Performance Analysis of HBase and Cassandra

Student No: 18179797

August 18, 2019

In this paper we examine the performance of single node Cassandra and HBase databases. We do this using YCSB open source software with a variety of core workloads. We find that HBase gives superior throughput performance in most cases. The read operation excepted however, Cassandra provides lower latencies. However, anomalous behaviour where the latency decreased with throughput was observed in this study. As such, the methods and results obtained in this paper will need to be revisited at some later date.

1 Introduction

Giant internet companies such as Google, Yahoo! and Facebook store the details and the content created by their millions of users. As these companies and their customer base grew, they were faced with the technological challenge of efficiently storing and processing ever larger amounts of data, on the scale of exabytes ($\approx 10^{18}$ bytes) and beyond. To complicate matters further, this data comes in a wide variety of forms from structured data such as user profile information to unstructured data in the form of text, audio and video files.

To address these requirements, these companies invested and created new database technologies that were distributed and easily scalable in nature. Two of these NoSQL (aka Big Data) systems, Cassandra and HBase, are the subject of this paper. Specifically we are interested in comparing their performance characteristics. To execute this performance benchmark, we use the Yahoo! Cloud Serving Benchmark (YCSB) software (**Cooper et al. 2010**).

YCSB has emerged as the de facto standard for executing simple benchmarks for NoSQL systems. It comes with a core set of workloads to evaluate different aspects of a system's performance. Each workload represents a particular mix of read/write operations, data sizes, request distributions, and so on.

In sections 2,3 and 4, we shall describe the architecture and features of the Cassandra and HBase database management systems. Section 5 consists of a brief literature review of the performance characteristics of the two systems. And finally, in sections 6,7 and 8 we relate the details and results of our own performance testing of the two databases.

2 Key Characteristics of HBase and Cassandra

2.1 Distributed Database Systems

HBase and Cassandra are examples of distributed database systems. Such systems were designed to specifically address storage scalability issues. The key innovation was to store data not on a single machine, but to distribute it across a cluster of independent servers. To increase storage capacity, one simply needed to add more servers to the cluster.

However, this ease of scalability comes with some significant drawbacks. For one, if you had 1000 servers (aka nodes) in your cluster for example, the probability of a server failing will be relatively large. And if this unfortunate event were to happen, the data on your failed server may be lost. As such, it is necessary for distributed systems to be made as fault tolerant as possible. This is achieved by replicating the data on every server across other servers so that we effectively always have our data backed up. This also has the fortunate effect of making the database highly available. That is, if a node is down or uncontactable, we can still retrieve the data from another node which has the replica data.

However, though data replication helps us in alleviating issues relating to fault tolerance and availability,

it unfortunately introduces the complication of having to ensure data consistency. That is, what if the replication fails to complete successfully so that we now have equivalent data entries with different results? Which of these data entries should then be returned to the user if queried for? This is the so-called entropy problem and there are various schemes and protocols that can be implemented to alleviate the issue, though usually at some cost e.g. Increased latency.

The issues of data consistency, database availability and fault tolerance addressed above are nicely summarised in the CAP theorem. This says that a distributed system can never satisfy all three of these properties. That is, the system can never guarantee that it is simultaneously:

- Consistent
- Available
- Partition Tolerant

So for example, suppose that when querying the database there was a network issue (aka a network partition event) so that we only had access to a single node where our requested data resided. Though the system could certainly return with the data available on the single node, it would do this without any guarantee that the information was correct. Alternatively it could wait for the replica nodes to come back online and then verify the veracity of the data before returning the query result. In the first approach, our system would be available but potentially not be data consistent, and in the latter, the inverse is the case.

Distributed systems are inherently more complicated than their simpler non-distributed cousins. This is primarily because the system is a cluster of independent nodes that communicate with each other across a network. As such, there is significant overhead in keeping track of the state of these nodes and the communication between them. Apache Zookeeper, a distributed in-memory database, has been adopted widely by many distributed systems to perform this task and is used in both HBase and Cassandra.

2.2 Column Store Databases

Most database systems are row-based systems. That is, a database table is broken down into a sequence of row records and stored in a series of blocks on harddisk. Scanning through a table would therefore typically involve executing multiple disk operations, with a subsequent impact on query performance. This is not a particular issue if one wants to retrieve the **entire** record. However, this is usually not the case. Typically one is only interested in retrieving a single or small number of fields from a table.

