# Scaling Cassandra Database under heavy user loads

*Ferhan Jamal*
*100953487*
Dept. of System and Computer Engineering
Carleton University
Ottawa, Canada
ferhan.jamal@carleton.ca

*Nikhil Nayyar*
*100941327*
Dept. of System and Computer Engineering
Carleton University
Ottawa, Canada
nikhil.nayyar@carleton.ca

***Abstract*— there are several distributed storage systems that have been proposed to provide high scalability and high availability for modern web applications. Due to exponential growth of internet in this era, interactive web applications have changed dramatically over the last 15-20 years, and so did the needs of managing data of those applications. Three interrelated mega trends- Rapid Big Data Management, Big users and Cloud Computing are driving the adoption of NoSQL technology forward, which is an efficient alternative to relational databases. In this paper, we present how Cassandra (a famous NoSQL database aiming to manage large amount of data with no single point of failure) will scale as the web user activities keeps on increasing. We improves Cassandra scalability by using better hardware such as SSD storage instead of deploying Cassandra on regular commodity hardware such as HDD. And also have tried to signify which data serialization format we should use while serializing the data into Cassandra database, as this might be very useful in long term duration with respect to scaling. Conclusion of our experiment shows that we can increase the Cassandra throughput by 40% if we deploy Cassandra database on a better hardware.**

***Keywords: Distributed Storage System; NoSQL Database; Cassandra database; Data Serialization***

## I. INTRODUCTION

Trend of Web applications now days probably have to deal with very large scale of data set. Mostly all the applications require following characteristics having high scalability, high availability and the ability to response quickly even when there exist hundreds of thousands of request concurrently. Comparing with these, strong data consistency and strict transaction support, such as ACID, sometimes can be weakened or even dropped up. Naturally, traditional Relational Database is not suitable for serving these kinds of applications, since most of them are transaction based and are hard to scale to very large size or with a very high cost. In addition, relational database always provide too large feature set that many of them will not be used, which only add the cost and complexity within [6, 7].

In the past few years, many scalable NoSQL related distributed storage systems have been confronted, for example, Google's Bigtable [2], Amazon's Dynamo [3] and Yahoo!'s PNUTS [4] and many more. Based on CAP theory [8], for any distributed system we could not have high availability along with a strong consistency. With such facts and reasoning, most of these systems rest on strong consistency and strict transaction to get high scalability and availability, therefore they can scale out dynamically to the internet size, have peak resilient to the node failure or network failure and provide well to the massive access. Data partition and data replication of data are the two technologies that these systems likely or tend to use to attain above mentioned goals. Data partition can be used to improve scalability and performance while data replication is a good way to get high availability and balance the load like a balancer. Now a days many organizations recognize that operating at a desired scale is better achieved on clusters of standard, commodity servers, and a schema-less data model which is often better for the variety and type of data captured and processed today. [1, 9]

Providing scalability and reliability in this big user's trend is very crucially important these days. Gone are the days when, managing 1000-2000 daily users of an application was sufficient, these days we have to deal and talk about minimum of 70,000 or 1000,000 daily users of an application and if respective databases cannot support or handle many concurrent requests, then it's a big loss of revenue. The large number of users along with the dynamic behavior of the query patterns is driving the need of more reliable and efficient database which should be up for handling concurrent and all time available to proceed with the request. Such database to which we could easily call as "Always Available Database". [9, 10]

In this paper, we present how Cassandra (a famous NoSQL database aiming to manage very large scale data without single point of failure) will scale as the web user activities keeps on increasing and how easy is to scale Cassandra database without having any single point of failure. We will also try to identify what data serialization format we should follow while storing the data into Cassandra so that we don't always run out of space in Cassandra which will also be beneficial in terms of read / write performance while scaling the Cassandra database. We will try to measure the scalability mostly by throughput. If Cassandra writes cannot keep up with user traffic, then we may see timeout when writing to Cassandra or resource utilization increases on Cassandra side, then it's time to increase

Cassandra capacity. Also, we have done several experiments which shows how to improve Cassandra scalability by deploying Cassandra on a very good hardware instead of deploying Cassandra on cheap commodity hardware and that's what lot of people suggests to use Cassandra with cheap commodity hardware.

