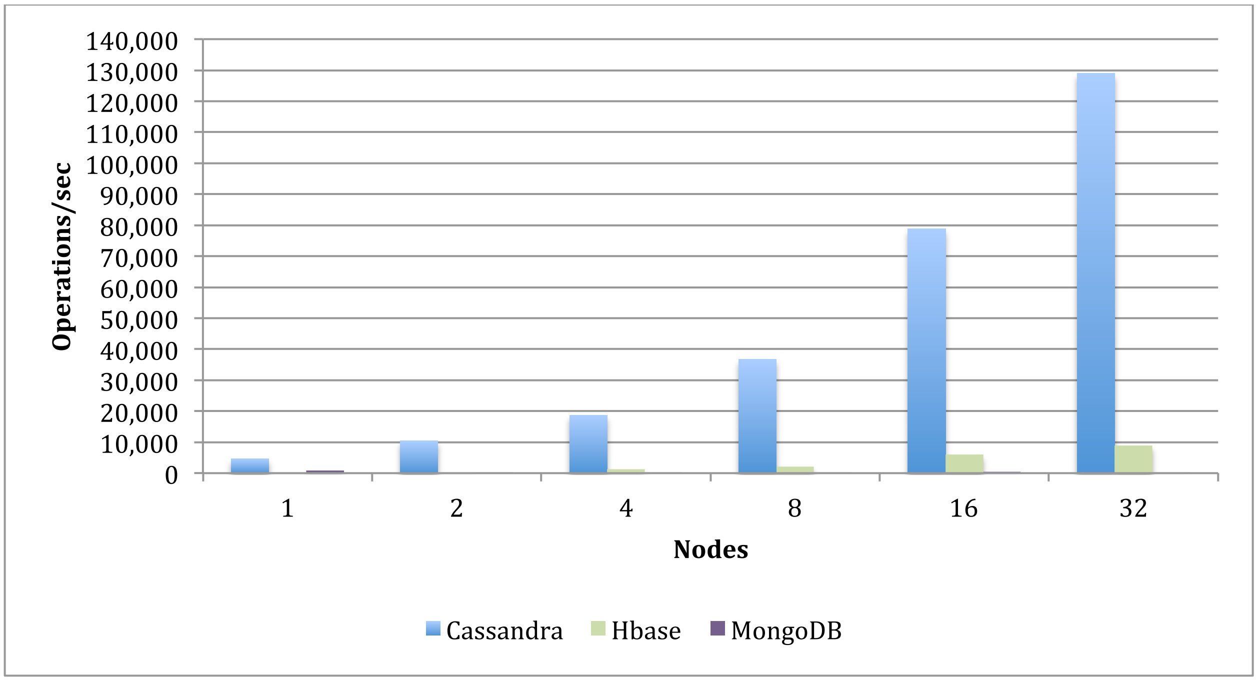
Read-Modify-Write Workload



Nodes	Cassandra	Couchbase	HBase	MongoDB
1	8,578.84	1,326.77	324.8	1,261.94
2	16,990.69	2,928.64	961.01	1,480.72
4	32,005.14	5,594.92	2,749.35	1,754.30
8	65,822.21	4,576.17	4,582.67	2,028.06
16	117,375.48	11,889.10	10,259.63	1,114.13
32	201,440.03	18,692.60	16,739.51	2,263.69

