Benchmarking Top NoSQL Databases

Research · May 2016

DOI: 10.13140/RG.2.1.2276.6969

1 author:

Athiq Ahamed
Technische Universität Braunschweig
8 PUBLICATIONS **5** CITATIONS

SEE PROFILE

Benchmarking Top NoSQL Databases

Database as a Service
Institute of Computer Science TU Clausthal
Prof. Dr. Hartmann, Sven

Author:	Md Athiq Ur Raza Ahamed
Matriculation Number:	3094160
Degree:	Internet Technologies and Information Systems (M. Sc.)
Date:	Clausthal, on May 18, 2016

Abstract

Relational databases have been the dominant choice for decades. It is becoming difficult for organizing massive amount of data and for retrieval applications with relational databases. In this modern era of web 2.0 applications, NoSQL databases have become significant and performing better than relational databases when data is massive. NoSQL is significant since the modern application has to scale to levels that were unimaginable a decade ago, scaling is one such factor. Moreover, the companies require their applications are always available with high performance. The traditional databases fail to satisfy these requirements. With these requirements into consideration, there have been several NoSQL databases introduced. Choosing a particular NoSQL database for the application is difficult. Hence, in this paper, they presented the importance of the performance of NoSQL databases. They tested top three NoSQL databases with the standard benchmark to choose a particular database for the application. As a result, they presented that Cassandra outperforms the other two databases, MongoDB, and HBase. Additionally, each database is clearly explained and concluded with a discussion.

Keywords NoSQL, Cassandra, MongoDB, HBase

Table of contents

Tables

Figures

1 Introduction

In this modern era within a couple of decades, there has been a great revolution in the technology. As a result, an enormous amount of data has been developed. Relational databases have been the first choice always with its simple declarative language (SQL). On the contrary, massive data has been doubling, and it has been difficult for these traditional databases. Moreover, the modern applications require to be available always, scale and perform fast. On the other hand, RDBMS has a strict schema and require complex joins to retrieve data. In this modern era enormous amount of different types of data has been produced (structured, semi-structured, and unstructured). These requirements and type of data have replaced traditional databases to NoSQL databases. According to CAP theorem (Consistency, availability, and partition tolerance), it is not possible for databases to perform significantly well simultaneously in all the three areas. On the other hand, NoSQL databases are very flexible, according to the application requirements, trade off is made on availability or consistency with partition tolerance and vice versa. With Google's BigTable [8] introduction there has been several NoSQL databases such as Amazon's Dynamo [12], Apache's Cassandra [2], Apache's CouchDB [7], Hypertable [15], Voldemort [6], MongoDB [9], and HBase [13] are few examples that have been developed on the basis of CAP theorem.

These NoSQL databases have been categorized in one of the four important categories of NoSQL databases Key-values Stores, Column Family Stores, Document Stores and Graph Databases. Key-value stores, is the simplest and easy to implement, Voldemort is an example of this type. Column Family Stores are widely used, Google's BigTable being the inspiration for several other databases. Document stores are important for semi-structured data, MongoDB being a significant example. Finally, Graph database are largely used for application where data has to be stored as a graph like, (social networking), Neo4j [5] is an example.

With the development of several NoSQL databases, it is hard to choose a particular database for a certain application for the companies. The most important criteria for selecting a particular database is the performance of the database. From several research works, it has been proven that Cassandra, MongoDB, and HBase are the best performing NoSQL databases. Hence, in this paper they presented the importance of benchmarking to choose a database for certain application. As a result, they tested these top NoSQL databases on different workloads to suggest the certain database for a particular application. These evaluations could help

the IT professionals to understand the advantages and disadvantages of NoSQL databases. Additionally these three databases are elaborately explained in the context of performance.

The rest of the paper is paper is organized as follows. A brief introduction to these top three NoSQL databases. Then, the importance of benchmarks and configuration is explained. Further the results of these three databases on different workloads are illustrated. Finally, the paper is concluded with a discussion.

2 Top NoSQL databases

According to several research works and analysis in NoSQL databases, there are three top NoSQL databases as follows.

2.1 Cassandra

With a new shift in data management applications occurred with web 2.0 applications. The data has to be managed in a massively scalable, high performance, and no failure nature. Apache Cassandra [11, 2] is a NoSQL database for massive scalability, high availability with no single point of failure without affecting the performance to handle significant amounts of data. The architecture of Cassandra as shown in the Figure 2.1, has replaced the legacy master-slave with peer-to-peer distributed architecture. The peer-to-peer distributed architecture is straightforward and easy to setup and maintains large-scale data. The traditional master node concept is replaced with this new architecture were all nodes are the same and communicate with each other using a gossip protocol. Since Cassandra has no single point of failure, it is always available. Important features of Cassandra are listed below.

- Elastic scalability: As data increases, it is elastically scalable by adding more systems.
- Decentralized and durable: Cassandra has no single point of failure, and data is distributed and replicated. As a node in a cluster goes down, a copy of that node's data is already available on another machine.
- Linearly scalable: It is linearly scalable by increasing the number of nodes. As a result, throughput increases with no interruption to applications or no downtime.
- Supports any data type: Cassandra supports any data type structured, semi-structured, and unstructured
- Column-oriented database: Cassandra's data model offers the convenience of column indexes and also supports Atomicity, Consistency, Isolation, and Durability (ACID).

- Cassandra Query Language: A simple yet powerful query language with which one can access Cassandra through any node as shown below.

```
CREATE TABLE sample(
   sample_id int PRIMARY KEY,
   sample_name text,
   sample_city text,
   sample_phone varint
   );
```

- Distributing and replicating data: In Cassandra data is automatically distributed across all nodes that participate in the database cluster in a randomized or ordered fashion. As a result, it is always available. Replication is easier when compared to other NoSQL databases; the user has to indicate the number of copies and the rest is taken care by Cassandra for replicating it automatically.

- Simple read and write: Cassandra supplies a true, "location independent" architecture where any node in the Cassandra can be read or written. When writing the data, it is first written to the commit log, for durability and safety purpose. From the commit log, it is written into the in-memory structure called memtable, a memory-resident data structure which then flushes the data into a disk structure called a sstable, sorted strings table, which is automatically replicated throughout the cluster. For reading the data, it gets values from the memtable and uses bloom filter a quick, non-deterministic algorithms for testing and finding whether the element belongs to the appropriate sstable.

- Data model: Column-oriented database with keyspace, an outermost container for data and contains a list of column families. Column families are the collection of rows with primary and secondary indexing features.

```
CREATE KEYSPACE Keyspacename
WITH replication = {'class':'example','replication_factor':4};
```

Use case: Typical use of Cassandra database is the E-commerce travel portals, and it does not suit when the data involves dynamic queries on columns.

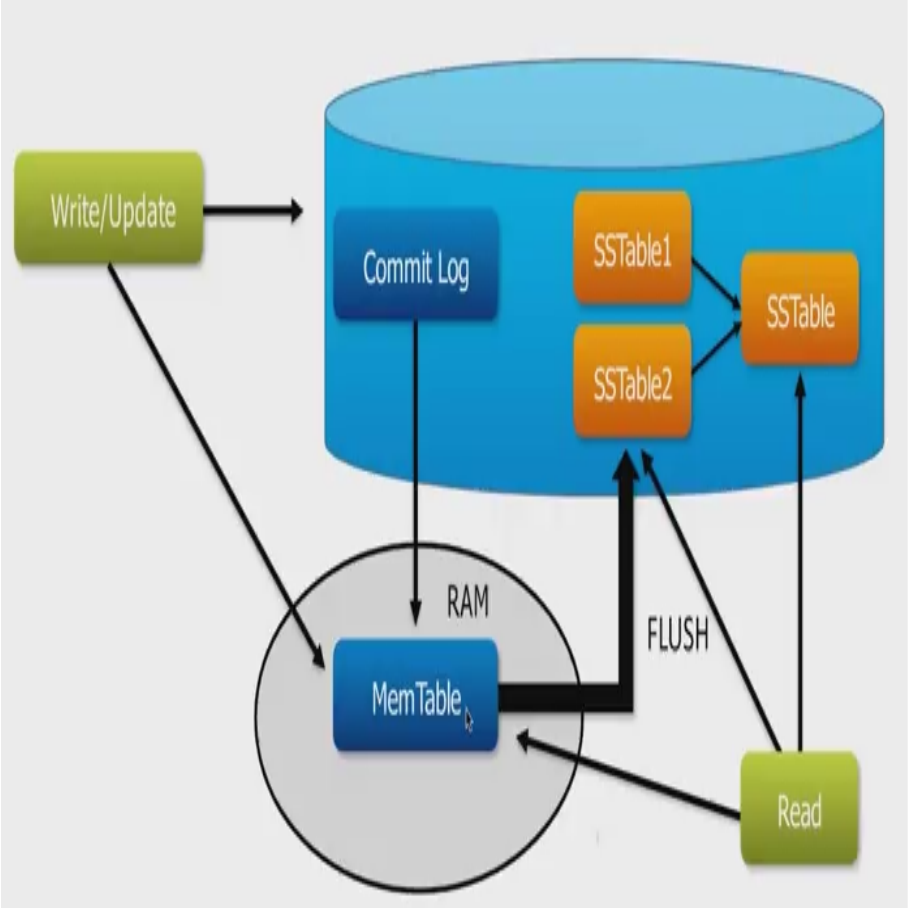

Figure 2.1: Cassandra architecture [1]

