Master Thesis
Electrical Engineering
June 2016

Monitoring and Analysis of CPU Utilization, Disk Throughput and Latency in servers running Cassandra database

An Experimental Investigation

Avinash Goud Chekkilla
CPU Utilization and Read Latency

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering with Emphasis on Telecommunication Systems. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author(s):
Avinash Goud Chekkilla
Email: avch15@student.bth.se

Rajeev Varma Kalidindi
Email: raka15@student.bth.se

External advisor:
Jim Håkansson
Chief Architect,
Ericsson R&D,
Karlskrona, Sweden

University advisor:
Prof. Kurt Tutschku
Department of Communication Systems

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

ABSTRACT

Context Light weight process virtualization has been used in the past e.g., Solaris zones, jails in Free BSD and Linux's containers (LXC). But only since 2013 is there a kernel support for user namespace and process grouping control that make the use of lightweight virtualization interesting to create virtual environments comparable to virtual machines.

Telecom providers have to handle the massive growth of information due to the growing number of customers and devices. Traditional databases are not designed to handle such massive data ballooning. NoSQL databases were developed for this purpose. Cassandra, with its high read and write throughputs, is a popular NoSQL database to handle this kind of data.

Running the database using operating system virtualization or containerization would offer a significant performance gain when compared to that of virtual machines and also gives the benefits of migration, fast boot up and shut down times, lower latency and less use of physical resources of the servers.

Objectives This thesis aims to investigate the trade-off in performance while loading a Cassandra cluster in bare-metal and containerized environments. A detailed study of the effect of loading the cluster in each individual node in terms of Latency, CPU and Disk throughput will be analyzed.

Method: We implement the physical model of the Cassandra cluster based on realistic and commonly used scenarios or database analysis for our experiment. We generate different load cases on the cluster for Bare-Metal and Docker and see the values of CPU utilization, Disk throughput and latency using standard tools like sar and iostat. Statistical analysis (Mean value analysis, higher moment analysis and confidence intervals) are done on measurements on specific interfaces in order to show the reliability of the results.

Results Experimental results show a quantitative analysis of measurements consisting Latency, CPU and Disk throughput while running a Cassandra cluster in Bare Metal and Container Environments. A statistical analysis summarizing the performance of Cassandra cluster while running single Cassandra is surveyed.

Conclusions. With the detailed analysis, the resource utilization of the database was similar in both the bare-metal and container scenarios. From the results the CPU utilization for the bare-metal servers is equivalent in the case of mixed, read and write loads. The latency values inside the container are slightly higher for all the cases. The mean value analysis and higher moment analysis helps us in doing a finer analysis of the results. The confidence intervals calculated show that there is a lot of variation in the disk performance which might be due to compactions happening randomly. Further work can be done by configuring the compaction strategies, memory, read and write rates.

Keywords: Cassandra-stress, NoSQL, Docker, VM, Virtualization, CQL, Bare-Metal, Linux

ACKNOWLEDGEMENTS

A special gratitude to my supervisor, Prof. Kurt Tutschku for giving me an opportunity to work with the thesis under his esteemed guidance, support and encouragement throughout the period of thesis work.

I sincerely thank Jim Håkansson, Christian Andersson, Marcus Olsson and Jan Karlsson at Ericsson R&D Karlskrona and Emiliano, Sogand at Blekinge Institute of Technology for their valuable suggestions and support with respect to experimental setup, academic guidance and time.

I am thankful to my thesis partner Rajeev Varma Kalidindi for his cooperation during the course of the project. Our journey through the completion of the project has been a rather exciting and informative path.

I am very thankful to my Family, Salto Boys, friendly seniors Datta, Manish, Sarat Chandra and my badminton friends Harish and Nanda whose constant support, guidance and motivation was very vital in achieving my goals.

I cannot thank the numerous people who have helped a great deal in pushing us every time and stood by our side throughout this arduous process.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

API	Application Programming Interface
AUFS	Advanced multi-layered Unification Filesystem
Btrfs	B-tree file system
BLOB	Binary Large OBject
CI	Confidence Interval
CPU	Central Processing Unit
CLI	Command Line Interface
CQL	Cassandra Query Language
DTCS	Data Tiered Compaction Strategy
IaaS	Infrastructure as a Service
I/O	Input/Output
IPC	Inter Process Communication
IT	Information Technology
LCS	Leveled Compaction Strategy
LXC	Linux Containers
MNT	Mount
NET	Networking
NoSQL	Not Only SQL
OS	Operating System
PaaS	Platform as a Service
PID	Process Identifier
SaaS	Software as a Service
SSTable	Sorted Strings Table
STCS	Size Tiered Compaction Strategy
TTL	Time To Live
TV	Television
UTS	Unix Timesharing System
VFS	Virtual File System
VM	Virtual Machine
VMM	Virtual Machine Monitor

1 INTRODUCTION

This chapter initially describes a brief description about need for NoSQL databases and cloud technologies in telecom. Successively, an introduction to Containers which are seen as a viable alternative to virtual machines to handle these enormous data in cloud environment.

The amount of information in digital form kept growing massively since 2000's because of digital transformation in the form of media – voice, TV, radio, print which mark the transformation from analog to digital world.

In order to handle such data, NoSQL databases are used [1]. Human assisted control of big data platform is unrealistic and there is a growing demand for autonomic solution [2]. Cassandra with respect to scalability is superior to other databases while at the same time maintaining continuous availability, Data location independence, Fault Tolerant, Decentralized and Elastic features.

Also, to keep up with this expansion of data and successive demand to rise the capability of data centers, run cloud services, consolidate server infrastructure and provide simpler and more affordable solutions for high availability, Virtualization Technologies which are hardware independent, isolated, secure user environments are being utilized by IT organizations.

Virtual Machines are widely used in cloud computing, specifically IaaS. Cloud Platforms like Amazon make VMs accessible and also execute services like databases inside VMs. PaaS and SaaS are built on IaaS with all their workloads executing on VMs. As virtually, all cloud workloads are presently running in VMs, VM performance has been a key element of overall cloud performance. Once an overhead is added by the hypervisor, no higher layer can remove it. Such overheads have been an inescapable tax on cloud workload performance.