Mixed Operational and Analytical Workload

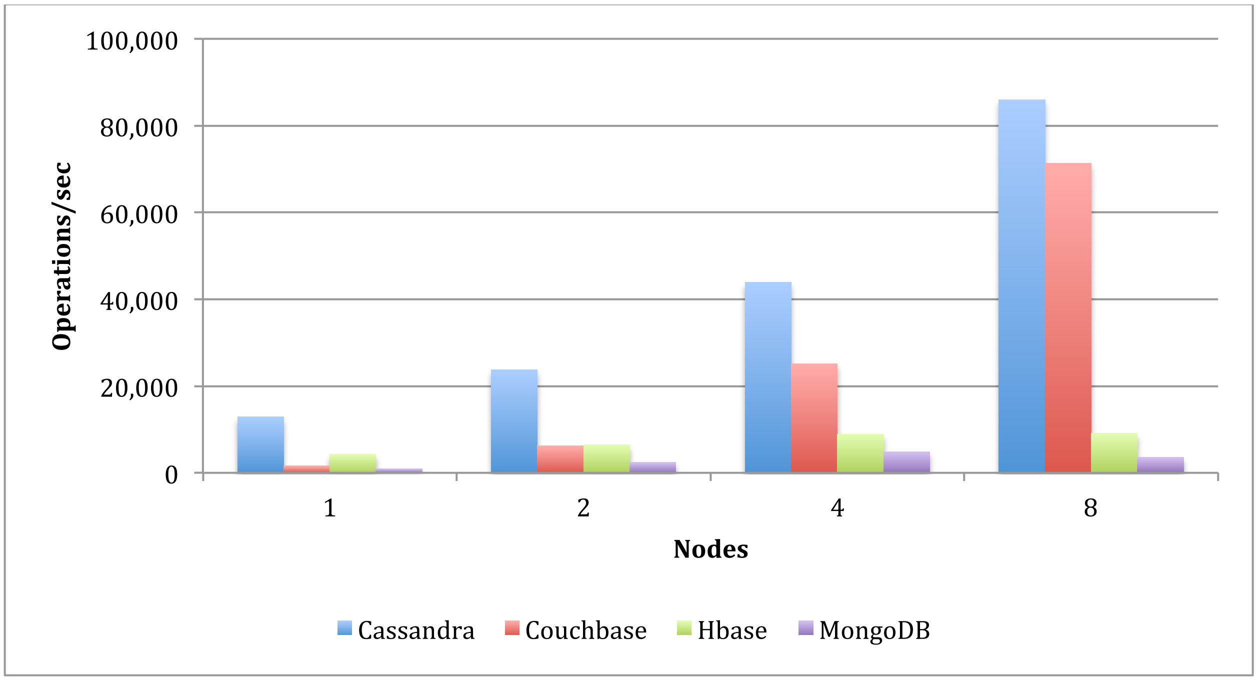
Note that Couchbase was eliminated from this test because it does not support scan operations (producing the error: “Range scan is not supported”).



Nodes	Cassandra	HBase	MongoDB
1	4,690.41	269.30	939.01
2	10,386.08	333.12	30.96
4	18,720.50	1,228.61	10.55
8	36,773.58	2,151.74	39.28
16	78,894.24	5,986.65	377.04
32	128,994.91	8,936.18	227.80

Insert-mostly Workload

Note that test runs on AWS against 16 and 32 nodes were not included. Despite repeated runs, technical issues were encountered that created inconsistent results for multiple databases involved in the test.