An alternative strategy therefore is to instead store the field/column in its own file or set of files. HBase and Cassandra both adopt this strategy. Retrieving the values of a single field from the table then requires less disk operations than in a row-based database. As such, HBase and Cassandra are not optimised for returning entire records from tables.

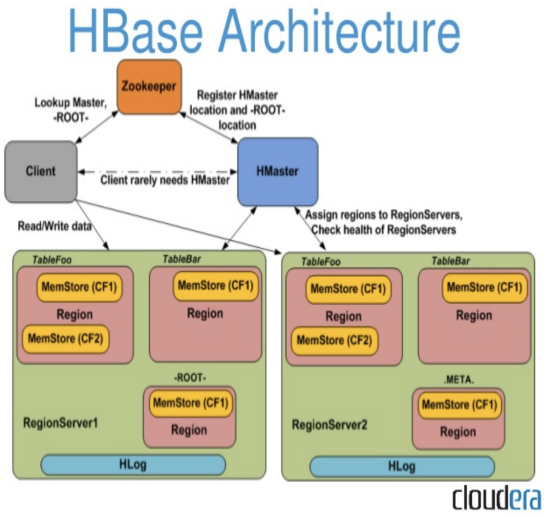

Figure 1: *Image Credit: Cloudera*

3 Database Architectures

A brief introduction to the architectures of HBase and Cassandra are given below. Essentially though, the key difference is that Cassandra has a masterless architecture, while HBase has a master-based one. As such, if the master server is down, HBase will be unavailable whereas Cassandra is highly available.

3.1 HBase Architectures

As shown in Fig. 1, HBase has a single HBase master node (HMaster) and several slaves referred to as region servers. All client requests are channeled through the HMaster who forwards it to the corresponding region server. The region server, essentially the worker node, then executes the read, write, update, and delete request as appropriate. A Region Server process runs on every HDFS (Hadoop Distributed File System) data node and consists of the following components:

- Block Cache - This is the read cache. Most frequently read data is stored in the read cache and whenever the block cache is full, recently used data is evicted.
- MemStore - This is the write cache and stores new data that is not yet written to the disk. Every column family in a region has a MemStore.
- Write Ahead Log (WAL) is a file that stores new data that is not persisted to permanent storage.
- HFile is the actual storage file that stores the rows as sorted key values on a disk.

As in many distributed systems, HBase uses ZooKeeper as a distributed coordination service. The

Figure 2: *Cassandra Ring Structure*

ZooKeeper service keeps track of all the region servers that are in an HBase cluster. HMaster can thus contact ZooKeeper to obtain the details of any region server. If nodes were to fail within an HBase cluster, ZooKeeper will automatically trigger error messages and start repairing the failed nodes.

3.2 Cassandra Architecture

The architecture of Cassandra has the following key features:

- It is a key-value store with no master or slave nodes.
- The nodes have an (hash) ordered ring structure (See Fig. 2)
- The keys are hashed to determine the node the data should be written to.
- Data is by default replicated in the two forward nodes in the ring.
- Data is kept in memory and lazily written to the disk.

It is classified an AP system, meaning that availability and partition tolerance are generally considered to be more important than data consistency in Cassandra. It does however possess tunable consistency in that it allows you to increase the level of data consistency, though with the knock-on impact to performance.

4 Cassandra and HBase Review

In this section, we shall describe in detail the data storage and retrieval mechanisms of the two databases and discuss some more on their scalability, availability and reliability.

4.1 Storage and Retrieval of Structured, Semi-structured and Unstructured Data

The architectures and key characteristics of both systems have already been described. Here, we simply give more detail on the specifics of the data storage and retrieval process in both systems.

4.1.1 Cassandra

The data writing process can be broken down as follows:

1. Data is first written to a commitlog on disk for persistence.
2. The data is then sent to the appropriate node as determined by its hash value.
3. The node then writes the data to an in-memory table called memtable, and from there is written to disk to a Sorted String table (SSTable). This table stores all the required updates to the table. Note that these SSTables are merged to save and free up disk space in a process referred to as compaction.
4. When the SSTable crosses a certain threshold, calculated based on data size and number of objects, it dumps itself to disk so that the data is updated to the actual table.