The remainder of the paper is organized as follows. Section II introduces the related work. Section III presents the background of Cassandra and its architecture along with our architecture diagram of the setup we have and also how we can improve Cassandra on the throughput. Section IV studies different experiments. And finally, we present conclusions and future work in Section V.

## II. Related Work

Bigtable and Dynamo all are widely studied and cited in distributed storage system domain. Bigtable is implemented as sparse, multidimensional sorted maps, which is richer than key-value data model but easy enough. It uses Google File System (GFS) [14] to store data and log information. GFS can divide file into fixed size chunks and balances the data load by distributing those chunks into different nodes evenly. However, GFS use primary copy, pessimistic algorithm to synchronize data between different duplicated nodes, which make it scalable and performance poorly in the write intensive and wide area scenario. [10]

Dynamo is a highly available, eventually consistent key value storage system that uses Consistent Hashing to increase scalability, Vector Clock to do reconciliation and Sloppy Quorum and Hinted hand off to handle temporary failure. In Dynamo, all the nodes and the keys of data items are hashed and then the output values are mapped to a "ring". The node output represent the position of this node in the "ring", while the key's output decide which node this item will be stored. In order to balance the data load, Dynamo map one real node into many virtual nodes, each of them occupy one position in the "ring", different node may have different number of virtual nodes, based on their capacity. [10]

## III. SYSTEM DESIGN

Cassandra is a distributed key-value store which is capable of arbitrarily scaling of large sets with no failure at any point. These data sets may vary server nodes, racks, and even multiple data centers. Several organizations measure their structured data storage needs, in terabytes and petabytes rather than gigabytes, technologies for working with information at scale are becoming more critical and huge. As the scale of these systems grow to cover local and wide area networks, it is important that they continue functioning in the face of faults such as broken links, crashed routers, and failure nodes. Usually failure of a single component in a distributed system is comparatively low, but the probability of failure at some point increases with a direct proportion to the number of components reside. Fault-tolerant, structured data stores for working with information at such a scale are in high demand. Furthermore, the importance of data locality is growing as systems vary large distances with many network hops. Moreover, Cassandra is designed to continue functioning in the face of component failure in a number of user configurable ways and enables high level of system availability by compromising data consistency, as well allows the client to catch up the extent of this trade off. Data in Cassandra is optionally duplicated onto N different peers in its cluster while a gossip protocol ensures that each node maintains state regarding each of its peers. This has the property of reducing the hard depends-on relationship between any two nodes in the system which increases availability partition tolerance [17]

Cassandra is inspired by Bigtable and Dynamo [17], it integrate Bigtable's Column Family based data model and Dynamo's Eventually Consistency behavior and thus can get both of their advantages. As with Dynamo, Cassandra use Consistent Hashing to partition and distribute data into different nodes by hashing both nodes and data's value into a "ring". Nodes discover each other using seed nodes and Gossip protocol. Using Gossip protocol, each node exchanges information about its knowledge of the cluster with up to three each second. This way, after a while, all the nodes know about each other. The data is stored in a per row basis. The PARTITIONER defines how rows are distributed in the cluster based on row id hashes. Allowed hash values define a range that can be viewed as a ring. Each node is responsible for a range in the ring. To improve availability and balance the load, all the data are replicated into N nodes, where N is the replication number that can be configured in advance. First, it will assign each data item a Coordinator Node, which is the first node this data item meet when walk clockwise around the "ring", then replicate this data item to the next N-1 clockwise successor nodes in the "ring". To adapt between strong consistency and high performance, Cassandra provide different consistency level options to both read and write operations. For write, Consistency.One, means the operation will only be routed to the closet replica node. Consistency.Quorum meaning system will route request to quorum, usually N/2+1, number of nodes and wait for their responses. Consistency.All means it will route request to all N replica nodes and waiting for their response. For read, the operation will be routed to all duplicates, but only will wait specific number of response, others will be received and handled in asynchronous way. For Consistency.One, this number is 1; for Consistency.Quorum, this number is N/2+1 and for Consistency.All this number is N, and so on [17]