2.2 HBase

HBase [13, 16] is a column-oriented NoSQL database management system build on top of the Hadoop Distributed File System (HDFS). HBase is best suited for sparse data sets were the data is broken into blocks and distributed across the cluster and uses a map and reduce functions to process it. Thus, large data sets are broken into smaller subsets and dealt in a faster fashion providing scalability. The data in the HBase could be sparse, distributed, multi-dimensional and consistent. Since HDFS does sequential searches and does not support fast individual record lookups. Hence, HBase is used for fast lookups and uses Hash tables to overcome the sequential search problem thus providing high performance. The HBase tables contain column families that are a logical grouping of columns, with time-stamped versions, where a key identifies each column value. Hence, each row has an implicit primary key, and the row key sorts the table.

HBase follows a master-slave type of architecture as shown in the Figure 2.2 and the data in HBase is stored in HDFS files. The architecture consists of a region server that is responsible for data reads and writes. HBase master is responsible for operations such as creating, deleting, and updating tables. HBase uses Zookeeper as a distributed coordination service for coordinating and maintaining the servers that are alive and available. Whenever the server fails, Zookeeper provides a notification. For reading and writing in HBase a catalog table is maintained called a meta table. Meta table, similar to a b-tree holds the location of the regions in the cluster. The table consists of the key-value pair. When the client wants to read the data first it discovers the Region Server that holds meta table using the information from Zookeeper. Then the client will query the meta server with respect to the row-key. Then it caches this information corresponding to the region server. For further reads, the client uses this cached information. For writing, the data is first written to write-ahead log (WAL). WAL is useful when the server crashes. After WAL the data is written into MemStore, it stores in-memory as sorted key-value and after accumulating enough data then it is stored in a HFile (HDFS). This write is sequential and very fast, avoiding moving disk drive head. Main benefits of HBase are:

- Highly consistent
- Automatic scaling
- Built in recovery/automatic failure support
- Integrated with Hadoop
- Real-time queries using bloom filters and blocked caches

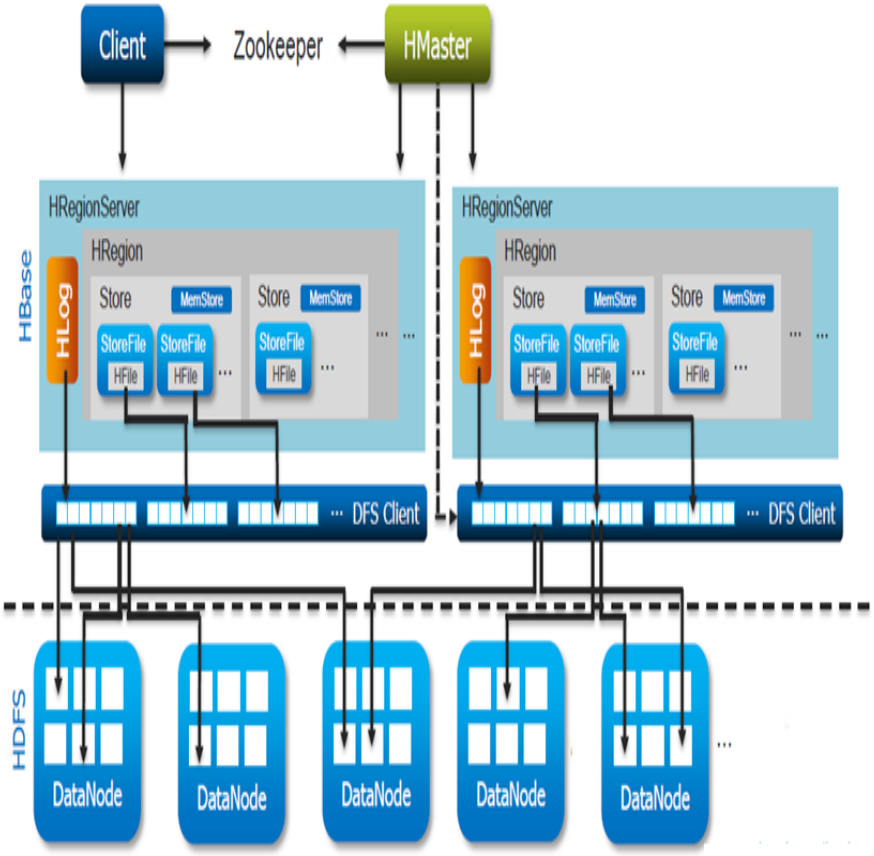

Figure 2.2: HBase architecture [3]

- random read/write capabilities
- An easy Java API for the client.

Use case: Used wherever there is an enormous amount of data, companies such as Facebook, Twitter, Yahoo use HBase. One shouldn't use HBase when the application requires SQL commands.

2.3 MongoDB

MongoDB [14, 9] is a NoSQL database that is a document-oriented. Their documents are JSON-alike and without a fixed schema (Flexible Data Model). MongoDB is highly available with high performance and ease of scalability. MongoDB provides high availability with replica sets. The tables are replaced by collection.

The collection is the document group of MongoDB. The document with a dynamic schema has a set of key-value pairs. Important features of MongoDB are the auto-sharding, fast in-place updates, easy to scale-out, and provides rich queries. The architecture of MongoDB as shown in the Figure 2.3, is a combination of significant capabilities of relational databases and NoSQL technologies. They are highly consistent, flexible, scalable, and has an expressive query language. It has a Javascript shell called mongo that is the client for connecting the mongod or its instance. It has a mongos that is the daemon which is a query router for the client and cluster. The most important aspect is the configdb that contains the metadata for the clusters. With such strong consistency, all the applications read data immediately once written in the database. With replicas, the read operation takes place with default primary copy but optionally read from secondary copies. The mongo receives the specific queries from applications and uses configdb metadata to route it to specific mongod instance. With hash-based sharding, documents are uniformly distributed using MD5 HASH of shard key. For write operations, MongoDB is acid compliant, and many fields are written in a single operation. That is mongos directly writes the data from a specific application to the shards. They follow a similar process of config server for metadata and route the specific write operation and additionally records it in oplog (operation log). Moreover, oplog is reproducible. Key benefits of MongoDB are:

- Document-oriented database
- High performance and automatic scaling
- High consistency and partition tolerant
- Replication and failover for high availability
- Low latency
- Flexible indexing
- Rich fast querying
- Aggregations in database

```
{
   id: ObjectId(090090909f)
   title: 'Sample',
   description: 'Sample for MongoDB',
   by: 'Athiq',
```

```
    url: 'http://xxxx',
    tags: ['MongoDB','NoSQL'],
    likes: 150,
    comments: [
       {
          user:'user100',
          message: 'Hello',
          dateCreated: new Date(2016,2,21,5,16),
          like: 0
       },
       {
          user:'user101',
          .....
       }
    ]
}
```