Perhaps the most notable feature is the use of hashing to determine the node on which data should be stored/retrieved. That is, a hash value (a 127 bit positive integer) is calculated based on the primary key of the data. We shall discuss the hashing process in the next section.

If a client sends a request to a particular node in the cluster, the read process in Cassandra proceeds as follows:

1. Read happens across all nodes in parallel.
2. The particular node(s) from which the data is read depends on "how far" the data is from the node to which the client sent its query. Preference is given to the closest node.
3. Data in the memtable and SSTable of the node is checked first so that the data can be retrieved faster if it is already in memory.

4.1.2 HBase

HBase is built on top of the Hadoop Distributed file system (HDFS). The HDFS was originally designed for batch processing. That is, it was designed for high throughput and high latency. HBase was thus originally designed to be a random access storage and retrieval data platform to help alleviate this limitation with Hadoop.

The write operation flow executes in the following sequence:

1. The client sends a write data request to the HRegionServer which then writes the data to the MemStore of HRegion.
2. When the MemStore reaches a preset threshold, the data is flushed into a StoreFile.
3. As the number of StoreFile files increases and exceed a certain threshold, the Compact merge operation is triggered, and multiple StoreFiles are merged into one StoreFile.
4. When the size of a single StoreFile exceeds another threshold, the Split operation is triggered to split the current HRegion into two new HRegions.

The read operation proceeds in the following manner:

1. The client accesses Zookeeper to obtain metatable information.
2. Using this information, the HRegion of the target data and the corresponding HRegionServer is identified.
3. The data is subsequently returned by the HRegionServer.

It should be noted that the memory of the HRegionserver is divided into two parts: MemStore and BlockCache. The former is mainly used to write data, while the latter is mainly used to read data.

4.2 Scalability, Availability and Reliability

In our discussion in section 2, we have already outlined in general terms the key characteristics of distributed systems. These are their scalability, their high availability and fault tolerance i.e. Reliability. As such, we shall thus briefly compare and contrast some details specific to HBase and Cassandra databases here.

4.2.1 Scalability

Both databases are inherently scalable by design. That is, both systems are designed so that new servers can easily be added into the existing cluster. HBase is built on top of the Hadoop Distributed File System (HDFS). One of the interesting features of HBase is auto-sharding. This simply means that a table, whose data is split into Regions, are dynamically distributed by the system when they become too large. That is, when a region becomes too large due to the adding of data, the region automatically splits into two, creating two roughly equal halves.

The key mechanism controlling the scalability of Cassandra is its use of consistent hashing. Essentially, hashing of the data is executed so that data gets distributed evenly over the available cluster nodes. This is done to ensure that no particular node is overworked i.e. It is done for load balancing reasons. With Cassandra, a specific type of order preserving hash known as consistent hashing is implemented. This ensures that if a node is removed or a new node added, that the mapping of the data to the nodes is only minimally affected. For example, if a node is removed, then its interval (i.e. the range of hash values that were mapped to it) is taken over by a node with an adjacent interval. All the other nodes remain unchanged.

4.2.2 Availability and Reliability

As mentioned earlier, high availability is another key property of distributed systems. By distributing and replicating the data over many nodes, one can access the data even if a node fails or goes offline. High availability is inherent in the design of Cassandra in that there is no master node that can act as a single point of failure. All nodes are equal.

On the other hand, while HBase provides high-reliability via the low-level storage support of HDFS, it does not offer high availability. For one, it possesses a master server. And secondly, in contrast to Cassandra which is an AP system, HBase is considered a CP system so that it prioritises data consistency and partition tolerance over availability. So for example, let us say that a network partition event were to occur so that some nodes could not communicate with each other. If a read request were to come into the HBase cluster during this event, and the system was unable to validate the requested data with the replica nodes, HBase would likely not return the requested data until the network issue was resolved and validation could be executed. The HBase system is therefore is not truly available in such a scenario.

5 Literature Survey

In this section, we shall review some papers relevant to the benchmarking of NoSQL database systems. Particular attention will be paid to performance results obtained for HBase and Cassandra.

In the original paper on YCSB (**Cooper et al. 2010**), a standard benchmarking framework to assist in the evaluation of cloud database systems was described and the performance of some databases analysed. With databases, there is always the inherent tradeoff between optimising for reads and optimising for writes. "Cassandra and HBase attempt to always perform sequential I/O for updates. Records on disk are never overwritten; instead, updates are written to a buffer in memory, and the entire buffer is written sequentially to disk. Multiple updates to the same record may be flushed at different times to different parts of the disk. The result is that to perform a read of a record, multiple I/Os are needed to retrieve and combine the various updates. Since all writing is sequential, it is very fast; but reads are correspondingly de-optimized."