Although the virtualization technology is mature, there are quite a few challenges in performance due to the overhead created by the guest OS. Containers with less overhead and fast startup and shutdown times are seen as a viable alternative in Big data applications that use NoSQL distributed storage systems. Containers run as a well isolated application within the host operating system. Containers play a vital role when Speed and flexibility, New workloads and quick deployment is a major consideration.

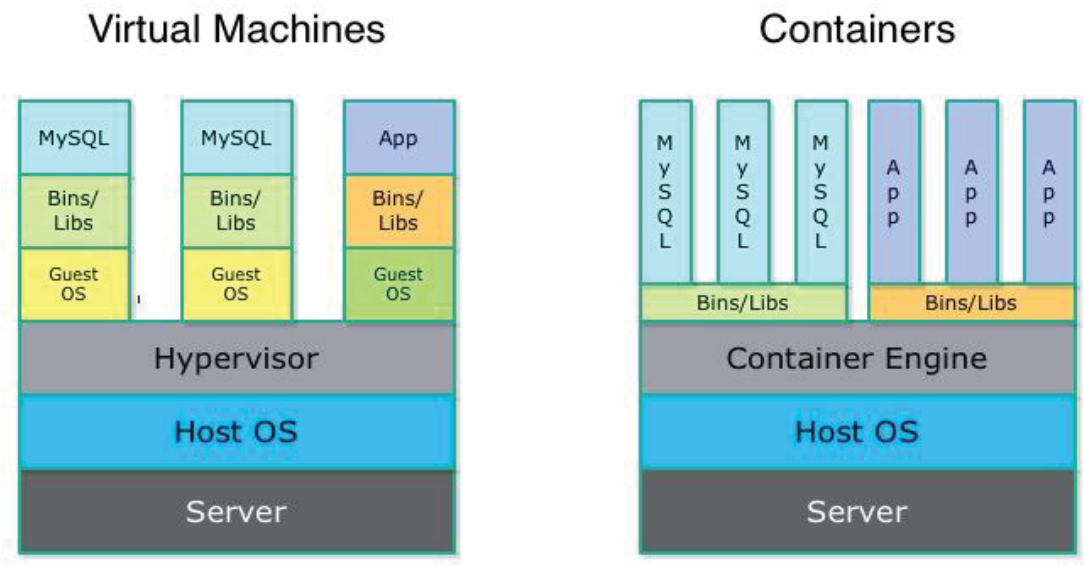

Figure 1:1Hypervisor vs Container Infrastructure [3]

This thesis mainly focuses on the implementation and performance evaluation of Cassandra which is a NoSQL database in bare-metal and containers. The performance of the database with necessary configurations in a bare-metal server is evaluated. Having the same configuration for containers, we do the performance testing of the database in containers. Finally, a trade-off in the performance while running the database in bare-metal and containers is observed and analyzed.

1.1 Motivation

The growth of vast amount of data specially because of digital transformation in the past few years paved a need to go for NoSQL database technologies that were intended to handle the demands of present day modern applications in the field of IT which couldn't be handled by traditional RDBMS which are not that dynamic and flexible. Running these databases in cloud makes it cost effective because of the advantages of reduced overhead, rapid provisioning, flexibility and scalability. As virtually all workloads run inside VM, its performance effects the overall cloud performance.

IT organizations place a problem with the significant growth of data and methods to handle it. Cloud computing which uses virtualization can be seen as a solution. But there is overhead because of the guest OS in the performance of the VM.

Containers which are light weight VM, can be seen as a viable alternative because of their advantage in avoiding the overhead created by guest OS in VM in running the Cloud.

1.2 Problem Statement

Virtualization Technology is an effective way for optimizing cloud infrastructure. However, there is an inherent problem in the overall performance of the cloud while running applications handling Big data inside the VM because of the overhead induced by the Guest OS. Container-based Virtualization provide a different level of abstraction in terms of virtualization and isolation. While hypervisor abstract the hardware and need a full OS which runs on the VM instance in each VM that results in overhead in terms of virtualizing the hardware and virtual device drivers, containers implement the isolation of processes at operating system level, thereby avoiding the overhead. These containers run on top of the kernel of underlying Host Machine. This advantage of running containers because of the

shared kernel makes it to achieve higher density in terms of disk images and virtualized instances with respect to hypervisors. By identifying and studying the performance of the overhead in both Bare Metal and Container environments in terms of CPU, Disk Utilizations and Latency, deeper understanding of the resource sharing in both environments, as well as better optimizations can be achieved. This can pave the way for the usage of containers or hypervisor based virtualization or both in optimizing cloud infrastructure in order to handle Big Data.

1.3 Aim of the Thesis

The aim of this thesis is to investigate the impact of using containers on the performance while running NoSQL systems for telecommunication applications processing large amount of data. Initially a NoSQL data base system is implemented by forming a Cassandra cluster on Bare Metal and then on Container environment. After that a comparison between Bare Metal and Containers which is a lightweight process level virtualization is studied while stressing the Cassandra cluster with load and then by using performance metrics like CPU Utilization, Disk Utilization and Latency on individual nodes in the cluster.

The cloud environment considered for this thesis comprises of Apache-Cassandra 3.3 deployed on Ubuntu 14.04 LTS and Docker version 1.2 and API version 1.23. The reason for choosing these are importantly, they are free open source and easy to deploy. Cassandra has been chosen among various other available NoSQL databases because of its reputation for fast write and read performance, and deliverability of true linear scale performance in a masterless, scale-out design [4].

Docker has been chosen among several management tools available for linux containers due to its feature set and ease of use and has become a standard management tool and image format for containers specially because of AUFS (Another Union FS) that provide a layered stack of filesystems which allow to reuse layers between containers that helps in reduction of space usage and simplifies the filesystem management [5].

1.4 Research Questions

Some of the research questions pertaining to this thesis are as follows:

- What is the methodology for analyzing the performance of databases in virtual machines?
- What is the CPU utilization of the server while running the database in bare-metal case and in containers?
- How does this performance vary with different load scenarios?

1.5 Split of work

Our areas of interest when understanding the resources utilization of the servers are the CPU utilization, disk throughput and latency of operations while running the cassandra database.