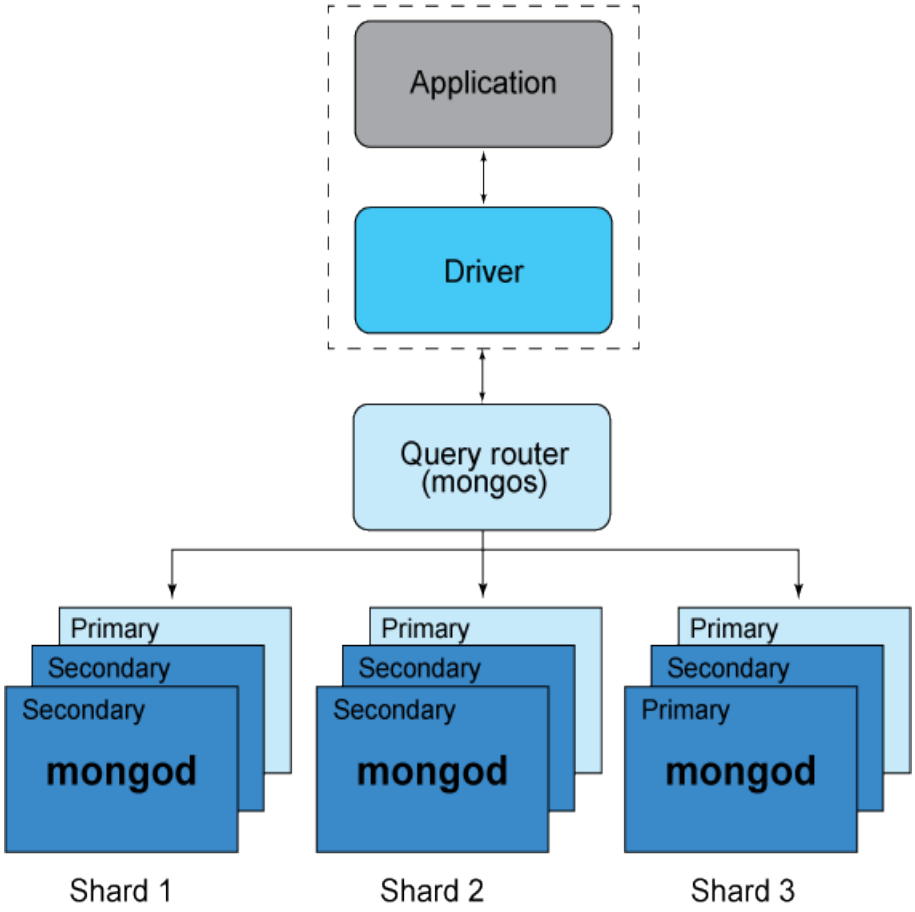

Figure 2.3: MongoDB architecture [4]

3 Benchmark configuration and results summary

This chapter is adopted from the original work [10]. "Amazon Web Services EC2 instances, an industry-standard platform for hosting horizontally scalable services" was used to run the tests on the three databases. Spot instances were used to ensure cost efficiency. The test ran on m1.xlarge size instances (15 GB RAM and 4 CPU cores). The instance used local storage for performance up to 4 instances could be added to the m1.xlarge. All booted into a single 1.7TB RAID-1 volume Instance is a Ubuntu 12.04 LTS AMI's. Oracle Java 1.6 installed. An automatic script for driving the benchmark process for configuration, termination of EC2 instances, and calculation of workload parameters. Benchmark workload is the YCSB (The Yahoo! Cloud Serving Benchmark), an open standard for comparative performance evaluation of different data stores. Client AMI's had the latest git checkout of YCSB. Finally, the AMI's built using zipfian_constant

3.1 Cassandra configuration

Apache Cassandra 1.1.6 was used. On start up node ID, and an initial token value was received from the head node, (node ID) * 2127/(node count) was used to calculate initial token for distributing the token range. The first node was declared as the default contact point and also seed provider, as a result, the hosts can find each other

3.2 HBase configuration

Stable Hadoop (1.1.1) was used. On start up node ID was received from the head node. First node by default is the master node and runs as the Hadoop NameNode. The first node is also the Zookeeper. The other nodes are the DataNodes. The driver script was used once the cluster is ready and created its objects. Example for a single data node in the cluster as follows.

```
create 'usertable', 'data', {{NUMREGIONS => (node count), SPLITALGO =>
'HexStringSplit'}}
```

3.3 MongoDB configuration

The MongoDB AMI (2.2.2) was used, additional scripts were written to start the mongos shard process since the package for installation had only init script. On start up, node ID was received from the head node, first node by default declared itself as the configuration server. As a result mongod instance has started all nodes then ran "sh.addShard()" for starting the MongoDB.

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

The above code was used to enable sharding and to pre-distribute the chunks by ID on each shard following command was ran.

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

3.4 Tested workloads

"The following workloads were included in the benchmark,

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

2. Read/write combination, based on YCSB's workload A: 50% read to 50% update ratio

3. Write-mostly workload: 99% update to 1% read

4. Read/scan combination: 47% read, 47% scan, 6% update

5. Read/write combination with scans: 25% read, 25% scan, 25% update, 25% insert

6. Read latest workload, based on YCSB workload D: 95% read to 5% insert

7. Read-modify-write, based on YCSB workload F: 50% read to 50% read-modify-write "

Table 3.1: The calculations for testing parameters work out to these values

Data Nodes	Client Nodes	Total Records	Records /Client	Total Threads	Threads /Client
1	1	15M	15M	128	128
2	1	30M	30M	256	256
4	2	60M	30M	512	256
8	3	120M	40M	1024	341
16	6	240M	40M	2048	341
32	11	480M	43.6M	4096	372

3.4.1 Testing parameters

In all the testing cases the fieldcount was set to default 20 so that it produces 2 KB records. In all workloads except workload6 the request distribution was by default "zipfian". For workload6 it was set to "latest". The workloads scans were limited to 100 that involve scans. The execution time was limited to 1 hour that involves scans. For others no time limit by setting an operation count of 900,000. Driver script was used to calculate other parameters. Record count was calculated based on requested data instances. Benchmark for every 3 data instance had a client instance. Insert count was calculated by distributing record count from different client instances. The whole test process also targeted 128 client threads for every data instance and distributed them across different client instances. "The calculations for testing parameters work out to these values" as show in the Table 3.1.

4 Results

Results Summary: For all the test results Cassandra outperformed the other two HBase and MongoDB for all node configurations (1 to 32 nodes) from throughput perspective. For latency, Cassandra had a lowest value than the other two databases for all the workloads in the testing process. Thus Cassandra proved to be the best performing database. All the results and table were adopted from [10].

4.1 Throughput results

Throughput/operations-per-second as shown in the the Figures 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8 and in the Tables 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8.

Cassandra outperformed the other two databases HBase and MongoDB. Moreover, MongoDB was not able to scale effectively for 32 nodes. "MongoDB also produced errors until the thread count was reduced to 20".

4.2 Latency results

The below results as show in the Figures 4.10, 4.11, 4.12 (less is better) and in the Tables 4.10, 4.11, 4.12. Cassandra had the lowest latency.

Table 4.1: Load process

Shards	**Cassandra**	**HBase**	**MongoDB**	**%HBase**	**%MongoDB**
1	6955.33	3126.29	4101.73	122.48%	69.57%
2	11112.07	4379.17	3572.88	153.75%	211.01%
4	19965.86	6528.82	4630.18	205.81%	331.21%
8	27131.40	13067.41	5307.89	107.63%	411.15%
16	69922.85	23250.44	6767.80	200.74%	933.17%
32	120918.17	42761.57	7777.36	182.77%	1454.75%