The paper also describes the "inherent tradeoff between latency and throughput: on a given hardware

setup: as the amount of load increases, the latency of individual requests increases as well since there is more contention for disk, CPU, network, and so on." As such as throughput increases, one expects the latency to also increase. Various performance scenarios are outlined. For example, as in the present paper, one can keep the system static and examine the throughput and latency response as the size of the workload increases. But they also describe other scenarios such as measuring the performance as the number of nodes is increased.

They benchmark four databases; Cassandra, MySQL, HBase and PNUTS. They found that "Cassandra and HBase have higher read latencies on a read heavy workload than PNUTS and MySQL, and lower update latencies on a write heavy workload."

They also reported that for update heavy workloads, Cassandra , which is optimised for write-heavy workloads, achieved the best throughput and the lowest latency for reads. HBase had "very low latency for updates, since updates are buffered in memory." Because of efficient sequential updates, Cassandra also provided superior throughput to HBase. For read-heavy workloads, both Cassandra and HBase underperformed Mysql and PNUTS. It was also notable that Cassandra had poor scan latency.

In the original paper introducing Cassandra (**Lakshman and Malik 2010**), it is described as "designed to run on cheap commodity hardware and handle high write throughput while not sacrificing read efficiency." It was designed by Facebook to meet their scalability and reliability requirements and specifically to meet the demands of their inbox Search feature. The system architecture is outlined with particular attention paid to partitioning, replication, failure detection and scaling. In their conclusion they state that "we have built, implemented, and operated a storage system providing scalability, high performance, and wide applicability. We have empirically demonstrated that Cassandra can support a very high update throughput while delivering low latency."

Though it has improved in recent times, lack of security features has proven to be a disadvantage with Big Data cloud storage servers. One solution is to encrypt the data on the client side before transmitting it to the cloud. In (**Waage and Wiese 2015**), using YCSB, they examine the performance of HBase and Cassandra under this scenario. The results show that for single read and write operations, the performance loss is acceptable while for range scans the impact can be quite severe. Interestingly, they quote the performance with no encryption and find that under the update heavy workload A that Cassandra gave superior throughput and lower average read and update latencies when compared to HBase. Under workload E, the throughput on Cassandra was only slightly superior.

In (**Manoharan 2013**), the performance of read, write, delete, and instantiate operations on key-value stores implemented on NoSQL and SQL databases are compared. While NoSQL databases are generally optimized for key-value stores, SQL databases are not. Yet, they find that not all NoSQL databases perform better than the SQL databases tested. Furthermore they discovered a wide variation in performance in the NoSQL databases based on the type of operation (such as read and write). They found that while Cassandra was slow on read operations, it performed relatively well for write and delete operations.

In a similar study on NoSQL databases (**Gandini et al 2014**), the performance of Cassanda, MongoDB and HBase was examined. Specifically, the authors were interested in the performance impact of different configurations e.g. Number of nodes in cluster. In their investigations, they also measured the performance of single node clusters and discovered that HBase gave superior throughput performance to Cassandra. Furthermore, though the update latencies in the two systems were approximately equal, the read latency was found to be much higher for Cassandra.

The common problem of determining the appropriate system to purchase was addressed in (**Klein et al 2015**). Here, the authors describe their strategy in selecting a NoSQL database system for a large, distributed healthcare organisation. Using YCSB, they present the results of both extensive performance and scalability testing on three NoSQL databases, namely Cassandra, Riak and MongoDB. In all cases they found that Cassandra provided the best overall performance. That is, as the cluster was scaled up in size, the performance did not degrade. It was noted however that though Cassandra achieved the highest overall throughput, it also delivered the highest average latencies.

They attribute this to several factors. "First, hash-based sharding spread the request and storage load better than MongoDB. Second, Cassandra's indexing features allowed efficient retrieval of the most recently written records. Finally, Cassandra's peer-to-peer architecture and data centre aware features provide efficient coordination of both read and write operations across replicas and data centres."