In this chapter we will talk about Cassandra Data Model and our architecture setup environment as well -

### *A. Data Model*

Cassandra, which is a distributed key-value organization, and also unlike SQL queries which allow the client to express arbitrarily complex constraints and joining criteria, that only allows data to be queried by its key. Cassandra has adopted abstractions (hidden data) that closely align with the design of

Bigtable [15, 16]. While some of its terminology is similar to those familiar with relational databases, the reader should be careful not to think of how its abstractions map onto Cassandra. The primary units of information in Cassandra parlance are outlined as under. [17]

Column: A column is the atomic unit of information supported by Cassandra and is expressed in the form name value and it is also referred as a tuple (triplet) that contains a name, value and a timestamp. This is the smallest data container.

Row: A Row is the uniquely identifiable data in the system which groups together columns and super columns. Every row in Cassandra is uniquely identifiable by its key. Row keys are important in understanding about Cassandra's distributed hash table implementation.

Column Family: A Column Family is the unit of abstraction containing all the rows of key, which group columns and super columns of highly structured data together. Column families have no defined schema of column names and types supported. All logic regarding data interpretation stays within the application layer. All column names and values are stored as bytes of unlimited size and are usually interpreted as either UTF-8 strings or 64-bit long integer types. Additionally, columns within a column family can be sorted either by UTF-8 encoded name, long integer, timestamp, or using a custom algorithm provided by the application. This sorting criterion is immutable and should be chosen wisely based on the semantics of the application.

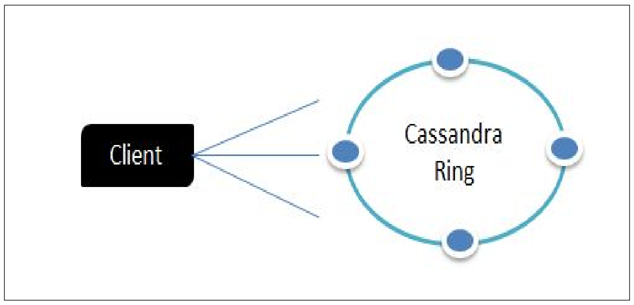
For our experiment we are using CQL (v 2.0) based data model and Datastax Java driver (v 1.0.3) for the Cassandra client [19, 20]

## B. Architecture

Initially we started with Single node Cassandra cluster and then went to four node cluster which has Cassandra 1.2.8 installed in it along with Datastax Opscenter. Also, we have written our client program in Java which is multithreaded program and is using Datastax Java driver as the client. With the use of that client program, we are generating lot of traffic against Cassandra database.

Below is our architecture diagram –

FIGURE 1. ARCHITECTURE DIAGRAM



We wired on the traffic with our client program which is multithreaded against single node Cassandra cluster and monitor the performance of it. Our client program is configurable with number of threads so we can change number of threads accordingly.

In this chapter we will introduce how we can improve scalability in Cassandra database under heavy user loads.

- Instead of installing Cassandra on commodity hardware, install it on SSD drives or much better hardware than HDD's.
- Try to minimize data storage as much as possible. Use better deserialization format such as Smile or Avro or Message Pack instead of plain JSON String or if data readability is not that important, then try to store as a plain Byte Array. This will improve READ/WRITE performance a lot while scaling the Cassandra Database.