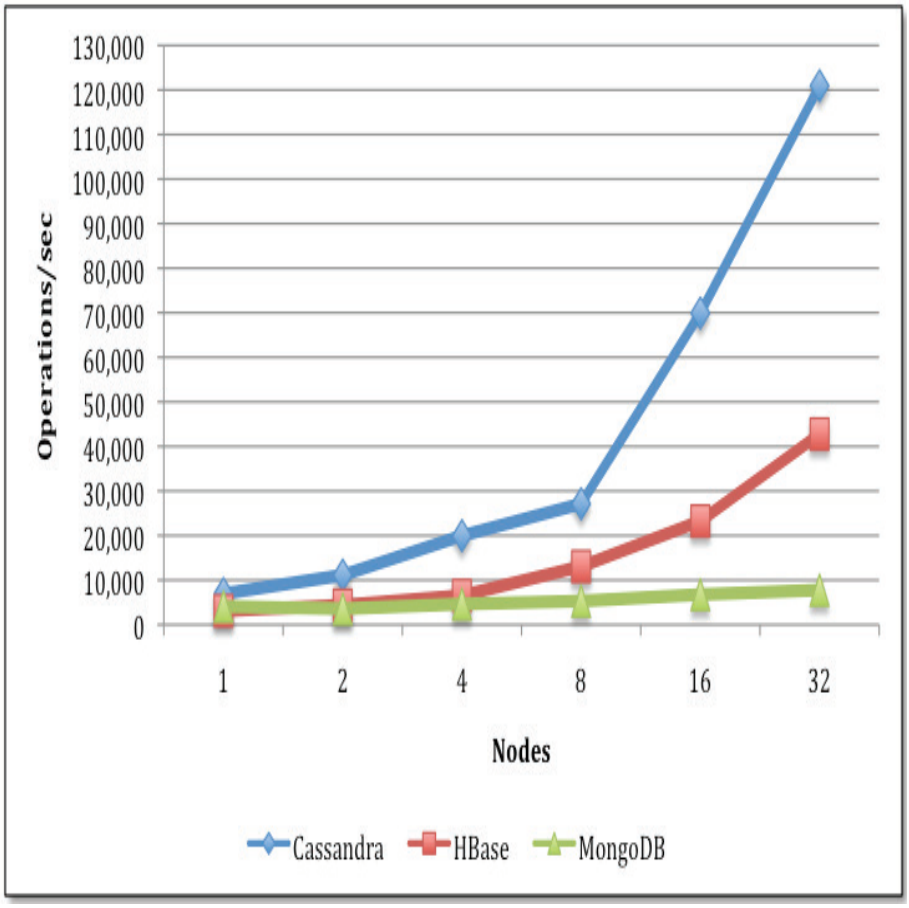

Figure 4.1: Load process

Table 4.2: Read-mostly workload

Shards	Cassandra	HBase	MongoDB	%HBase	%MongoDB
1	624.31	206.06	115.47	202.97%	440.67%
2	1091.83	318.18	183.03	243.15%	496.53%
4	2608.90	433.89	298.69	501.28%	773.45%
8	5000.98	784.60	746.26	537.39%	570.14%
16	10016.62	1467.96	1006.41	582.35%	895.28%
32	19277.25	2969.33	1575.21	549.21%	1123.79%

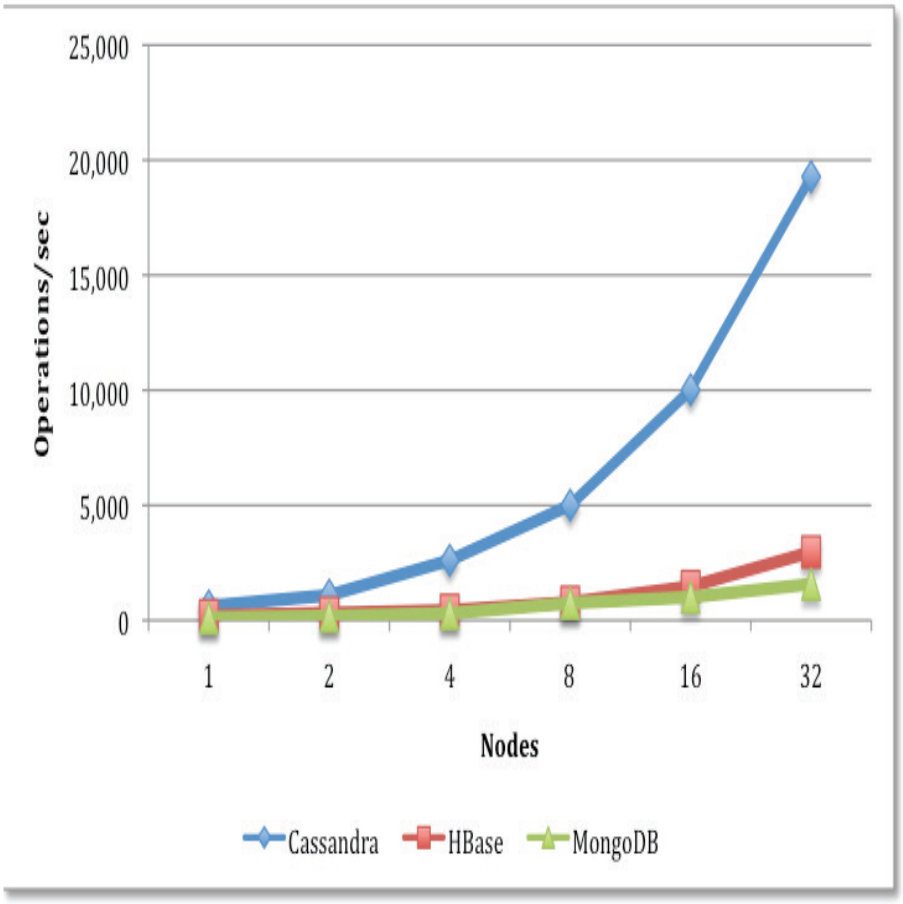

Figure 4.2: Read-mostly workload

Table 4.3: Read/Write mix workload

Shards	Cassandra	HBase	MongoDB	%HBase	%MongoDB
1	1118.90	448.86	122.80	149.28%	811.16%
2	1781.36	762.83	195.85	133.52%	809.55%
4	4543.78	1249.72	268.54	263.58%	1592.03%
8	8827.11	2542.40	869.21	247.20%	915.53%
16	18454.53	6101.27	1257.67	202.47%	1367.36%
32	33767.99	8477.16	1807.26	298.34%	1768.46%

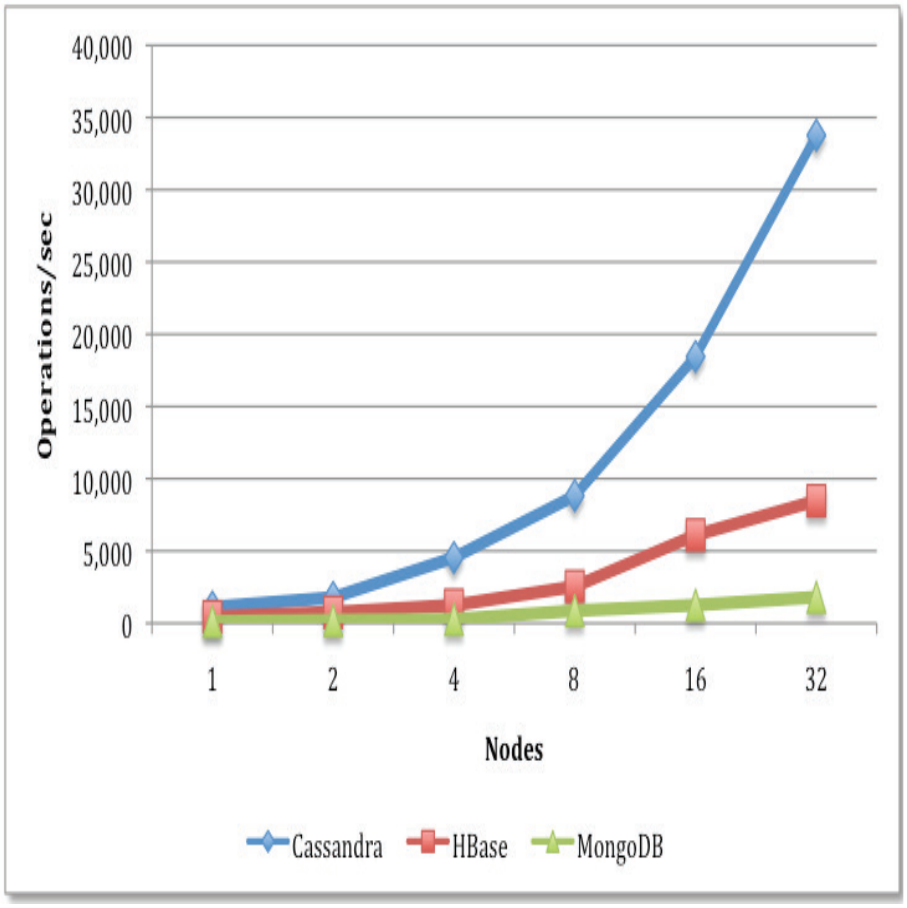

Figure 4.3: Read/Write mix workload

Table 4.4: Write-mostly workload

Shards	Cassandra	HBase	MongoDB	%HBase	%MongoDB
1	20692.26	5548.53	123.18	272.93%	16698.39%
2	23032.20	8358.53	185.97	175.55%	12284.90%
4	44253.28	14493.68	277.38	205.33%	15854.03%
8	51718.85	30328.45	865.66	70.53%	5874.50%
16	117909.48	46251.65	1148.93	154.93%	10162.55%
32	215271.59	75008.83	1715.92	186.99%	12445.55%