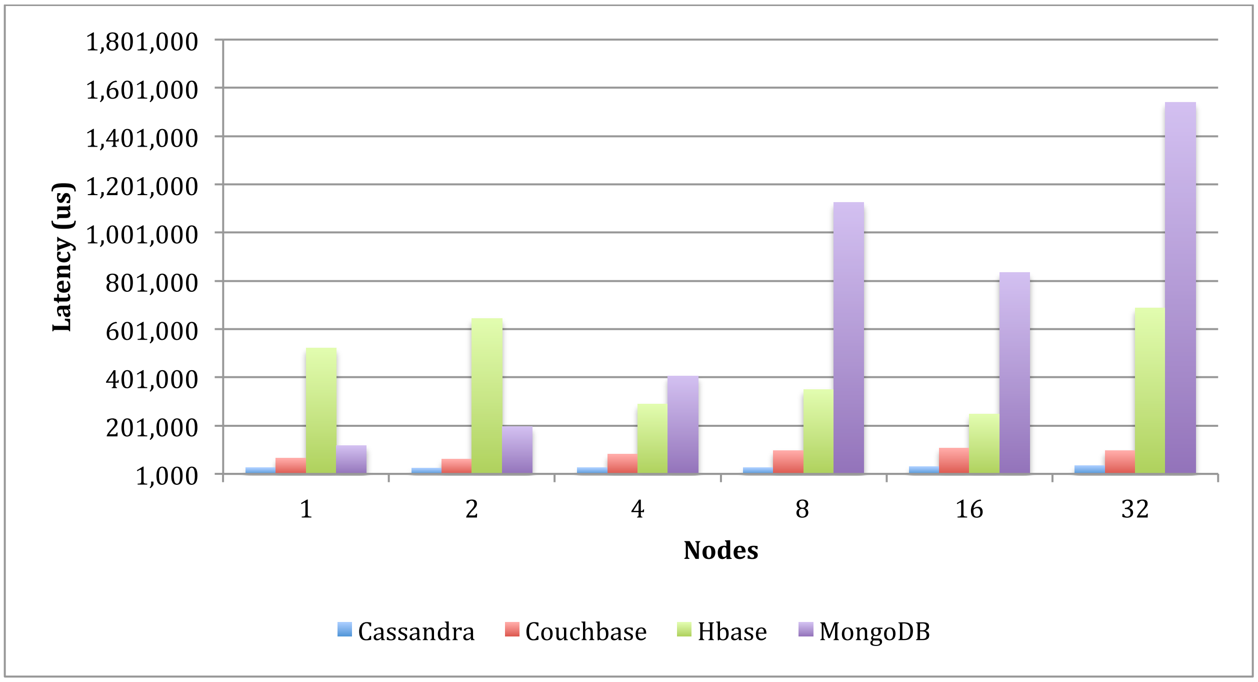


Nodes	Cassandra	Couchbase	HBase	MongoDB
1	12,879.19	1,624.63	4,274.86	984.89
2	23,779.90	6,241.02	6,417.96	2,446.06
4	43,945.00	25,185.76	8,913.46	4,853.85
8	85,974.82	71,307.78	9,101.47	3,641.98

Average Latency by Workload

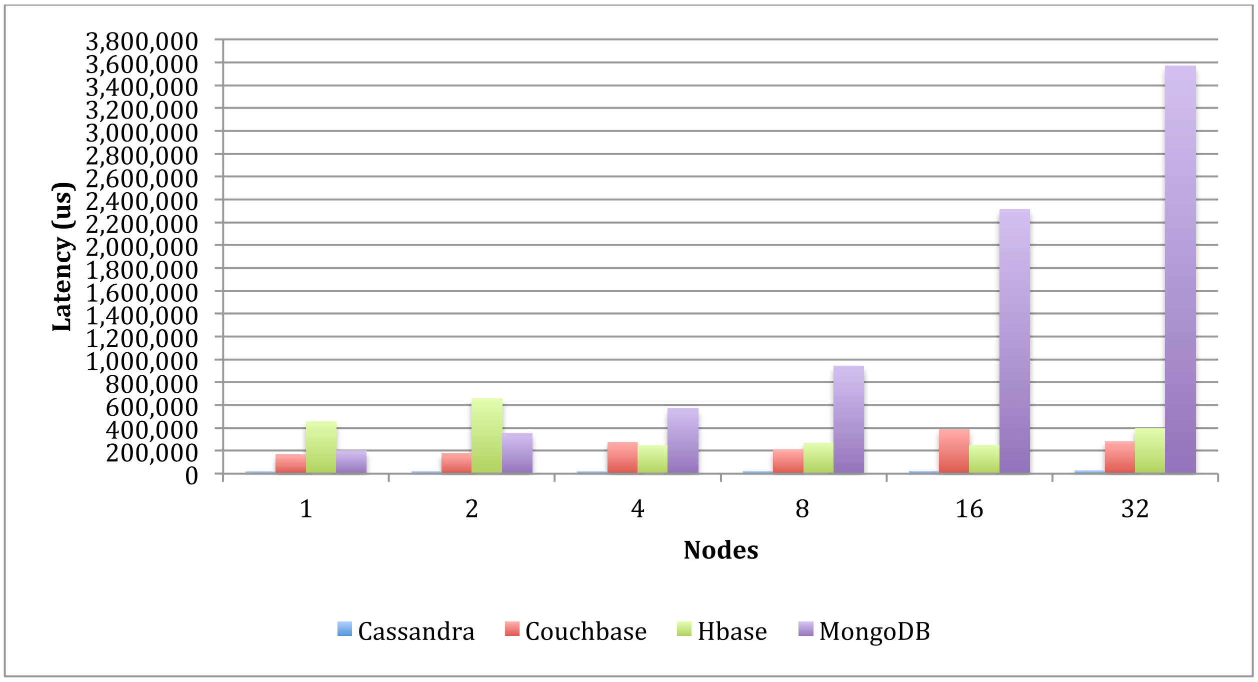
The following latency results (**less is better**) are displayed below in microseconds. In general, Cassandra exhibited the lowest and most consistent latency numbers for each test.