## IV. EXPERIMENT

We run series of experiments to evaluate the result. The base Cassandra version we use is 1.2.8 and we are also using Datastax Opscenter as well to monitor the performance of Cassandra. We have set up a 1 node Cassandra cluster on a commodity server; all nodes are within the same LAN and same Rack.

### A. Hardware Types

We have setup single node Cassandra cluster with below CQL data model and used our Client program to generate traffic against Cassandra database. In this test, we have got two machines; one is the machine running HDD and second machine is running SSD Storage. We ran similar test on both of the machines which are running Cassandra database.

Below is our CQL data model in the `dbmsks` keyspace–

```
CREATE TABLE USER_ACTIVITIES (
        USER_ID TEXT,
        LAST_ITEMS_VIEWED TEXT,
        LAST_ITEMS_PURCHASED TEXT,
        LAST_PRODUCTS_PURCHASED TEXT,
        FAVORITE_SEARCHES TEXT,
        FAVORITE_SELLERS TEXT,
        GEOLOCATION TEXT,
        PRIMARY KEY (USER_ID)
);
```

Each column name except USER_ID is represented as a JSON String in this test. LAST_ITEMS_VIEWED is the column which will contain the items being viewed by that user. LAST_ITEMS_PURCHASED is the column which will contain the items purchased by that user. LAST_PRODUCTS_PURCHASED is the column which contains last products purchased by the user. FAVORITE_SEARCHES is the column which contains data about the user favorite searches. FAVORITE_SELLERS is the column which contains the data about its favorite sellers and GEOLOCATION is the column which will contain about its address and zipcode data.

As a sample example, LAST_ITEMS_VIEWED data will look like this which is a simple JSON String which contains the ITEM ID that this user has viewed –

```
final String LAST_ITEMS_VIEWED =
"{\"lv\":[{\"v\":{\"itmId\":1234567890123,\"u
serId\":" + userId +
"}},{\"v\":{\"itmId\":167,\"userId\":" +
userId +
"}},{\"v\":{\"itmId\":1679,\"userId\":" +
userId +
"}},{\"v\":{\"itmId\":1671671679,\"userId\":"
+ userId + "}}],\"lmd\":1366672386876}";
```

And then we have used SMILE on top of this JSON as our Data Serialization format (see section B). In our test, average row length size was 500 - 600 bytes for most of the columns we have.

Replication Factor is setup as 1 and Key Cache is enabled as well. We have setup these settings at the keyspace level (`dbmsks`).

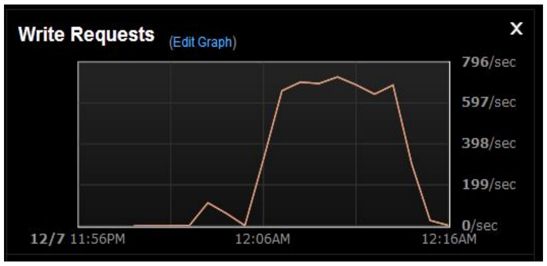
NOTE: - When we ran these tests CPU was idle and no processes were running at that time. We did some dry run before actually started measuring the performance.

TABLE I. HDD VS. SSD IN TERMS OF THROUGHPUT

| Threads | HDD | SSD |
|---|---|---|
| | **Throughput** | **Throughput** |
| 10 | 800 writes /sec | 2000 writes /sec |

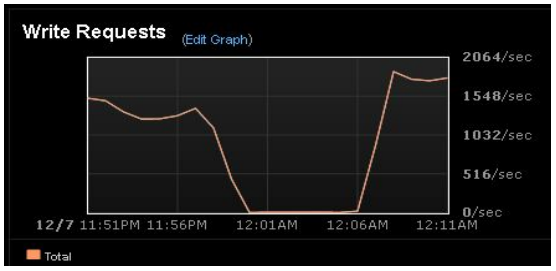
When we ran our client program which is a multithreaded in nature on 10 threads on a machine which is running HDD, we saw maximum throughput as 800 writes per sec and CPU utilization was almost 100% but if we ran the same program on machine which is running SSD's we saw max throughput reached to 2000 writes per second. This clearly shows that if we need to scale Cassandra database under heavy loads we should be installing Cassandra database under a very good hardware not on cheap commodity hardware. Below is the graph which shows the difference –

FIGURE 2. HDD THROUGHPUT



Above graph shows the max throughput we got on the machine which was running HDD's. And after that we tried to increase the load on machine which was running HDD's and suddenly we started seeing 100 % CPU utilization.