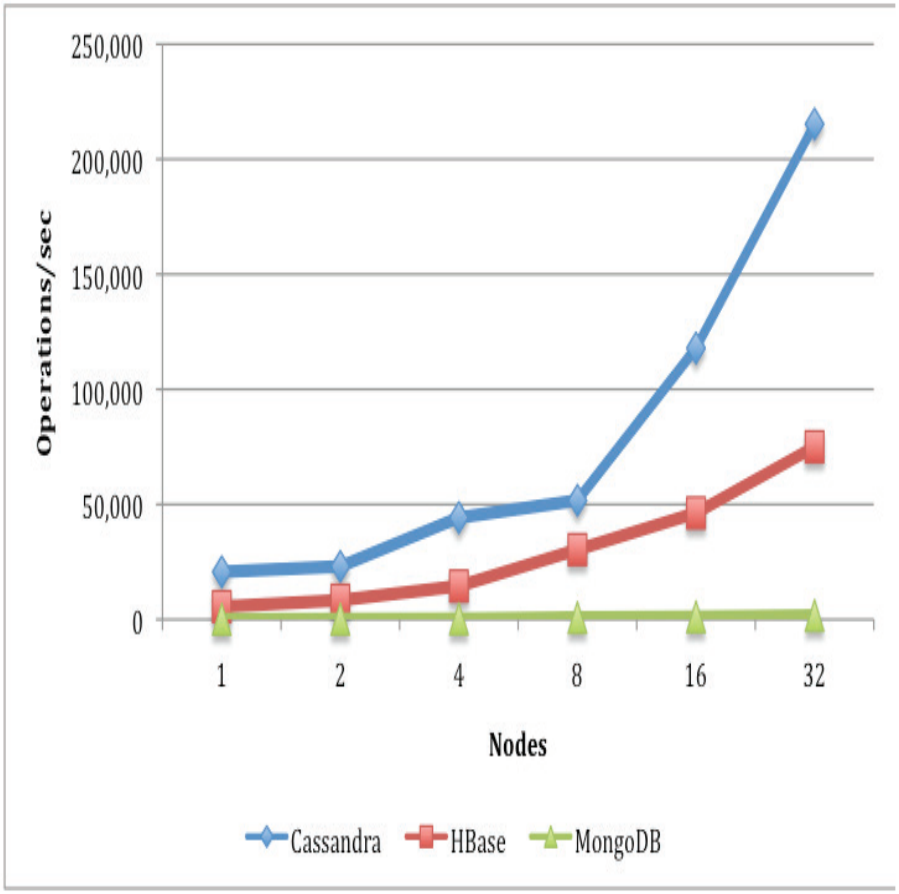

Figure 4.4: Write-mostly workload

Table 4.5: Read/Scan mix workload

Shards	Cassandra	HBase	MongoDB	%HBase	%MongoDB
1	149.09	80.19	1.61	85.92%	9160.25%
2	256.95	144.90	3.33	77.33%	7616.22%
4	555.20	225.41	5.06	146.31%	10872.33%
8	739.59	430.03	17.39	71.99%	4152.96%
16	1443.29	886.22	23.35	62.86%	6081.11%
32	2495.07	1103.07	47.71	126.19%	5129.66%

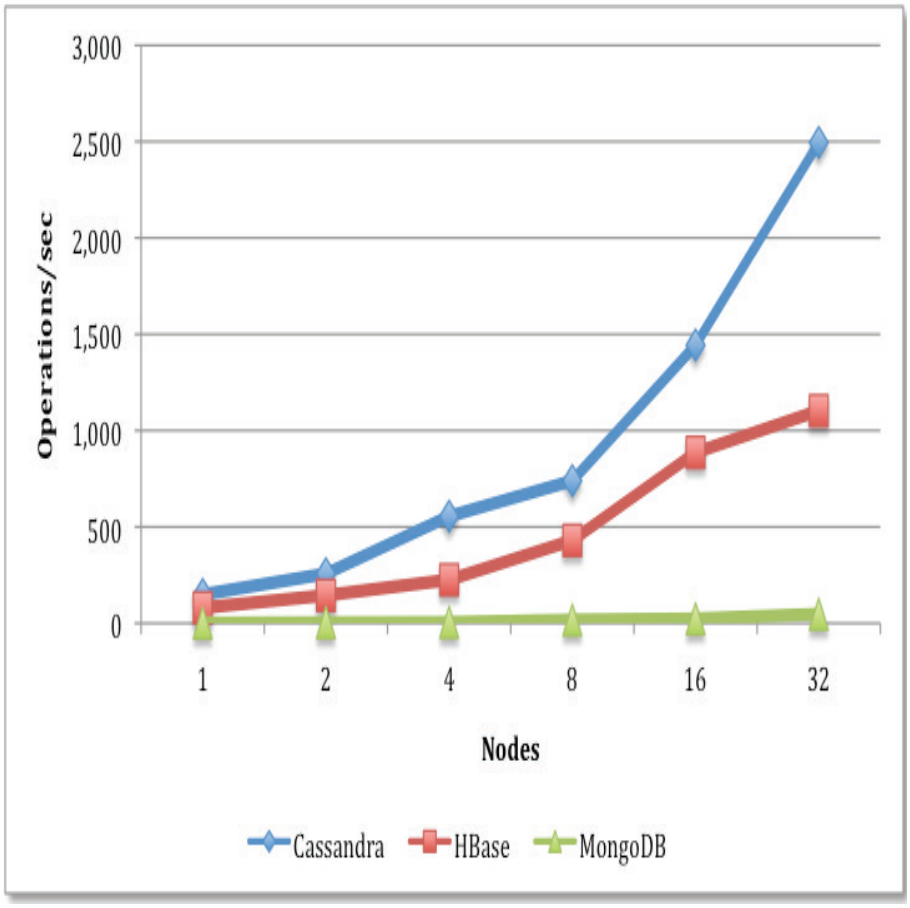

Figure 4.5: Read/Scan mix workload

Table 4.6: Write/Scan mix workload

Shards	Cassandra	HBase	MongoDB	%HBase	%MongoDB
1	261.75	150.27	3.49	74.19%	7400.00%
2	437.15	282.82	6.67	54.57%	6453.97%
4	974.57	447.27	10.19	117.89%	9463.98%
8	1522.25	853.03	39.74	78.45%	3730.52%
16	2568.09	1698.38	57.20	51.21%	4389.67%
32	4668.04	2213.02	157.30	110.94%	2867.60%