- Monitoring and analysis on disk utilization of the server while running the database and latency of the operations are performed by Rajeev.
- Monitoring and analysis of CPU utilization of the server while running the database is performed by Avinash.
- Both of us take the same load scenarios of maximum load and 66% of maximum load and evaluate the database on mixed load operation of 3 reads and 1 write.

- Introduction to Cassandra in telecom is written by Avinash. The motivation, problem statement and research questions are written by both Avinash and Rajeev.
- Both Rajeev and Avinash have together worked on the related work section
- Avinash has worked on the technological overview of the NoSQL systems and Cassandra database. Rajeev has worked on the Virtualization section.
- System analysis methodologies section. Different ways to study a system and methodology for analyzing a system are written by Rajeev. Statistical analysis and ways to study a database system are done by Avinash
- In the results section, the analysis for disk throughput and latency values are evaluated by Rajeev and CPU utilization is analyzed by Avinash
- Individual conclusions and general conclusion of the thesis is presented in the conclusion section

1.6 Contribution

The primary contribution of this thesis is the insight it provides into the understanding of the working of Linux environment, Light weight virtualization, Cassandra and Docker containers.

Secondly, the thesis gives an understanding on the performance of each load type in terms of latency, cpu and disk utilizations in Bare metal and Docker container environments when affected by external load onto the Cassandra cluster.

Finally, Quantitative analysis of resource sharing in bare metal and docker environments from the obtained results in graphical representation is shown. This helps in understanding the Resource Sharing in Bare Metals and Light weight virtualization platform (Container) environments in a greater detail that can lead in optimizing cloud performance.

1.7 Thesis Outline

Chapter 1 provides an overview and background of the research area. It also provides the motivation for this thesis, the problem statement, research questions and contribution of this work. Chapter 2 explains the technologies involved in this work. It gives an account of NoSQL databases, Virtualization, Apache-Cassandra and Docker Containers. Chapter 3 deals with related research work to this thesis. Chapter 4 presents the methodology employed for the execution of this thesis work. It explains the evaluation technique used, metrics and factors considered for the evaluation. Furthermore, it illustrates the design and scenarios of the experiment with a detailed account of the experiment setup. Chapter 5 presents the results, and a detailed analysis of the obtained results of the experiment. This includes the performance metric values of latency, cpu and disk utilizations on individual node while stressing the cluster with a load generator for different workloads and their measurements in graphical manner along with their analysis. Chapter 6 includes conclusions derived from the experiments. It also presents a way for future research work.

2 TECHNOLOGY OVERVIEW

This chapter gives an explicative overview about the concepts of Virtualization, Apache-Cassandra, Docker Containers that are used in the experimentation. The objective of this chapter is to give an insight about these technologies involved.

2.1 NoSQL

Not only SQL (NoSQL) refers to progressive data management engines that were developed in response to the demands presented by modern business applications, such as scaling, being available always and very quick. It uses a very flexible data model that is horizontally scalable with distributed architectures.

NoSQL databases provide superior performance, more scalable and address problems that couldn't be addressed by relational databases by:

- Handling large volumes of new, rapidly changing structured, semi-structured, and unstructured data.
- Working in Agile sprints, iterating quickly and pushing the code every week or sometimes multiple times a day.
- Using Geographically distributed scale-our architecture instead of using monolithic, expensive one.
- Using easy and flexible Object Oriented Programming.

NoSQL provides this performance because it incorporates the following features:

1. **Dynamic Schemas**

   NoSQL databases can have data insertion without any predefined schema, which makes it easy to make significant application changes in real work environments. This makes it faster, reliable code integration.

2. **Auto-sharding**

   NoSQL databases natively and automatically spread data across an arbitrary number of servers, without the application requiring to know the composition of server pool. Data and query load are balanced automatically. A server can be transparently and quickly replaced with no disruption in the application when a server goes down.

3. **Replication**

   NoSQL databases support automatic database replication to maintain the availability in the case of event failure or planned maintenance events. Few NoSQL databases offer automated failover and recovery, self-healing features, as well as the ability to geographically distribute database across various locations in order to withstand regional failures and to enable data localization.

4. **Integrated Caching**

   Many NoSQL databases have integrated caching feature, keeping recurrently used data as much as possible in the system memory and thereby removing the need for a separate catching layer. Few databases have fully managed, integrated in-memory database management layer for workloads requiring high throughput and low latency.

Typically, NoSQL datatypes can be classified based on any of the four Data Models:

a) **Document Model**

They pair each key with a complex data structure known as a document whose structure is in JSON (JavaScript Object Notation). The schema is dynamic which allows a document to have different fields. It makes it easy to

add new fields during development of an application. Documents can contain many key-value or key-array pairs or even nested documents. It has the broadest applications due to its flexibility, ability to query on any field and natural mapping of the data model to objects in modern programming languages.
Examples: MangoDB and CouchDB

b) Graph Model

It uses graphical structures with nodes, edges and properties to represent data. Data is modeled as a network of relationships between specific elements. It is useful for cases in which traversing relationships are applications core, like navigating through networks, social network connections, supply chains.
Examples: Giraph and Neo4j

c) Key-Value Model

Key-Value Stores is one of the most basic type of non-relational database type. Every single item is stored in the database as an attribute name, or key, together with its value. Data can only be queried with the use of the key. It is a very useful model in case of representing polymorphic and unstructured data, as the database doesn't enforce a set schema across the key-value pairs. Some key-value stores allow each value to have a type, like 'integer' that's adds functionality.
Example: Riak, Redis and Berkeley DB

d) Wide Column Model

Wide column stores use a sparse, distributed multi-dimensional sorted map for data storage. Every record can vary with the number of columns stored. Columns can be grouped to form column families, or can be spread across multiple column families. Data is retrieved with the use of primary key per column family.
Example: HBase and Cassandra

2.1.1 Cassandra

Cassandra is a distributed NoSQL database for managing large amounts of structured, semi-structured, and unstructured data across multiple data centers and the cloud. Cassandra delivers continuous availability, linear scalability and its simplicity in operation across many servers with no single point of failure, along with a powerful dynamic data model which is designed to achieve flexibility and low latency which enables fast response times.