FIGURE 3. SSD THROUGHPUT



Now we ran the same program with same configuration and with same set of data on machine which was running SSD's with Cassandra deployed on it and we saw max throughput as 2000 writes per second as shown in the above graph.

Also, we tried to see the write performance of Cassandra running on HDD's as compared with SSD's drive. Again, we have seen SSD's performing way better than HDD's which might help a lot while scaling the Cassandra database under heavy user loads

TABLE II. HDD VS. SSD IN TERMS OF WRITE PERFORMANCE

| Threads | HDD | SSD |
|---|---|---|
| | **Write** | **Write** |
| 10 | 10-15 ms /op | 4-5 ms/op |

Below is the graph for write latency while we were writing into Cassandra database under heavy loads. It clearly shows how the write latency got affected and because of this reason we might need to scale Cassandra database in future. But if we have deployed Cassandra database on SSD drives or some better hardware then this wouldn't be happening at all as the write latency was quite good when we were writing into Cassandra database.

FIGURE 4. HDD LATENCY

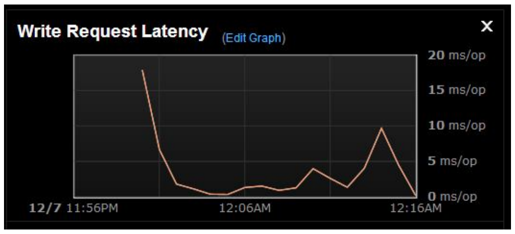
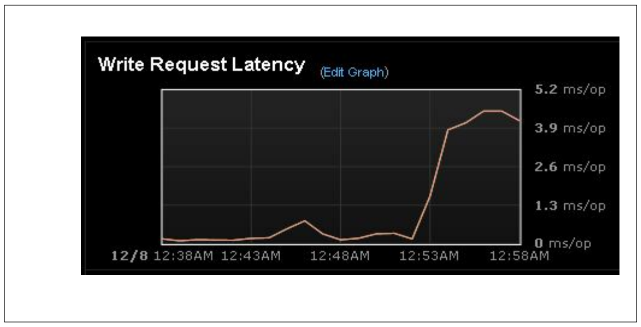


FIGURE 5. SSD LATENCY



Thus with the above tests, we concluded that we saw almost 3x throughput increase if we have deployed Cassandra on a better hardware which will be useful while scaling the Cassandra database.

*B. Data Serialization Format*

In this test, we have tried to figure out which data serialization format we should follow while storing the data into Cassandra so that we don't always run out of space in Cassandra which will also be beneficial in terms of read / write performance while scaling the Cassandra database.

There are several applications that revolve around sending and receiving data over the network. The rapid adaptivity in data exchange over internet has made the selection of a proper data serialization format increasingly important. Data serialization is the process of writing the state of an object to a stream, and is the process of rebuilding the stream back into an object [11]. The two most common modern data serialization formats are eXtensible Markup Language (XML) [12] and JavaScript Object Notation (JSON) [13]. They are both widely used, but were developed before the onset of smart phones. Recently, binary data serialization formats such as Apache Thrift and Protocol Buffers were being used in uplift of these common text-based formats. [27]

While Thrift and PB [18] differ primarily in their scope, Avro and MessagePack is compared in light of the more recent trends: rising popularity of dynamic languages, and JSON over XML. As most every web developers knows, JSON is now ubiquitous, and easy to parse, generate, and read, which explains its popularity. JSON also requires no schema, provides no type checking, and it is a UTF-8 based protocol - in other words, easy to work with, but not very efficient when put on the wire.