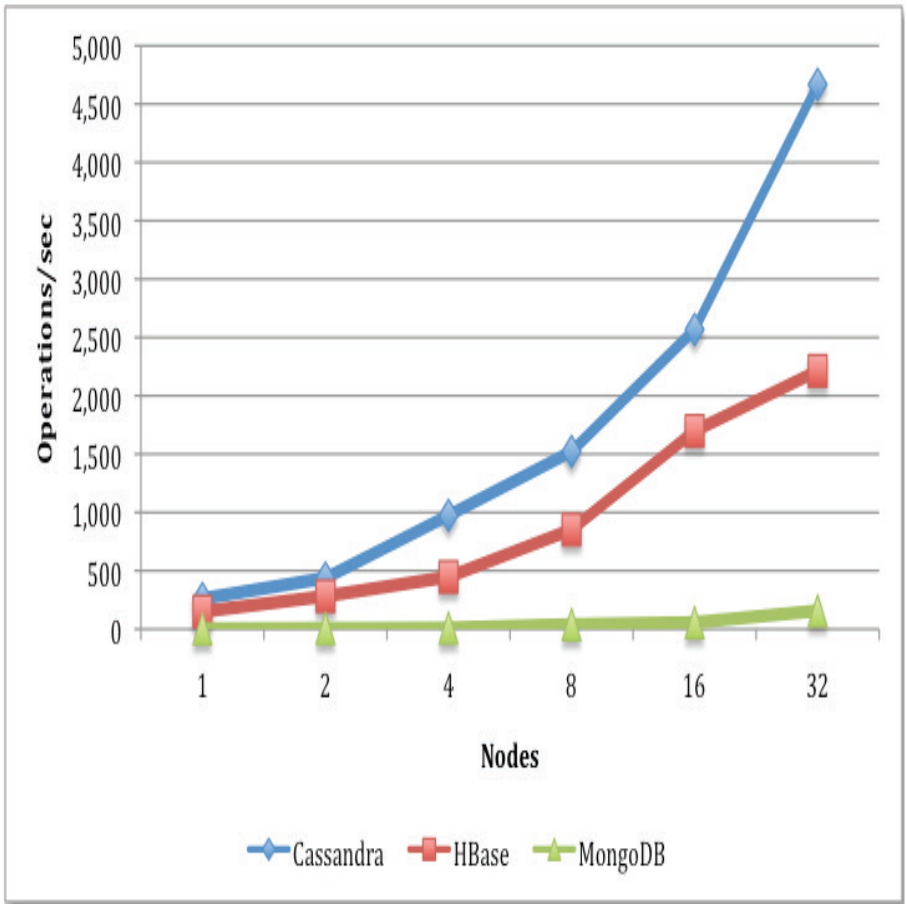

Figure 4.6: Write/Scan mix workload

Table 4.7: Read-latest workload

Shards	Cassandra	HBase	MongoDB	%HBase	%MongoDB
1	1632.87	300.57	445.68	443.26%	266.38%
2	2196.98	580.01	604.21	278.78%	263.61%
4	3567.77	981.69	829.58	263.43%	330.07%
8	8534.39	1998.04	1902.02	327.14%	348.70%
16	12402.94	3976.66	1772.83	211.89%	599.61%
32	23462.98	5151.08	2611.50	355.50%	798.45%

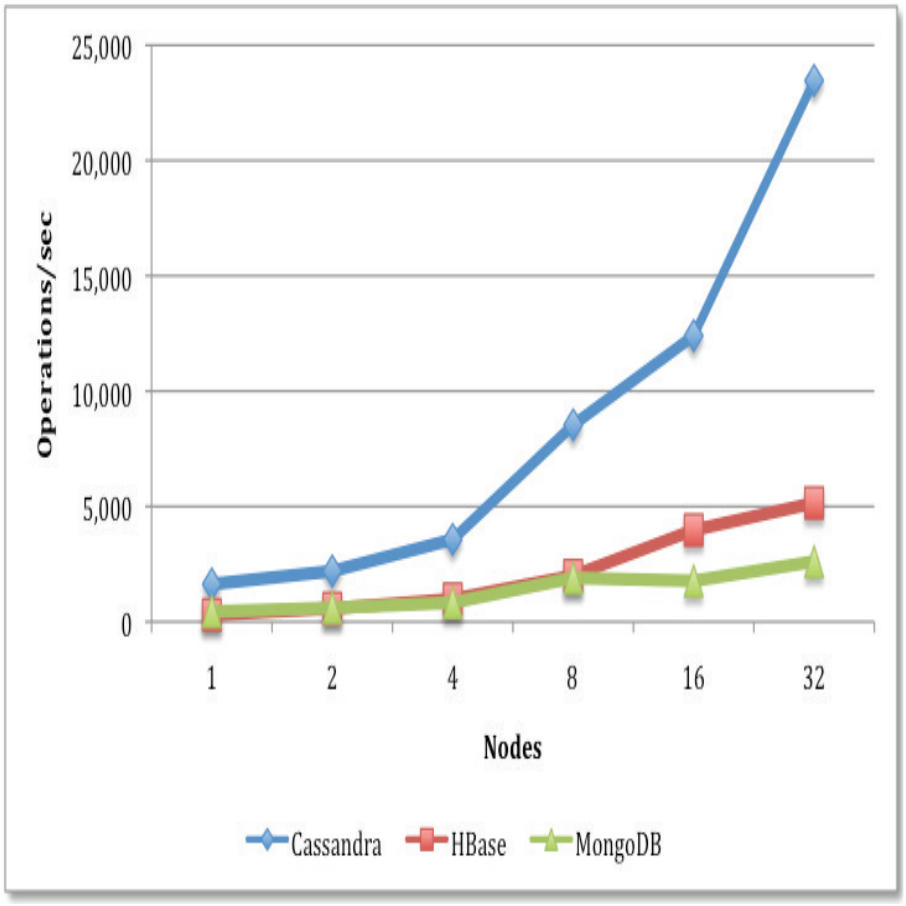

Figure 4.7: Read-latest workload

Table 4.8: Read-modify-write workload

Shards	Cassandra	HBase	MongoDB	%HBase	%MongoDB
1	467.34	210.73	126.78	121.77%	268.62%
2	937.64	412.90	196.89	127.09%	376.23%
4	1873.46	725.07	335.35	158.38%	458.66%
8	3347.56	1136.38	831.00	194.58%	302.84%
16	7333.63	2889.74	1025.19	153.78%	615.34%
32	13708.59	4213.97	1381.47	225.31%	892.32%

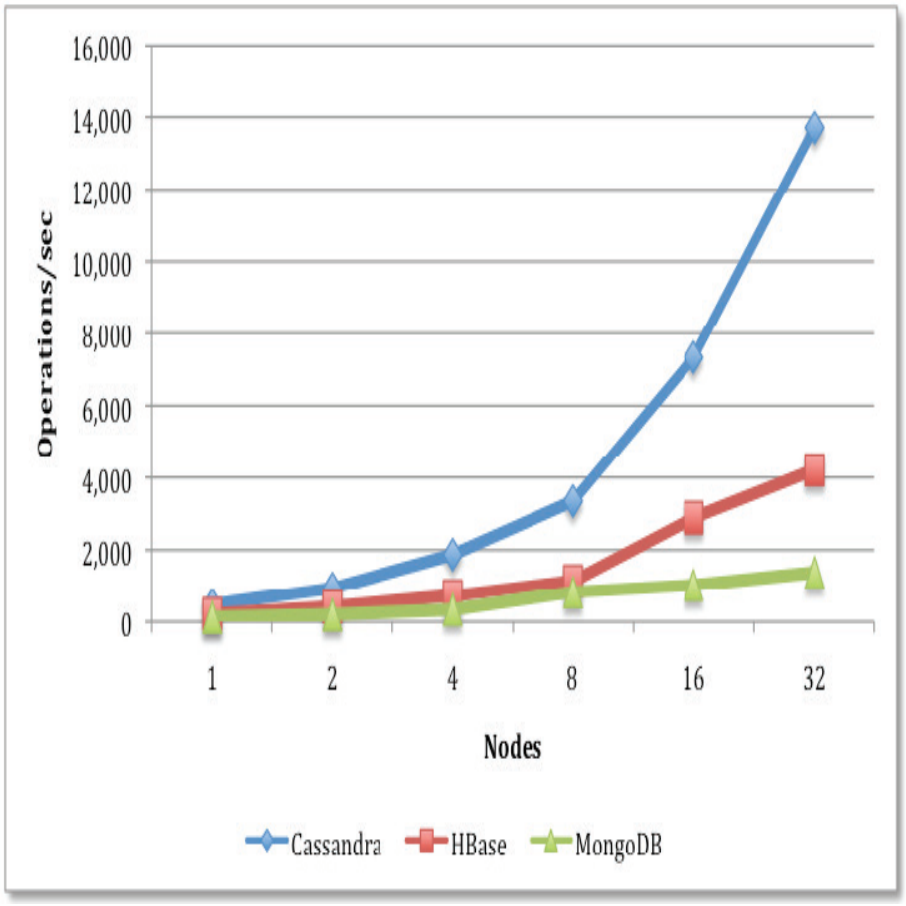

Figure 4.8: Read-modify-write workload

Table 4.9: Read latency across all workloads

Shards	Cassandra	HBase	MongoDB	%HBase	%MongoDB
1	225509.64	492450.15	499810.74	-118.37%	-121.64%
2	371531.12	638811.96	625352.17	-71.94%	-68.32%
4	340136.80	713152.11	719959.11	-109.67%	-111.67%
8	392649.61	766141.42	626442.52	-95.12%	-59.54%
16	395099.39	768299.09	751722.25	-94.46%	-90.26%
32	388945.97	1147158.86	585785.52	-194.94%	-50.61%