Read-mostly Workload



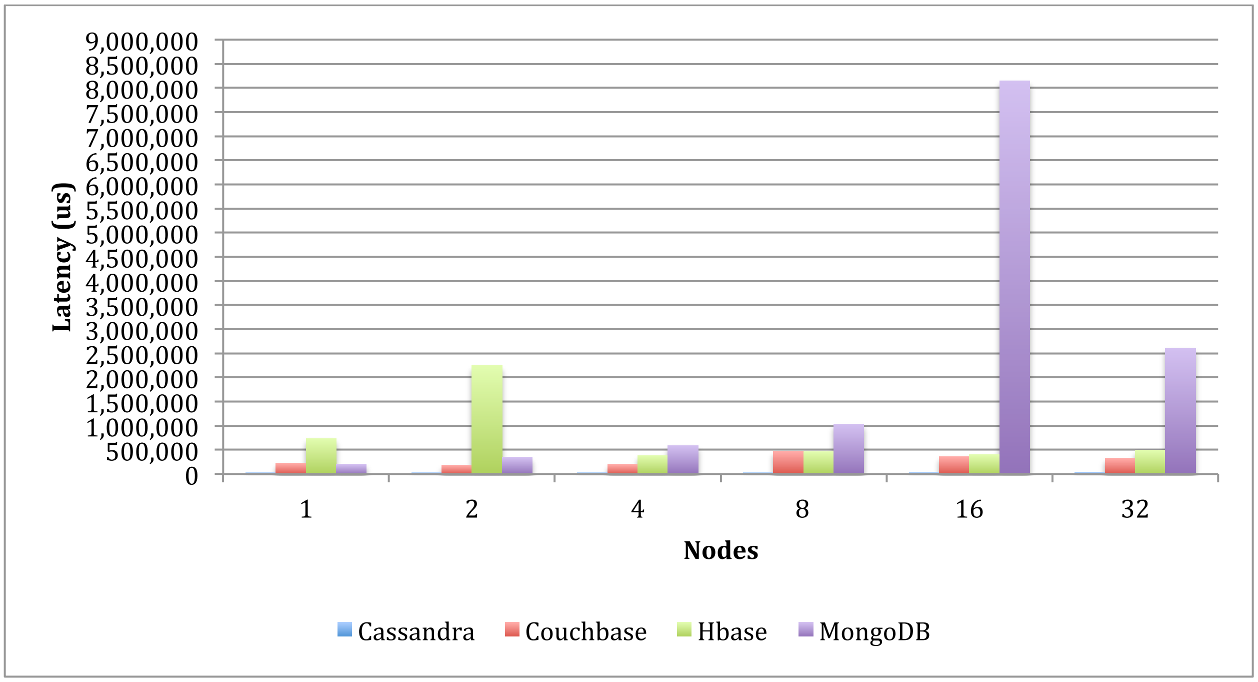
Nodes	Cassandra	Couchbase	HBase	MongoDB
1	26,312.23	66,166.99	522,835.62	119,139.25
2	25,170.28	62,683.11	645,326.42	197,806.93
4	26,816.47	82,561.51	291,440.74	407,118.75
8	26,973.70	98,514.50	350,383.54	1,126,244.08
16	30,918.79	109,061.54	248,318.31	835,206.29
32	36,322.71	97,200.37	689,000.88	1,540,875.22

Balanced Read/Write Mix Workload



Nodes	Cassandra	Couchbase	HBase	MongoDB
1	18,473.45	165,084.83	455,937.93	200,175.57
2	18,271.39	178,785.20	658,088.14	35,5291.28
4	19,966.76	273,068.34	246,699.88	575,302.93
8	21,525.42	212,504.65	268,893.02	942,975.54
16	23,730.58	385,804.73	250,156.02	2,311,434.45
32	26,842.97	282,499.64	393,333.39	3,571,231.75