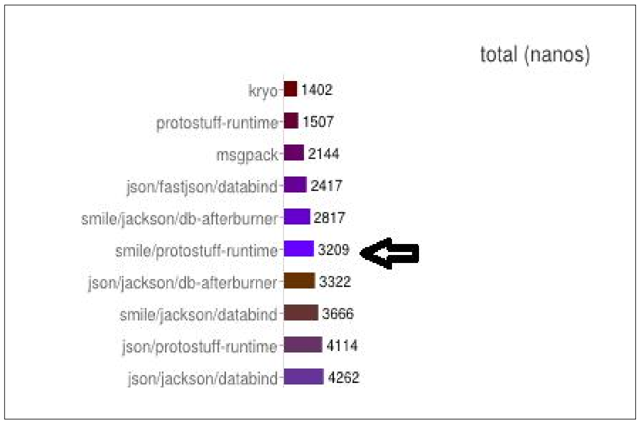
MessagePack is effectively JSON, but with efficient binary encoding. Like JSON, there is no type checking or schemas, which depend on one's application that could be either a pro or a con of it. But, if you are already streaming JSON via an API or using it for storage, then MessagePack could be a drop-in replacement respectively.

"The Best" Serialization Format

Reflecting on the use of Protocol Buffers at Google and all of the above competitors it is clear that there is none definitive, "best" option. Rather, each solution makes perfect sense in the context it was developed and hence the same logic should be applied to your own situation.

If looking for a battle-tested, strongly typed serialization format, then Protocol Buffers is a great choice. If you also need a variety of built-in RPC mechanisms, then Thrift is worth investigating. Finally, if needed strong-typed aspects, but want the flexibility of easy interoperability with dynamic languages, then Avro may be one of the best bet at this point of time. [18]

FIGURE 6. SERIALIZATION FORMAT [31]



For our experimental results, we have used SMILE [31] which is efficient, 100% JSON API compatible binary data format as shown in above image. We were using JSON for our experiment, thus using SMILE on top of that was the best way to minimize the data storage and increase the Cassandra scalability with a throughput. It matches MessagePack speed and produced slightly more compact output as well all Jackson code worked similar to JSON, including JAX-RS providers, datatypes (Guava, Joda etc) and etc., Thus we have not lost the convenience of JSON (except like all binary formats, Smile is not as readable as JSON), but attained the better storage and performance efficiency.

*C. Additional Hardware*

In this test, we will see how easy is to scale Cassandra database under heavy user loads without having any single point of failure. We measure the scalability mostly by throughput in this test. If Cassandra writes cannot keep up with user traffic, then we may see timeout when writing to Cassandra or resource utilization increases on Cassandra side, then it's time to increase Cassandra capacity by adding more nodes to the Cassandra cluster. And in this test we will start with single node Cassandra cluster with very low traffic and then we will gradually increase the traffic and see how the Cassandra is holding up and if Cassandra starts breaking up (as CPU usage was 100 %) because of high traffic, then we will increase the Cassandra capacity by adding more nodes to the cluster without having any single point of failure and without affecting client side code.

"Why" because when we add another node to the Cassandra cluster, any Java Cassandra client will auto discover those nodes automatically and then there won't be any code change as such on the client side.

TABLE III. CASSANDRA SCALIBILITY

| **Threads** | **Cassandra Throughput** | |
|---|---|---|
| | **Single Node** | **Four Node** |
| 40 | 2500 writes / sec | 7000 writes / sec |

Initially, we started with 40 threads from our client program which generated lot of traffic against the Cassandra database and after some time we saw CPU usage at 100% almost (shown in the graph below) and because of that we were not able to take much traffic and got stuck on 2500 throughput and performance also started getting affected along with timeouts as well.