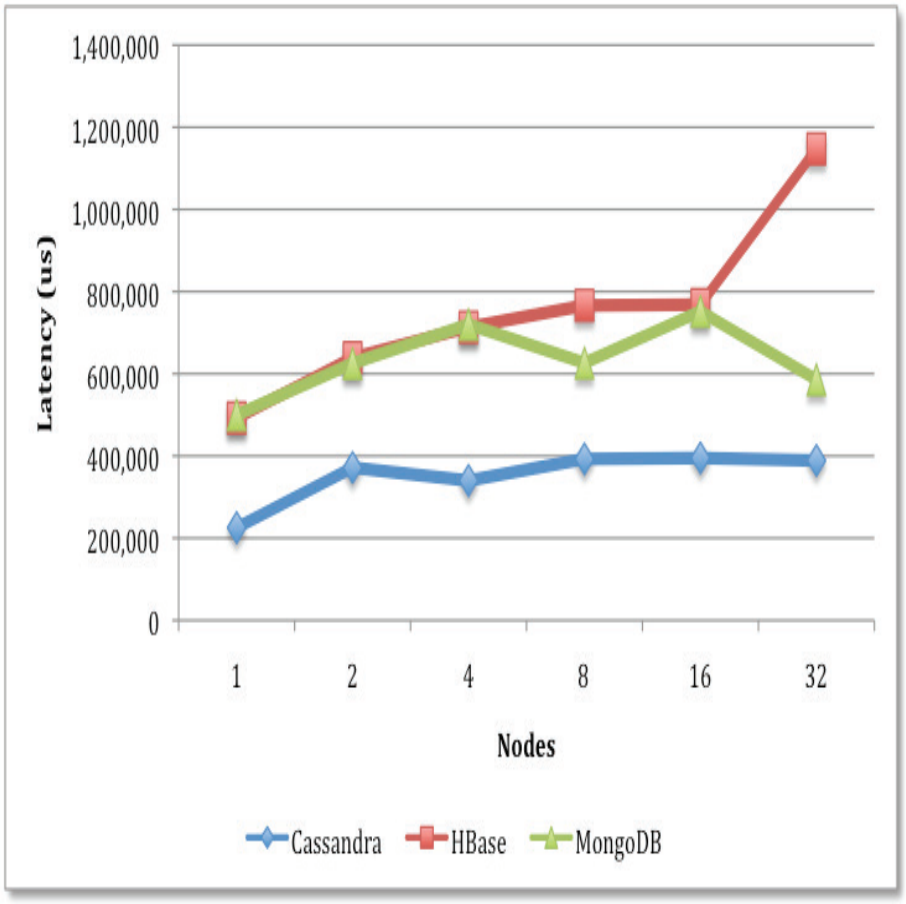

Figure 4.9: Read latency across all workloads

Table 4.10: Update latency across all workloads

Shards	Cassandra	HBase	MongoDB	%HBase	%MongoDB
1	3653.41	497232.17	287122.54	-13510.08%	-7759.03%
2	4187.51	592311.03	1453788.10	-14044.71%	-34617.24%
4	3654.28	707863.61	6481448.75	-19270.81%	-177265.96%
8	5432.62	700204.14	5489490.76	-12788.88%	-100946.84%
16	5007.98	723751.86	10566402.33	-14351.97%	-210891.30%
32	3138.56	1311862.19	18919252.94	-41698.22%	-602700.42%

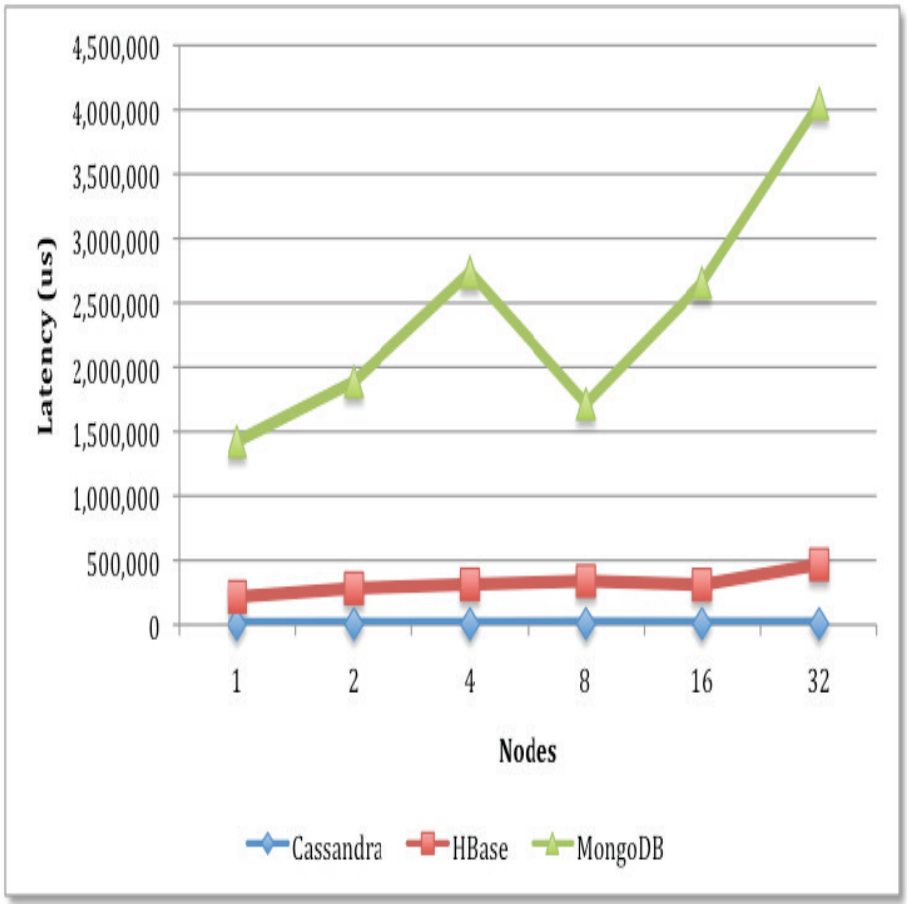

Figure 4.10: Update latency across all workloads

Table 4.11: Scan latency across all workloads

Shards	Cassandra	HBase	MongoDB	%HBase	%MongoDB
1	3292.73	216787.62	1423493.49	-6483.83%	-43131.41%
2	5244.74	282853.22	1887408.14	-5293.08%	-35886.69%
4	5477.54	313344.90	2738696.57	-5620.54%	-49898.66%
8	7875.85	339324.92	1716909.25	-4208.42%	-21699.67%
16	6975.09	310811.07	2656536.27	-4356.02%	-37986.05%
32	7052.18	467145.59	4050434.43	-6524.13%	-57335.21%