Cassandra built-for-scale architecture enables to achieve massive volumes of data handling, higher concurrent users/operations per second, fast write and read performance, and deliverability of true linear scale performance in a masterless, scale-out design compared to other NoSQL databases.

Cassandra consists of a cluster of nodes, where each node is an independent data store. A node is independent (It can be a server or a VM in the cloud). Each node in Cassandra is responsible for a part of the overall database. It writes copies of data on different nodes so as to avoid any single point of failure. Replication factor is used to set the number of copies of data in the node. A replication strategy is used to replicate data across multiple servers and data centers. In Cassandra, all nodes play equal roles; with nodes communicating with each other equally. There is no master node so there is no single point of failure and all the data has copies in other nodes which secures the data stored. It is capable of handling large amounts of data and thousands of concurrent users or operations per second across multiple data centers.

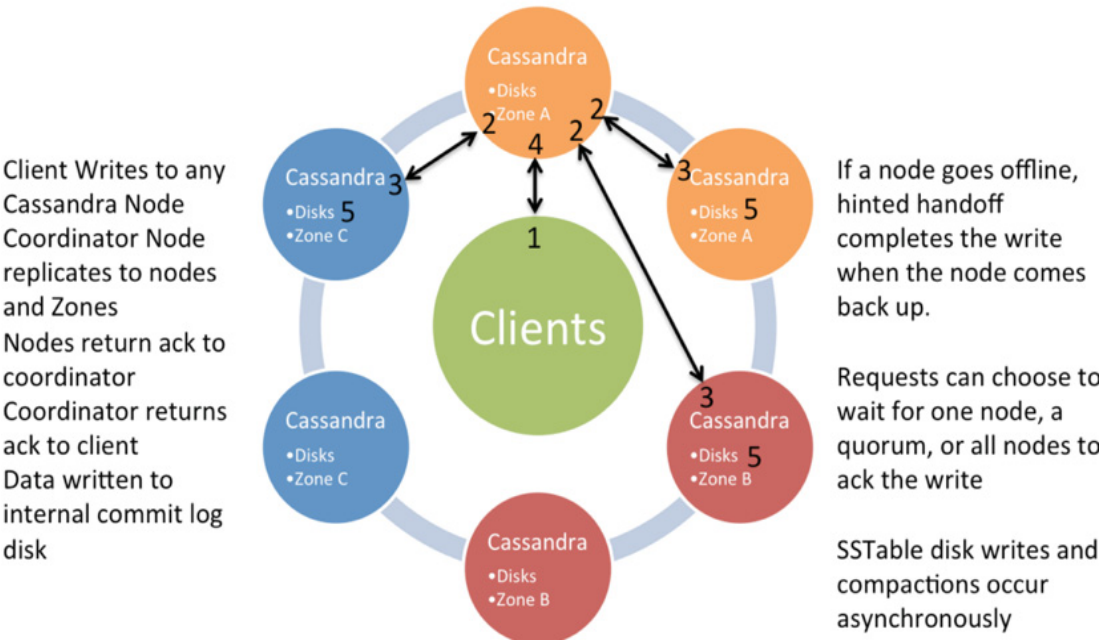

Figure 2:1Cassandra Architecture[6]

2.1.2 Cassandra Architecture

The dataflow of Cassandra operation is detailed in this section to give an overview of Cassandra operation. It's hardware is based on the understanding that system hardware failures can and do occur. Cassandra address these failures by maintaining a peer-to-peer distributed system across nodes among which data is distributed in the cluster. Every node contains either all or some parts of the data present in the Cassandra system. Each node exchanges information across the cluster or to any other node in the cluster every second depending on the consistency level configured. When data is entered into the cluster which is a write operation, it goes to a local persistent called Commit Log, which maintains all the logs regarding the data in Cassandra.

2.1.3 Cassandra Data Insertion

Cassandra processes data at several stages on the write path, starting with the immediate logging of a write operation till its compaction:

a) Logging writes and memtable storage

When data enters into Cassandra, which is a write operation, it stores the data in a structure in memory, the memTable, which is a write-back cache of the data partitions, which Cassandra looks with the use of a key, and also updates to the commit log on disk, enabling configurable durability. The commit log is updated for every write operation made to the node, and these durable writes survive permanently even in case of power failures.

b) Flushing data from memtable

When the contents in a memtable, which includes indexes exceed a configurable threshold, it is put in a queue to be flushed to disk. The length of the queue can be configured using the memtable_flush_queue_size option in the cassandra.yaml file. If the data which is to be flushed is more than the queue size, Cassandra stops write operations until the next flush succeeds. Memtables are sorted by token and then written to disk.

c) Storing data on disk in SSTables

When the memtables are filled, data is flushed in sorted order into the SStables (sorted string tables). Data in the commit log is purged after its corresponding data

in memtable is flushed to an SSTable. All writes are automatically partitioned and replicated throughout the cluster.

Cassandra creates the following structure for each SSTable:

- Partition Index, which contains a list of partition keys and positions of starting points of rows in the data file.
- Partition summary, which is a sample of the partition index.
- Bloom filter which is used to find out the SSTable which most likely contains the key, which is an off-heap structure associated with each SSTable that checks if any data for the requested row exists in the SSTable before doing any disk I/O.

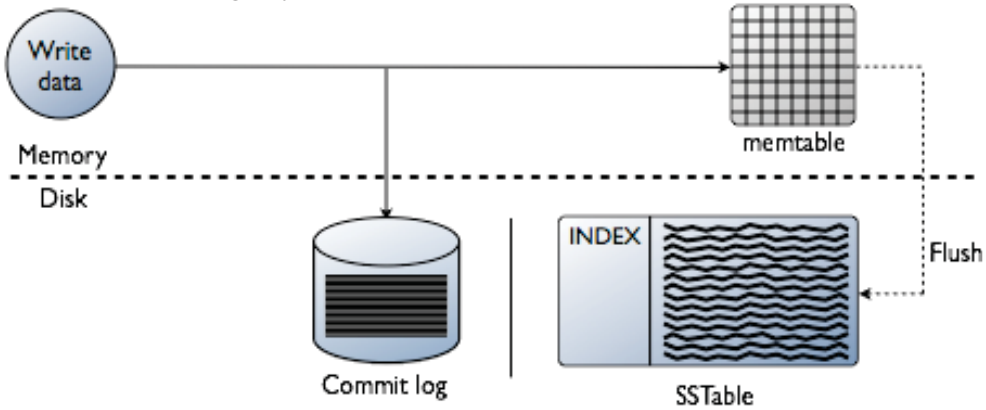

Figure 2:2 Cassandra Data Insertion [7]

d) Compaction

Cassandra using compaction, periodically consolidates SSTables, discarding obsolete data and tombstone, which is a marker in a row that indicates a column was deleted and exists for a configured time defined by gc_grace_seconds value set on the table. During compaction, marked columns are deleted. Periodic compaction is essential as Cassandra doesn't insert or update in place. Cassandra creates a new timestamped version of inserted or updates data in another SSTable.