6 Performance Test Plan

The objective in this project was to benchmark the performance of single node Cassandra and HBase databases using the open source YCSB software. Both databases (together with the YCSB software) were installed on a VM hosted on a MacBook Pro laptop. The VM was an Ubuntu 18.04.2 image which was allocated 4 CPUs with a base memory of 8192 MB of base memory and 20 GB of disk space. The versions of Cassandra and HBase installed were 3.9 and 1.4.10 respectively.

The testing was to be done using a test harness contained in a simple linux shell script. This shell script when executed reads in a number of configuration files containing the database types to be tested, and the number of operations and YCSB workloads to be exe-

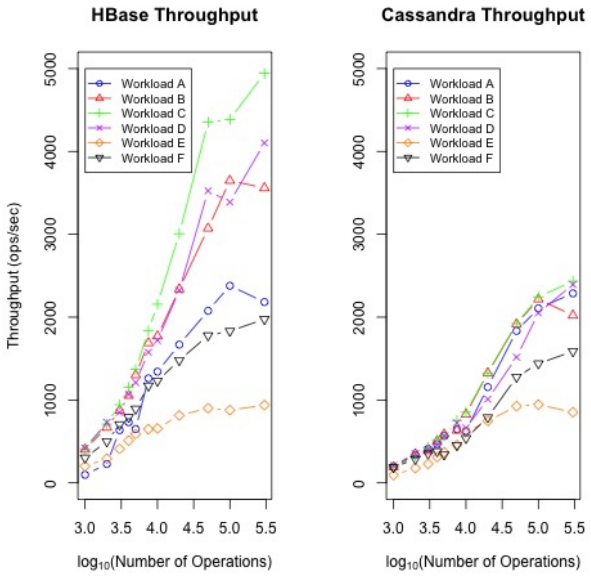

Figure 3: *Throughput Performance on workloads A-F*

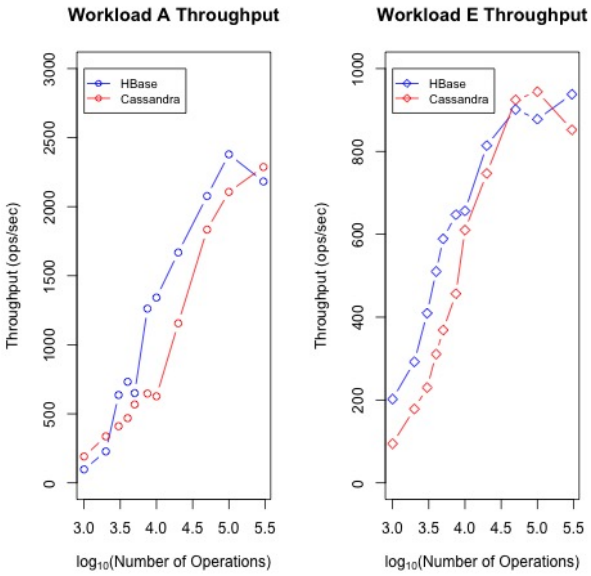

Figure 4: *HBase and Cassandra Throughput Performance for workloads A and E*

cuted. Output files containing performance statistics are then automatically generated.

For the testing, all six of the YCSB core workloads were to be executed for 1000, 2000, 3000, 4000, 5000, 7500, 10000, 20000, 50000, 100000 and 300000 operations (In YCSB, these are the values assigned to the parameter operationcount for each test). Therefore a total of 11*6*2 = 132 tests were to be executed (It should be noted that the test harness cleans down the test database table (named usertable) and loads it with dummy data before executing each test).

7 Evaluation and Results

The data from the YCSB output files were parsed and the relevant statistics extracted (using simple awk and sed commands).

We shall begin by looking at the overall throughput performance of the two databases before examining the performance statistics of each workload. In Fig. 3 we display the overall throughput for each workload as a function of the number of required operations to be executed. We immediately observe that for most workloads, HBase gives superior throughput.

Only in the case of the update heavy workload A and Scan heavy workload E did the throughputs become comparable (See Fig. 4). Note that in some cases the throughput actually decreases as the load increases, but this is likely due to the statistical noise. Ideally one would run the tests a number of times and obtain averaged results to minimise this experimental error.

Another point of note in figures 3 and 4 is that there are signs of saturation in throughput occurring. That is, as the workload placed on the database increases, the throughput approaches the maximum capacity of the system so that it essentially levels off. We can see evidence of this in the plots.