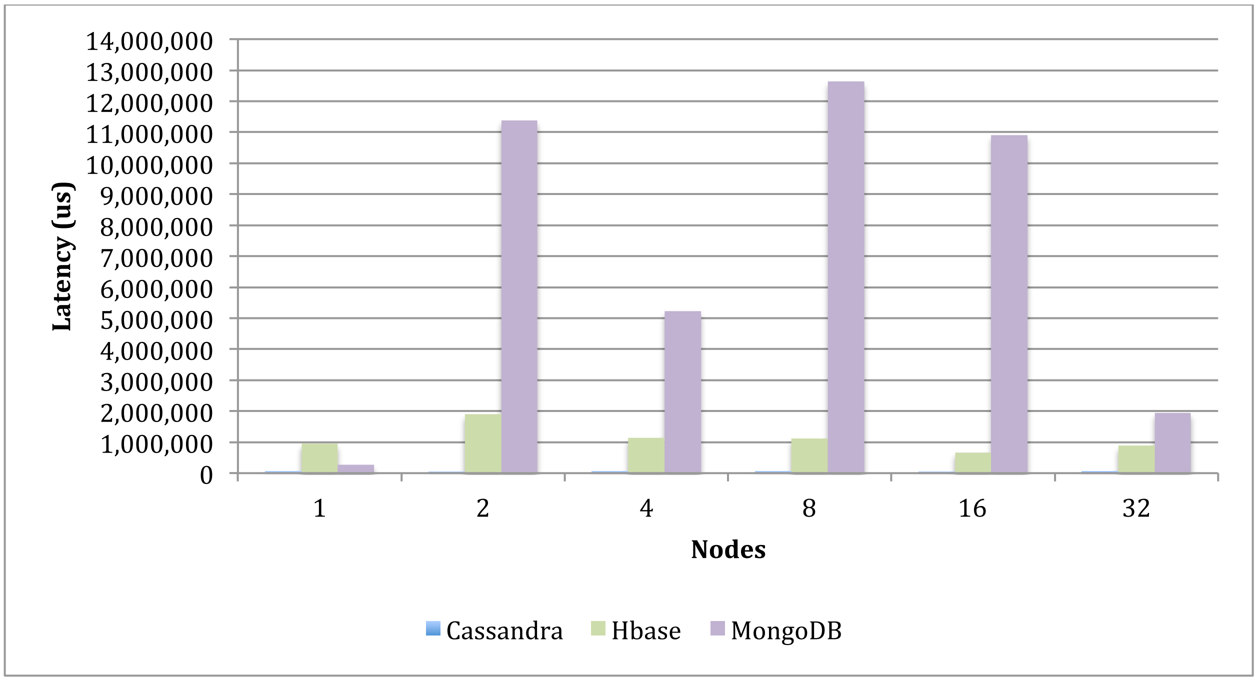
Read-Modify-Write Workload



Nodes	Cassandra	Couchbase	HBase	MongoDB
1	29,895.26	229,390.80	738,184.99	202,886.08
2	30,188.09	187,251.15	2,244,388.89	346,955.42
4	32,042.08	201,774.86	385,169.04	591,915.40
8	31,200.83	476,732.96	467,970.50	1,032,139.72
16	35,194.75	366,455.90	399,242.59	8,150,553.76
32	40,357.81	330,497.80	491,754.39	2,606,085.12