Compaction process is depicted in below Figure 2.3.

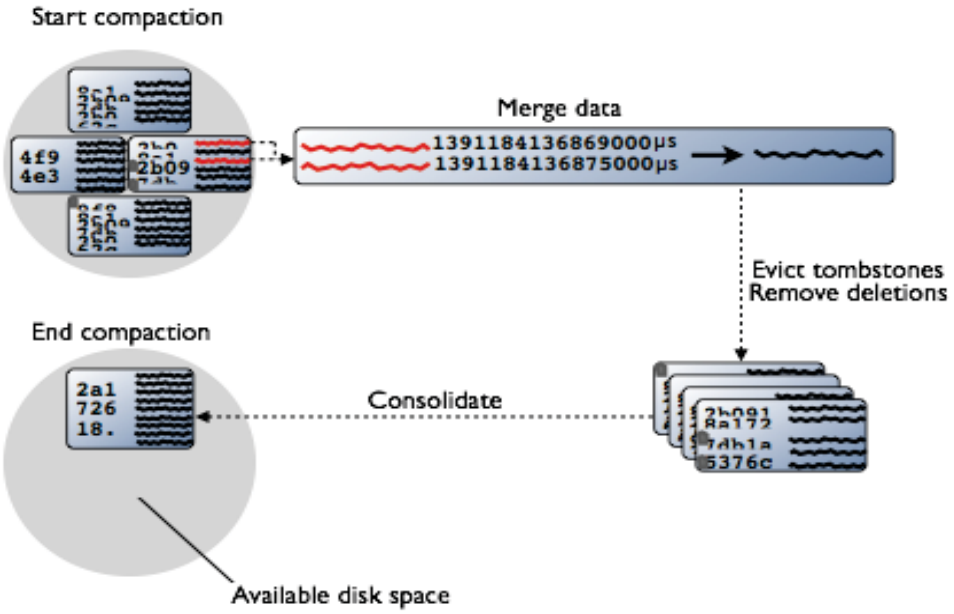

Figure 2:3 Cassandra Compaction Process [7]

Compaction uses partition key and merges data in each SSTable. It selects the latest data for storage by using its timestamp.
Cassandra can merge the data, without any random IO with the help of partition key within each SSTable that enables rows to be sorted. Subsequently evicting tombstones and removing deleted data, columns and rows, SSTables are merged together into a single file in this compaction process.

Old SSTables are deleted once the last reads finish using the files, which enables in creating new disk space available for use. Cassandra with its newly built up SSTable helps in handling more read requests efficiently than before compaction.

Withstanding no random I/O occurs, compaction can be considered as a heavyweight operation. When the old and new SSTables co-exist, there is a spike in the disk space usage during compaction. To minimize the read speeds, compaction runs in the background.

To reduce the impact of compaction on application requests, Cassandra does the following operations:

- Controls compaction I/O using compaction_throughput_mb_per_sec (default 16MB/s)
- Requests OS to pull latest compacted partitions into page cache

Compactions that can be configured and designed for to run periodically are:

- Size Tiered Compaction Strategy (STCS) for write-intensive workloads
- Date Tired Compaction Strategy (DTCS) for time-series and expiring data
- Leveled Compaction Strategy (LCS) for read-intensive workloads

2.1.4 Cassandra Read Operation

Cassandra fundamentally incorporates results from potentially multiple SSTables and active memtables to serve a read. When a read request is raised, it checks the bloom filter. Every SSTable has a bloom filter that is incorporated with it that inspects the probability of having any data for the partition requested in the SSTable before proceeding to do any disk I/O.

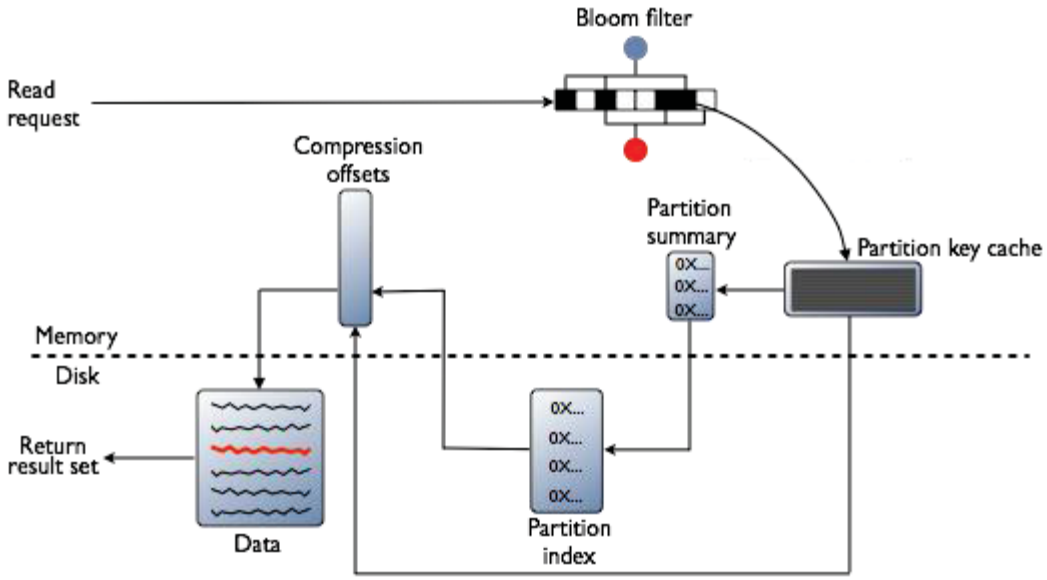

Figure 2:4 Cassandra Read Path [8]

If the Bloom filter doesn't rule out the SSTable, Cassandra reviews the partition key cache, which contains the partition index for a Cassandra table and successively exhibits one of the following actions based on finding of the index entry in the cache:

1. If index entry is found in cache:
    a) Cassandra finds the compressed block having the data by going to the compression offset map.
    b) It returns the result set by fetching the compressed data.