In Fig. 5 we show the average read latency for workloads A-E in both Cassandra and HBase. The results are surprising. One would expect that as the workload increased, the average latency would increase also as requests get backed up in a queue. The opposite is in fact observed. Note that similar graphs for the update, insert and scan latencies were also plotted (not shown here) and in all cases showed the same behaviour. That is, the average latencies decreased as the operation count increased. The same effect manifests itself if one were to look at the 95th or 99th percentile latebcy values.

As of this time, I have no explanation for this anomalous behaviour. A number of possibilities present themselves. It may be user error: Perhaps the number of client threads (which was kept at 1) should have been increased. Or I am mistaken in my interpretation. The results might also be some peculiarity arising from the test environment in the VM. Regardless, it is unfortunate as this warps all the proceeding analysis in this section.

In the remaining figures I present latency vs throughput curves for different workloads. The first point to note is that the anomalous latency behaviour manifests itself here again. That is, one expects that as throughput increases, the average latency should increase. Here we observe the opposite.

In Fig. 6 we show the latency vs throughput curves for the update heavy (50% Read/ 50 % Write) workload A (This mimics the workload signature of, for example, a session store that records the actions in a user session). We can see that throughputs are comparable in both Cassandra and HBase for both read

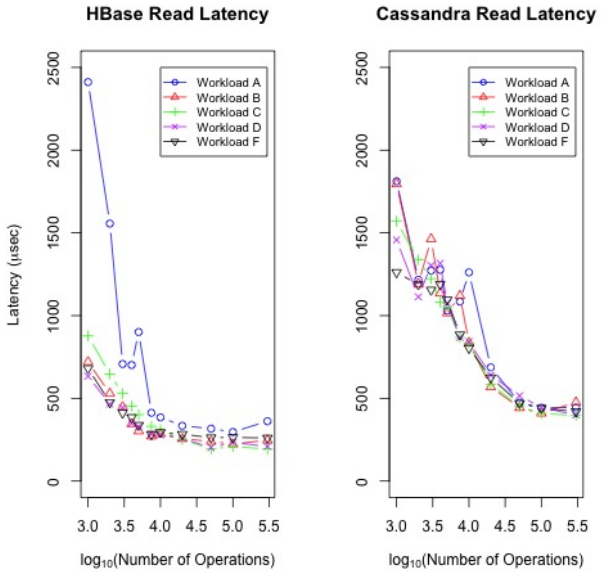

Figure 5: *HBase and Cassandra Read Latencies*

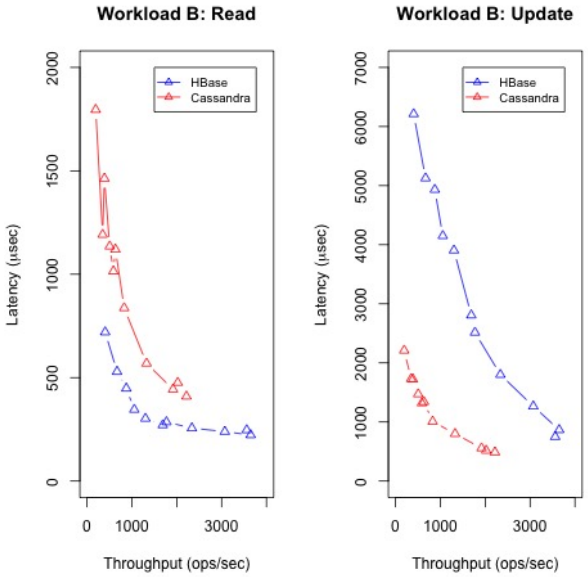