Mixed Operational and Analytical Workload

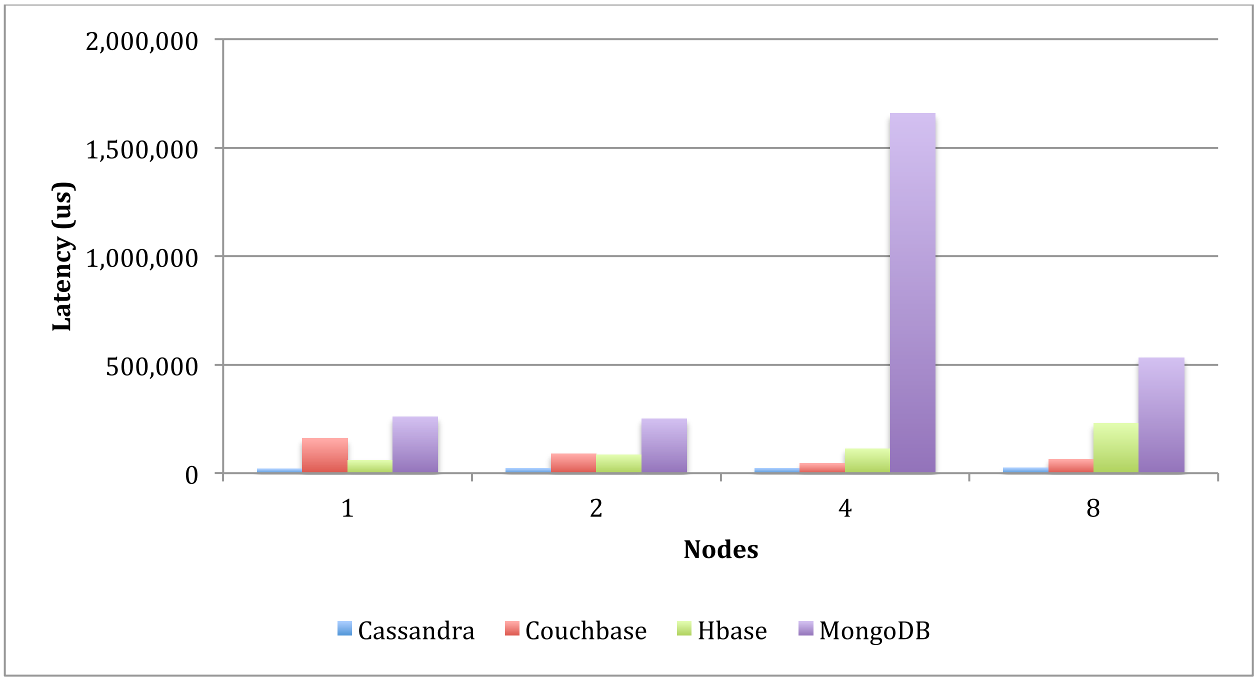
Note that Couchbase was eliminated from this test because it does not support scan operations (producing the error: “Range scan is not supported”).



Nodes	Cassandra	HBase	MongoDB
1	55,001.83	950,351.87	277,348.38
2	49,440.76	1,893,644.27	11,380,410.86
4	56,954.14	1,135,559.52	5,230,777.00
8	56,607.53	1,119,466.70	12,632,668.57
16	52,224.27	672,295.11	10,913,314.11
32	63,127.07	881,659.74	1,953,631.26

Insert-mostly Workload

Note that test runs in AWS against 16 and 32 nodes were not included due to a few technical issues that created inconsistent results.



Nodes	Cassandra	Couchbase	HBase	MongoDB
1	19,937.79	160,622.42	59,772.32	260,428.22
2	21,557.27	89,849.13	84,481.85	251,837.52
4	23,279.06	44,634.37	113,526.94	1,657,984.78
8	24,443.49	64,524.66	229,343.72	531,601.25

Database Idiosyncrasies and Difficulties Encountered

Couchbase was difficult to compile from source, and would initially appear to run as expected, but would then error out part way through the load process. It is not clear exactly what was different between the builds, but installing the pre-built package directly from Couchbase fixed this.

Additionally, Couchbase doesn't actually support a scan operation, with the client reporting: “Range scan is not supported”. Because of the random operation nature of the tests, the clients do get to execute some other types of operations, and report the results of those. But otherwise, the Couchbase results for the mixed operational/analytical workload should be discounted.

The Couchbase client seems to use more memory per thread and database connection, and at the 32-node level, the Java environment for YCSB had to be allowed to use more memory. Also, as previously noted above, Couchbase could not process the full data volume with the "value ejection" setting, so "full ejection" was used instead.