2. If index entry is missing in cache:
    a) Cassandra determines the index entry approximate location on disk by searching the partition summary.
    b) Later, to search the index entry, Cassandra hits the disk, and performs a singles seek, sequential column reads in the SSTable if the columns are adjoining.
    c) Cassandra finds the compressed block having data by going to the compression offset map like in previous case.
    c) Finally, it returns the result set by fetching the compressed data.

2.1.5 Data Deletion

Data in Cassandra column has TTL (Time to live), which is an optional expiration date that can be set by using CQL. Once the requested amount of time expires the TTL data is marked with a tombstone whose period is set by gc_grace_seconds. Tombstones are automatically deleted during compaction process.

Running a node repair is essential if there is any node failure during compaction process as the node down will not be having the delete information sent by Cassandra after gc_grace_seconds and may lead to appearance of deleted data.

2.1.6 Hinted Handoff

It is a unique feature of Cassandra that helps in optimizing the consistency process in a cluster when a replica-owning node is unavailable to accept a successful write request.

When hinted handoff is enabled during a write operation, a hint that indicates a write needs to be replayed to one or multiple nodes about dead replicas is stored by the coordinator in the local system hint tables for either of these cases:

a) A replica node for the row is known to be down before time.
b) A replica node doesn't respond to write request

A hint consists of:

- Location of the replica that is down
- Actual Data written
- Version metadata

Once a node discovers that the node it holds hints for is up, it sends data row corresponding to each hint to the target.

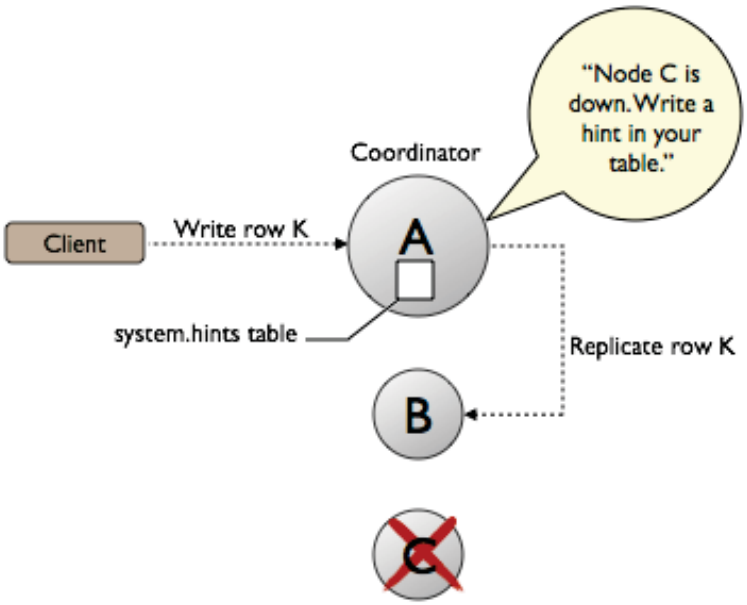

Figure 2:5 Cassandra Hinted Handoff [9]

Consider a three node cluster A,B and C, each row is stored on two nodes in a keyspace with replication factor of 2. When node C is down and a write request is raised from client to write row K, coordinator write a hint for node C and replicates data to node B.

When client specified consistency level is not met, Cassandra doesn't store a hint.

2.1.7 Consistency

It refers to the process of updating and synchronizing Casandra on all its replicas in the cluster. Cassandra extends eventual consistency, that is if the data item has no new updates, eventually all those who access the data item will return the last updated value by using the concept of tunable consistency.

Cassandra can choose between strong and eventual consistency based on the need. The number of replicas that need to acknowledge the write request to the client application is determined by the write consistency level and the number of replicas that must respond read request before returning data to client application is specified by read consistency level. Consistency levels can be set globally or on a per-operation basis.

Few of the most used consistency levels are stated below:

a. **ONE**
   A response from one of the replica nodes is sufficient
b. **Quorum**
   A response from a quorum of replicas from any data center. The quorum value is found from the replication factor by using the formula
   $$\text{Quorum} = (\text{Replication Factor}/2) + 1$$
c. **All**
   All the replicas have to respond

2.2 Virtualization and container-based virtualization

2.2.1 Virtualization

Virtualization refers to the act of creating a virtual version of something. It is used as a technique for portioning or dividing resources of a single server into multiple separated

and isolated environments. The result is the creation of logical units called virtual machines. The virtual machines are given access to the hardware resources and controlled by a software called Virtual Machine Monitor (VMM) or hypervisor

It aims to make the most use of the server hardware resources. It creates multiple virtual machines (VMs) in a single physical machine. Virtualization makes use of hypervisors to create virtual machines. The server is the host and the created virtual machines are known as guests on the server. Each virtual machine has a part of the host server's resources (like CPU, disk, memory, I/0 etc.) which are handled by the hypervisor.

2.2.2 Container based virtualization

Container-based virtualization differs from virtualization in the traditional sense that it does not have an operating system of its own but uses the kernel features such as namespaces and cgroups to provide an isolated layer. Operating system is virtualized while the kernel is shared among the instances. A separate guest OS is not needed in the container case as the instances isolated by the container engine share the same kernel with the host OS.