Figure 7: *Workload B Latency vs Throughput curves*

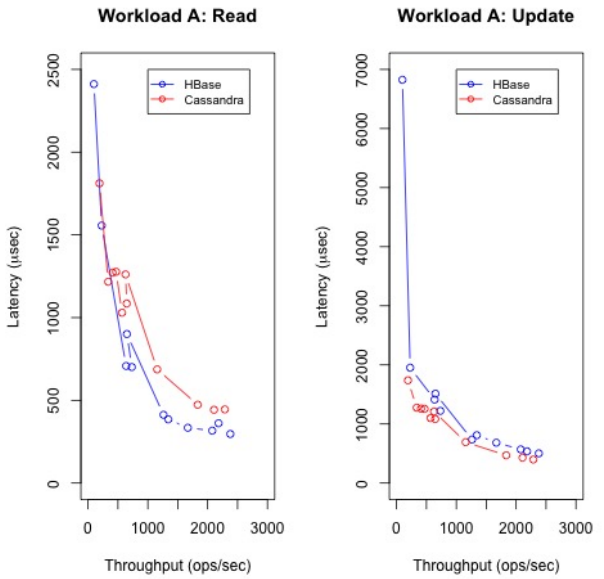

Figure 6: *Workload A Latency vs Throughput curves*

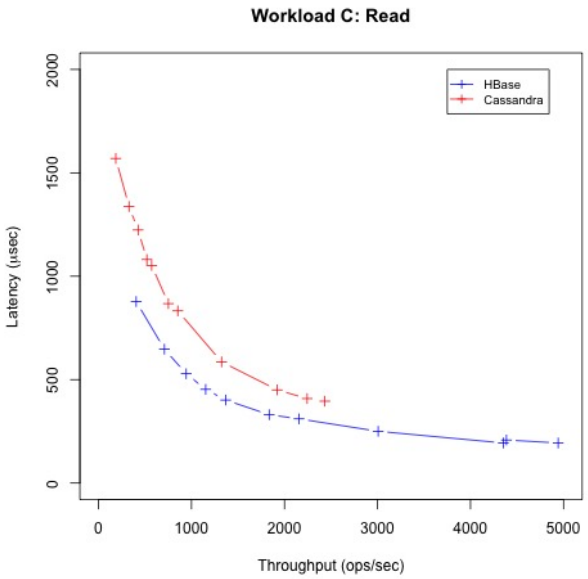

Figure 8: *Workload C Latency vs Throughput curves*

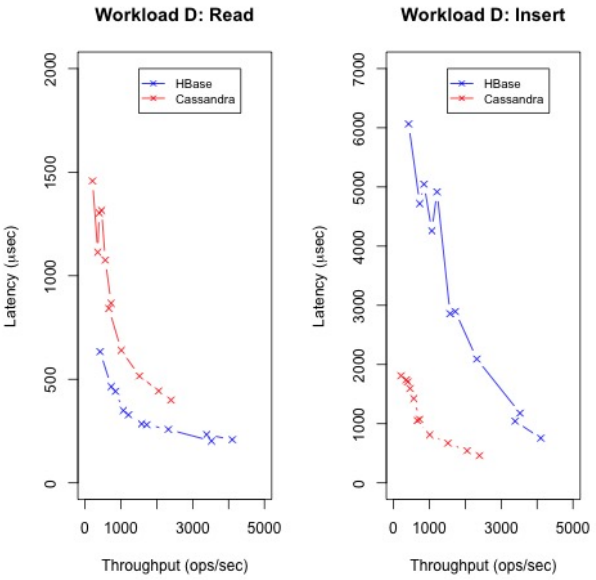

Figure 9: *Workload D Latency vs Throughput curves*

and updates. There is also minimal difference in the latencies between the two systems (though Cassandra shows slightly higher latencies for read operations but lower latencies for writes).

Differences become more stark however when we examine the read heavy (95% Read/ 5 % Write) workload B in Fig. 7. Here we see superior throughputs in HBase. We see that the read latencies in Cassandra are larger for reads and significantly lower than HBase for writes. The same pattern for the read only workload C is seen in Fig. 8 and again in the left hand panel of Fig. 9. In fact the latency-throughput curves of Fig. 9 which show the performance of the systems under the read latest (95% Read/ 5% Insert) workload D look very similar to Fig. 7.

In the results for the short range (95% Scan/ 5% Insert) workload E of Fig. 10 it is interesting to note that the curves for the scan operation converge at large throughput intimating similar performance levels.

We also see identical performance in the right hand panel of Fig. 11 where we show the results for workload F. In summary then, the results suggest that HBase provides superior performance for read operations in that it has both superior throughputs and lower latencies. For insert and update operations however, it is Cassandra that provides lower latencies (and in some cases significantly lower). The performance on scanning is very similar between the two systems.