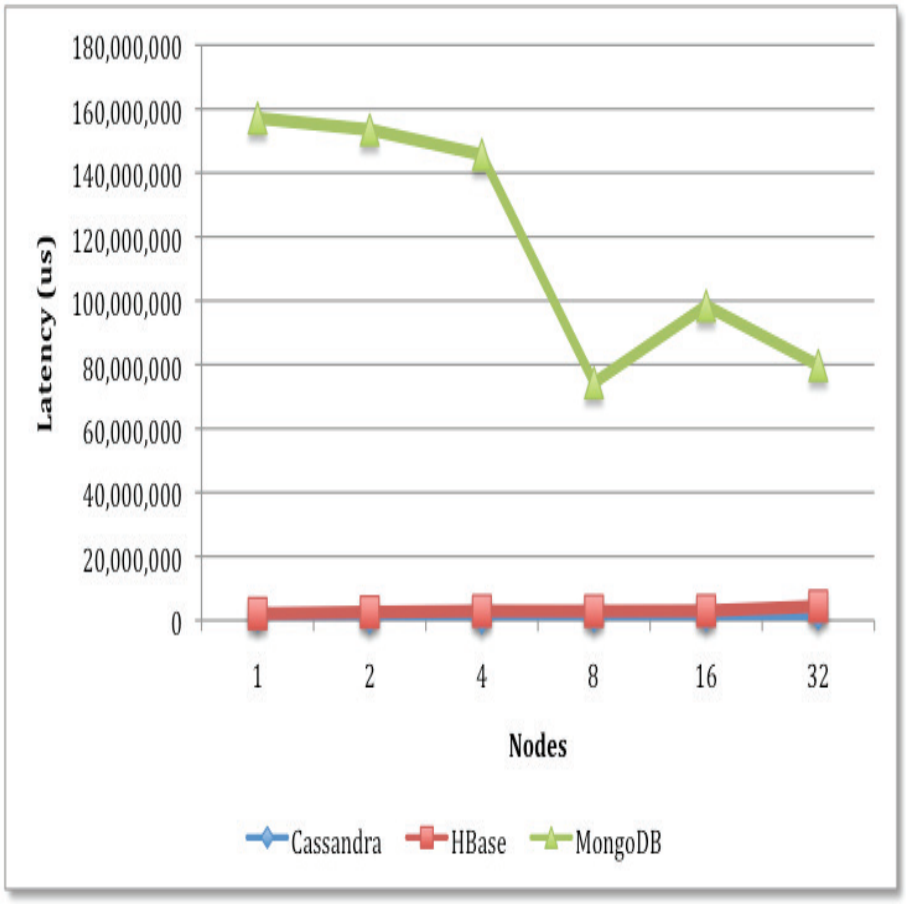

Figure 4.11: Scan latency across all workloads

Table 4.12: Insert latency across all workloads

Shards	Cassandra	HBase	MongoDB	%HBase	%MongoDB
1	1528571.77	2112239.43	157116882.28	-38.18%	-10178.67%
2	958200.98	2591140.03	153509366.73	-170.42%	-15920.58%
4	879313.90	2940485.45	145702505.42	-234.41%	-16470.02%
8	1398940.60	2907451.70	74329305.89	-107.83%	-5213.26%
16	1670090.64	2997596.75	98435275.64	-79.49%	-5794.01%
32	2113042.54	4403775.62	79787876.25	-108.41%	-3675.97%

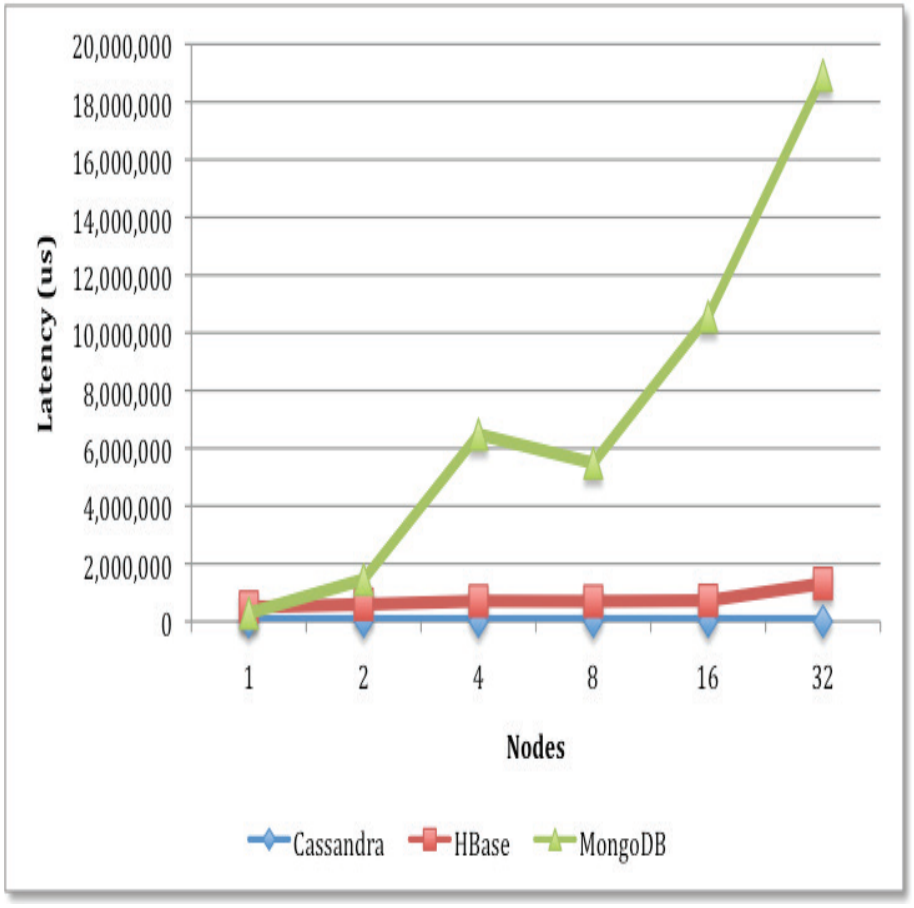

Figure 4.12: Insert latency across all workloads

5 Conclusion

In this paper, they show that NoSQL databases have replaced tradition relational databases for the modern applications. They also presented that performance is the key feature for choosing a particular NoSQL database. In this paper, they have analyzed the performance of the top three NoSQL databases. They show the importance of the performance for the application in this modern era. In this paper, they show that Cassandra outperforms all the other NoSQL databases. Moreover, they suggest that IT professionals have to work on understanding the application before choosing a particular database. Furthermore, they describe the importance of benchmarks as used in their research work to choose a particular database. These benchmarks are critical to the IT professionals on deciding a particular database for the application. However, they conclude that the best way to choose a database is to conduct a formal POC(Proof-of-Concept) an environment in which the application with the corresponding database would run with different workloads or using the idea of benchmarking.

5.1 Discussion

This paper presented only the advantages of Cassandra database and tested only similar type of a dataset. According to several other research work below factors are to be considered before choosing the database

- The data model for the application has to be identified
- Corresponding data sets have to be known prior choosing the database
- Replication is a feature of NoSQL database, identify whether the application requires replication
- Identify whether the application requires transaction support
- The most important step is to identify the performance requirements, whether the application requires a high performance or could compensate in certain cases
- Then prototype the application and test its performance

- The database concerning the requirements has to be chosen. Since there is no perfect solution for choosing the database.

Literature

[1] *Cassandra Architecture,*
http://www.edureka.co/blog/introduction-to-cassandra-architecture/
(visited on 2016-05-04) vii, 5

[2] *Cassendra Homepage,*
http://cassandra.apache.org/ (visited on 2016-05-04) 1, 3

[3] *HBase Architecture,*
http://www.edureka.co/blog/overview-of-hbase-storage-architecture/
(visited on 2016-05-04) vii, 7

[4] *MongoDB Architecture,*
https://www.ibm.com/developerworks/data/library/techarticle/dm-1306mongodb/ (visited on 2016-05-04) vii, 10

[5] *Neo4j Homepage,*
http://neo4j.com/ (visited on 2016-05-04) 1

[6] *Voldemort Homepage,*
http://projectvoldemort.com/ (visited on 2016-05-04) 1

[7] Anderson, J C. ; Lehnardt, Jan ; Slater, Noah: *CouchDB: the definitive guide.* " O'Reilly Media, Inc.", 2010 1

[8] Chang, Fay ; Dean, Jeffrey ; Ghemawat, Sanjay ; Hsieh, Wilson C. ; Wallach, Deborah A. ; Burrows, Mike ; Chandra, Tushar ; Fikes, Andrew ; Gruber, Robert E.: Bigtable: A distributed storage system for structured data. In: *ACM Transactions on Computer Systems (TOCS)* 26 (2008), Nr. 2, S. 4 1

[9] Chodorow, Kristina: *MongoDB: the definitive guide.* " O'Reilly Media, Inc.", 2013 1, 7

[10] CORPORATION, DATASTAX: Benchmarking Top NoSQL Databases. 2013 11, 14

[11] CORPORATION, DATASTAX: Introduction to Apache Cassandra. 2013 3

[12] DeCandia, Giuseppe ; Hastorun, Deniz ; Jampani, Madan ; Kakulapati, Gunavardhan ; Lakshman, Avinash ; Pilchin, Alex ; Sivasubramanian, Swaminathan ; Vosshall, Peter ; Vogels, Werner: Dynamo: amazon's highly available key-value store. In: *ACM SIGOPS Operating Systems Review* Bd. 41 ACM, 2007, S. 205–220 1

[13] George, Lars: *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011 1, 6

[14] Inc, MongoDB: MongoDB Architecture Guide. 2015 7

[15] Judd, Doug: Scale out with HyperTable. In: *Linux magazine, August 7th* (2008) 1

[16] Vora, Mehul N.: Hadoop-HBase for large-scale data. In: *Computer science and network technology (ICCSNT), 2011 international conference on* Bd. 1 IEEE, 2011, S. 601–605 6