After we expanded the Cassandra cluster to four nodes in the background without affecting the client side code, guess what? We started seeing 3x more throughputs on our Cassandra database which was pretty good and also pretty good performance on the write latency as well. This clearly shows that if the load gets increased for whatever reason, then the CPU usage will be very high and because of that, you won't be able to increase the throughput and you will start seeing pretty bad performance which ultimately means, timeouts happening on the client side and client call will starts taking lot of time which is not we want. Below is our graph –

Figure 7 shows the throughput we got with 40 threads running on single node Cassandra cluster

FIGURE 7. THROUGHPUT (40 THREADS) ON SINGLE NODE CLUSTER

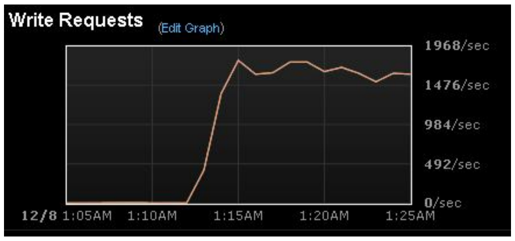


Figure 8 shows that after running for few minutes we started seeing CPU usage as 100% and because of that our throughput got affected and performance started degrading as well. After that we decided lets increase the capacity of our Cassandra cluster by adding couple more nodes and then our throughput started increasing a lot and it touched 7000 writes per seconds (Figure 9) and also performance started getting stabilized as well.

FIGURE 8. CPU USAGE 100%

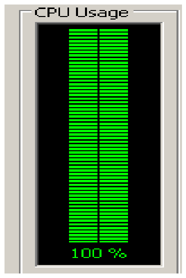
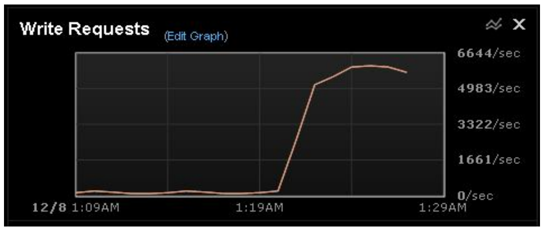


FIGURE 9. THROUGHPUT (40 THREADS) ON FOUR NODE CLUSTER



## V. CONCLUSION AND FUTURE WORK

In this paper we have presented how to improve Cassandra scalability by deploying Cassandra on a very good hardware instead of deploying Cassandra on cheap commodity hardware. We have done several experiments related to this which proves that if Cassandra is deployed on a good hardware then the throughput we get on those machines will be very high ( 3x times) as compared to cheap hardware.

We also have talked about the data serialization format which we should follow while storing the data into Cassandra so that we don't always run out of space in Cassandra which will also be beneficial in terms of read / write performance while scaling the Cassandra database.

For now, we only assume all the nodes are within the same datacenter, we will extend our research to different datacenter in the future. And currently all data has the same replicas number, in the next step; we will think to add additional adaptive replicas for those nodes that contain spot hot data. Also, currently, we have done our experiment on HDD's and SSD's only; in future we would like to expand this to DRAM or SanDisk SSD Storage etc.

## REFERENCES

[1] Dileepa Jayathilake, Charith Sooriaarachchi, ThilokGunawardena, BuddhikaKulasuriya and Thusitha Dayaratne, “A Study Into the Capabilities of NoSQLDatabases in Handling a Highly Heterogeneous Tree”

[2] F. Chang et al., “Bigtable: A distributed storage system for structured data”, In Proc. OSDI, 2006, pp 205–218.

[3] G. DeCandia et al., “Dynamo: amazon’s highly available key-value store”, In Proc. SOSP, 2007, pp. 205-220.