8 Conclusion and Discussion

In this paper we benchmarked the performance of Cassandra and HBase single node clusters. We found that HBase generally gave superior throughput performance, the exceptions being workloads A and E

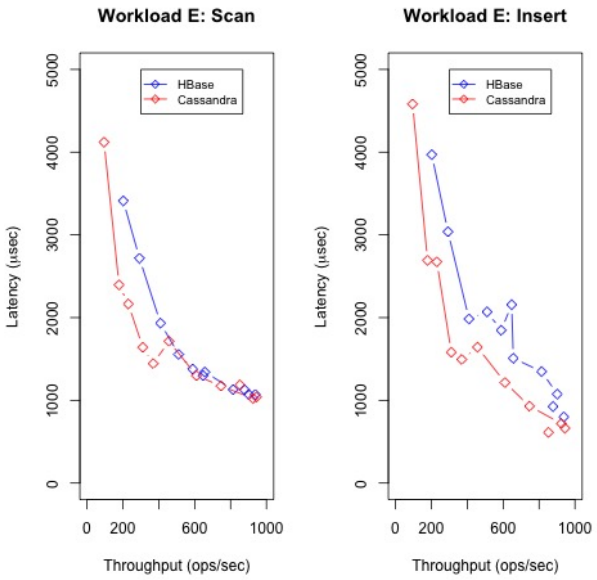

Figure 10: *Workload E Latency vs Throughput curves*

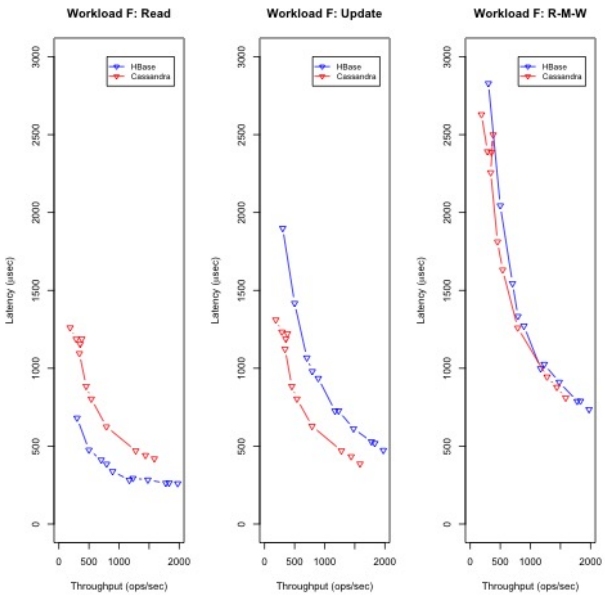

Figure 11: *Workload F Latency vs Throughput curves*

where Cassandra matched it. Certainly in terms of read operations, HBase gave superior results in both throughput and latency. Though this matches the findings of (**Gandini et al 2014**), it disagrees with (**Cooper et al. 2010**).

For insertion and update operations, our results show that HBase suffered from relatively large latencies.

All the above observations and conclusions though should be tempered as the anomaly observed in the latency results casts some suspicion on the work performed in this paper. As throughput increases, the latency is expected to increase. The opposite was observed in our results here. It would thus be wise to perform the same set of tests again on a different system to see if the above anomaly persists.

References

[1] Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R. (2010) Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154.

[2] Lakshman, A., Malik, P. (2010) Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Syst. Rev. 44(2), 35–40

[3] Waage T., Wiese L. (2015) Benchmarking Encrypted Data Storage in HBase and Cassandra with YCSB. In: Cuppens F., Garcia-Alfaro J., Zincir Heywood N., Fong P. (eds) Foundations and Practice of Security. FPS 2014. Lecture Notes in Computer Science, vol 8930. Springer, Cham

[4] Y. Li, S. Manoharan (2013) "A performance comparison of SQL and NoSQL databases", 2013 IEEE Pacific Rim Conference on Communications Computers and Signal Processing (PACRIM), pp. 15-19

[5] Gandini A., Gribaudo M., Knottenbelt W.J., Osman R., Piazzolla P. (2014) Performance Evaluation of NoSQL Databases. In: Horváth A., Wolter K. (eds) Computer Performance Engineering. EPEW 2014. Lecture Notes in Computer Science, vol 8721. Springer, Cham

[6] J. Klein, I. Gorton, N. Ernst, P. Donohoe, K. Pham, and C. Matser (2015) “Performance evaluation of nosql databases: A case study,” in Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems, ser. PABS ’15. New York, NY, USA: ACM, pp. 5–10.