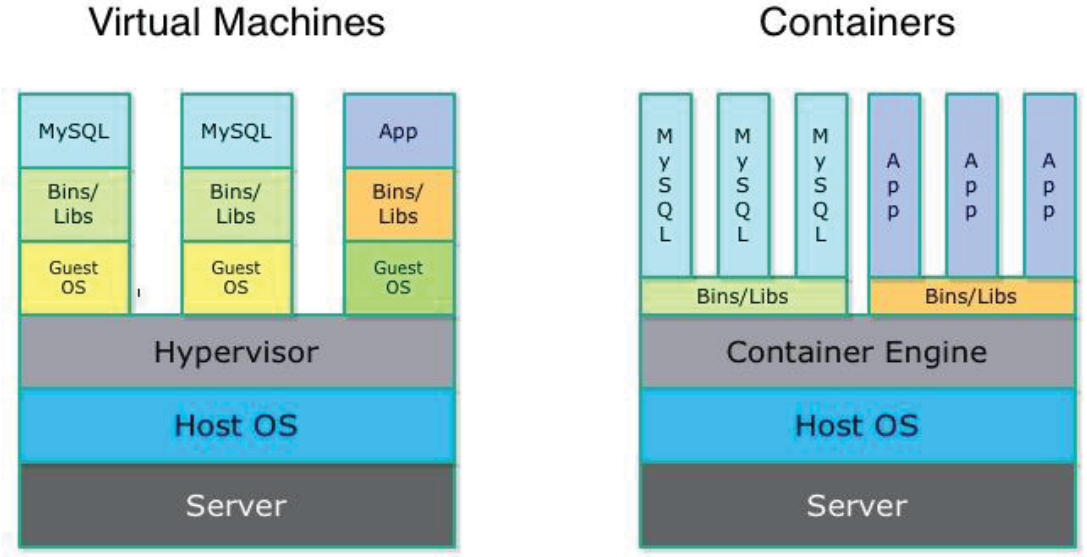

Figure 2:6 Virtual Machine vs Containers [3]

2.2.3 Container based virtualization vs Traditional virtualization

Table 2:1 Virtual Machines vs Containers

Virtual Machines	Containers
Represent Hardware-level virtualization	Represent operating system virtualization
They can be run on any operating system guest OS	They work only on Linux based systems
They are heavyweight because they run operating systems	They are lightweight
Slow provisioning	Fast provisioning and scalability
Fully isolated and hence more secure	Process level isolation and hence less secure

2.2.4 Docker

Docker is an open platform for developers and system administrators to build, ship and run distributed applications with the help of Docker Engine, which is a portable, light and run-time and packing tool and Docker Hub, that is like a cloud based registry service which

helps to link to code repositories, build own images and test them, store the manually pushed images to Docker Cloud that enables deploying images to local hosts.[10]

2.2.4.1 Why Docker?

Docker helps in isolation of infrastructure from applications, that enables treating infrastructure as managed application. Docker aids to ship, test, deploy code rapidly thereby shortening the time between writing code and running code. Kernel Containerization features with workflows and tooling enable docker in managing and deploying applications.

2.2.4.2 Faster Delivery of applications

Docker is optimum in aiding with the development lifecycle. Docker allows developers to develop on local containers that contain applications and service and later integrate everything into a continuous integration and deployment workflow.

2.2.4.3 Deploying and Scaling

High portable workloads can be handled because of its container based platform which also enable it to run on local host, physical or virtual machines in a data center, or in the cloud.

Dynamical Management of workloads is possible because of Docker's portability and lightweight nature. Scaling up and tearing down applications and services is quicker in docker which enables scaling to be almost real time.

2.2.4.4 Higher Density and Multiple Work Load Applications

Because of its lightweight and fast nature, docker provides a cost-effective, viable alternative to hypervisor-based virtualization which enhances its usage in high density environments: for example, in building a Platform-as-a-service PAAS or a own cloud. It is also effective in usage for small and medium size deployments where one wants to get more of the limited resources possessed.

2.2.5 Docker Platform

Docker platform helps in running applications in an isolated and secured way. Few features of docker platform are described below:

- Getting and supporting components of applications in docker containers.
- Distribute and ship the applications for testing and further development processes.
- Deploy Applications directly in production environment, if it is a cloud or a local data center.

2.2.6 Docker Engine

Docker Engine is a client-server application with three major components:

- Docker Daemon - A server which is a type of long running program
- REST API - An Interface that programs can utilize in order to communicate and instruct the daemon.
- CLI – A Command Line Interface client

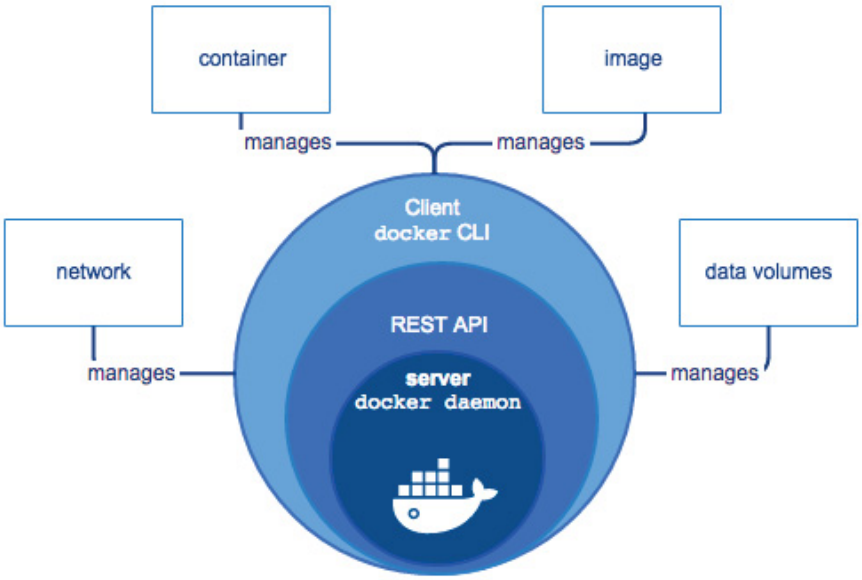

Figure 2:7 Docker Engine Architecture [11]

Docker REST API helps docker CLI to interact with docker daemon with the usage of CLI commands or scripting.

Docker Daemon creates and manages Docker objects which include images, containers, networks, data volumes and so on.

2.2.7 Docker Architecture

Docker adapts a client-server architecture. The docker client communicates with docker daemon which performs operations like building, running and distributing docker containers. Docker Client and Docker Daemon can run on the same system or one can connect docker client to daemon on a different system. Sockets and RESTful API help in communication between docker client and docker daemon.