HBase and MongoDB both store ranges of data in discrete chunks, marked with start and end key values, which makes it difficult to evenly distribute data among their shards. Furthermore, the string-based range for the keys combined with YCSB's generation of keys without leading zeros virtually guarantees hotspots through part of the load process (i.e. separate clients inserting rows with keys “user1”, “user1000”, and “user10000000” are all placed into the same chunk). The technique used to pre-split the ranges tries to work around this as best as possible, and as data is loaded, the ranges do eventually even out.

At first each database was explicitly set to disable any on-disk compression of the data, but HBase seemed to encounter a problem where higher node counts (8 nodes and above, specifically) would run out of disk space on all of the shards at once during the load process. Turning on compression for HBase solved this.

MongoDB seemed unable to efficiently handle range scans in a sharded configuration. When running on a single node, range scan operations returned in a reasonable amount of time. However, any multi-node configuration resulted in range scan operations becoming CPU-bound and taking an extremely long time to return. Individual latency values varied, likely due to the key value of the start of the scan and how many other concurrent CPU-bound scans were taking place.

Configuration Details

These are the configuration files in effect for the databases, with the comments and default values removed in the interest of brevity:

Cassandra

cassandra.yaml

```
cluster_name: 'cassandra_{node_count}'
authenticator: org.apache.cassandra.auth.AllowAllAuthenticator
seed_provider:
        - class_name: org.apache.cassandra.locator.SimpleSeedProvider
        parameters:
        - seeds: "{master_IP}"
listen_address: {local_IP}
broadcast_rpc_address: {local_IP}
endpoint_snitch: SimpleSnitch
commitlog_sync: batch
commitlog_sync_batch_window_in_ms: 2
concurrent_writes: 64
```

Couchbase

couchbase/etc/couchdb/local.ini

```
[couchdb]
database_dir = /mnt/couchbase/data
view_index_dir = /mnt/couchbase/data
file_compression = none
[compactions]
default = [{dbfragmentation, “70%”}, {view_fragmentation, “60%”}]
```

couchbase/etc/couchdb/default.d/capi.ini

```
[couchdb]
database_dir = /mnt/couchbase/data
view_index_dir = /mnt/couchbase/data
```

MongoDB

No configuration files were used for MongoDB; instead the services were started with the parameters in the command line. For the first instance that has a configuration database:

```
bin/mongod --fork --logpath /var/log/mongodb/mongocfg.log --logappend --
dbpath /mnt/mongodb/mongocfg –configsvr
```

Each service was then started:

```
bin/mongod --fork --logpath /var/log/mongodb/mongodb.log --logappend --dbpath
/mnt/mongodb/mongodb --shardsvr --storageEngine wiredTiger

bin/mongos –fork --logpath /var/log/mongodb/mongos.log --logappend --configdb
$MONGO_MASTER
```

HBase

Hadoop core-site.xml

```
<?xml version="1.0" encoding=”UTF-8”?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
          <name>fs.defaultFS</name>
          <value>hdfs://{master_IP}</value>
  </property>
</configuration>
```

Hadoop hdfs-site.xml

```
<configuration>
  <property>
    <name>dfs.namenode.name.dir</name>
    <value>file:///mnt/hdfs/name</value>
  </property>
  <property>
    <name>dfs.datanode.data.dir</name>
    <value>file:///mnt/hdfs/data</value>
  </property>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
</configuration>
```

HBase hbase-site.xml

```
<configuration>
  <property>
    <name>hbase.zookeeper.quorum</name>
    <value>__MASTER__</value>
    <description>The directory shared by RegionServers.
    </description>
  </property>
  <property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/mnt/hdfs/zookeeper</value>
    <description>Property from ZooKeeper's config zoo.cfg.
    The directory where the snapshot is stored.
    </description>
  </property>
  <property>
    <name>hbase.rootdir</name>
    <value>hdfs://__MASTER__/hbase</value>
    <description>The directory shared by RegionServers.
    </description>
  </property>
  <property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
    <description>The mode the cluster will be in. Possible values are
      false: standalone and pseudo-distributed setups with managed Zookeeper
      true: fully-distributed with unmanaged Zookeeper Quorum (see hbase-env.sh)
    </description>
  </property>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
</configuration>
```