[4] B. F. Cooper et al., “PNUTS: Yahoo!’s Hosted Data Serving Platform”, Proc. VLDB Endow, vol. 1, pp. 1277-1288, August 2008.

[5] A. Lakshman and P. Malik, “Cassandra: Adecentralized structured storage system”, SIGOPS Oper. Syst. Rev. vol. 44, pp. 35-40, 2009.

[6] M. Stonebraker, “SQL Databases v. NoSQL Databases”, Commun. ACM, vol. 53, pp. 10-11, April 2010.

[7] N. Leavitt, “Will NoSQL Databases Live Up to Their Promise?”, Computer, vol. 43, pp. 12-14, February 2010.

[8] E. A. Brewer, “Towards robust distributed systems”, Principles of Distributed Computing, Portland, Oregon, July 2000.

[9] Why NoSQL?[http://www.couchbase.com/why-nosql/nosql-database].

[10] Zhen Ye and Shanping Li, “A request skew aware heterogeneous distributed storage system based on Cassandra”

[11] Marjan Hericko, Matjaz B. Juric, Ivan Rozman, Simon Beloglavec, and Ales Zivkovic, “Object Serialization Analysis and Comparison in Java and .NET”, SIGPLAN No. 38:44–54, August 2003.

[12] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, and Francois Yergeau, “Extensible markup language (XML)” 1.0 (fifth edition), World Wide Web Consortium, Recommendation REC-xml-20081126, November 2008

[13] Json, http://www.json.org. Accessed: 7/26/2011

[14] S. Ghemawat, H. Gobioff and S. Leung, “The Google File System”, In 19th Symposium on Operating Systems Principles, Lake George, New York, 2003, pp. 29–43.

[15] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A Distributed Storage System for Structured Data OSDI’06: Seventh Symposium on Operating System Design and Implementation”, 2006, Seattle, WA, 2006

[16] J. Ellis, et. al., Cassandra Wiki http://wiki.apache.org/cassandra/FrontPage, 2010

[17] Dietrich Featherston, “Cassandra: Principles and Application"

[18] Data Serialization Format Wiki http://www.igvita.com/2011/08/01/protocol-buffers-avro-thrift-messagepack/

[19] CQL; Wiki http://cassandra.apache.org/doc/cql/CQL.html

[20] Datastax Java driver Wiki https://github.com/datastax/java-driver

[21] Networks: Josep M. Pujol, Vijay Erramilli, Member, IEEE, GeorgosSiganos, Xiaoyuan Yang,NikolaosLaoutaris, Member, IEEE, ParminderChhabra, and Pablo Rodriguez, Member, IEEE. "The Little Engine(s) That Could: Scaling OnlineSocial"

[22] Santo Lombardo, Elisabetta Di Nitto and DaniloArdagna, “Issues in Handling Complex Data Structures with NoSQLdatabases”

[23] Cattell, Rick, "Scalable SQL and NoSQL data stores." ACM SIGMOD Record39.4 (2011): 12-27.

[24] Jing Han, Haihong E and Guan Le, “Survey on NoSQLDatabase”

[25] Will NoSQLDatabases Live Upto Their Promise? : Neal Leavitt

[26] Han, Jing ; Song, Meina ; Song, Junde "A Novel Solution of Distributed Memory NoSQL Database for Cloud Computing"

[27] Audie Sumaray and S. Kami Makki, “A Comparison of Data Serialization Formats For Optimal Efficiency on a Mobile Platform”, pp 2-3.

[28] Bagade, P.; Chandra, A.; Dhende, A.B "Designing performance monitoring tool for NoSQL Cassandra distributed database".

[29] ] Okman, L. Deutsche Telekom Labs., Ben-Gurion Univ., Beer-Sheva, Israel Gal-Oz, N. ; Gonen, Y. ; Gudes, E. ; Abramov, J. "Security Issues in NoSQL Databases"

[30] JVM Serializers Wiki https://github.com/eishay/jvm-serializers/wiki