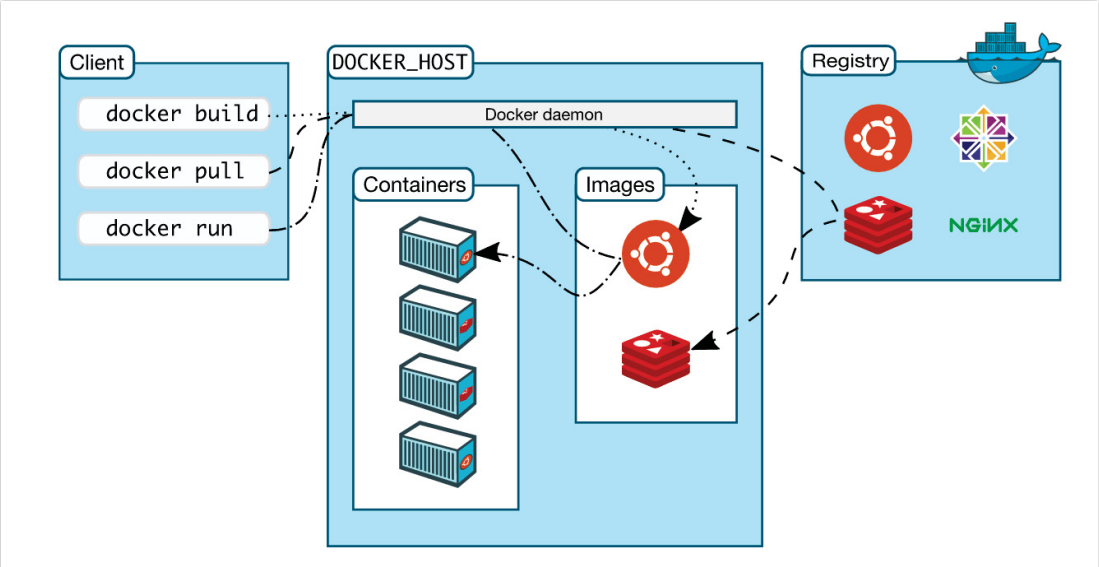

Figure 2:8 Docker Architecture [12]

2.2.7.1 Docker Daemon

Docker Daemon runs on a host machine and the user interacts with the daemon via docker client and not directly.

2.2.7.2 Docker Client

Docker client is the primary interface to docker which communicates with docker daemon by accepting commands from the user.

2.2.7.3 Docker Internals

Docker Internals require and understanding of the following three docker resources:

a) Docker Images

A docker image is a read-only template and is a *build* component of docker that is used to create docker containers. Docker provides an easy way to build new images or update existing images or download images that others have created.

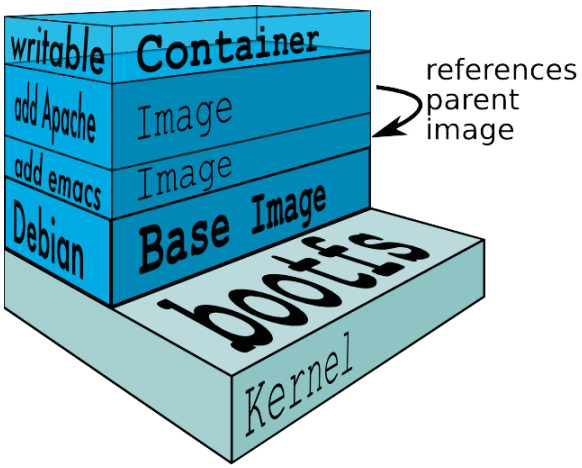

Figure 2:9 Docker Image [13]

b) Docker Registries

Docker Registries are the *distribution* component of docker and are public or private stores of docker images from which one can download or upload docker images. Docker Hub is the public registry which has a collection of existing images for use. There can be images one can create oneself or use previously built images.

c) Docker Containers

Docker containers are the *run* component of docker and are created from docker images and contain binaries, libraries, file system i.e. everything that is necessary to run an application. Docker containers can be run, stopped, moved and deleted whenever necessary. Each container is an isolated and secure application platform.

2.2.7.4 Docker Image Operation

Docker image are read only images and consist of a series of layers which are combined with the help of union file system to form a single image. Union file systems allow

files and directories of separate file systems, known as branches, to be transparently overlaid, forming a coherent file system.

The lightweight nature of docker is because of these layers. When a change is made in a docker image such as an update in an application, a new layer gets built. This addition or updating of layer instead of replacing or rebuilding the whole image, distribution of update instead of a new images makes it feasible and viable for Docker to be preferred than Virtual Machines.

Every image is based on a base image on which other binaries and images called layers are added up to form an image. Base Images are used to build docker images using instructions which is a descriptive set of steps, with each instruction adding a new layer in the image. Instructions are used to create actions like the following for example:

- Run a command
- Add file or directory
- Create an environment variable
- Scheduling processes when launching a container from its image

All the instructions are stored in a Docker file, which is a text based script that contains instructions and commands for building an image from the base image. When building a new image from an existing base image, Docker reads the instructions from Docker file.

2.2.7.5 Docker Registry Operation

Docker Registry is a place to store docker images. Once an image is built it can be pushed to Docker Hub which is a public registry or our own private registry. Docker client helps in looking for prebuilt images in registry and pulling them to docker host in order to build docker containers based on the pulled image.

2.2.7.6 Docker Container Operation

Containers contain Operating system, user-added files, and meta data. Containers are built from an image which tells docker what the container contains, instruction set, and configuration data. When a container is run, docker adds a read-write layer on top of the images using Union File System in which the application can run.

Docker container is run by Docker daemon, upon being instructed by docker client that uses Docker binary or API. An example of running a container built on Ubuntu base image in bash mode is shown below

```
$ docker run -it Ubuntu /bin/bash
```

Docker binary launches the Docker Engine client. Option run is used to run a new container. In order for Docker Client to instruct Docker Daemon, it has few minimum requirements that include:

1. Docker Image from which the container is built. Ex: Ubuntu
2. Commands to run inside the container upon launch. Ex: /bin/bash

Docker Engine does the following when the above command is run:

- **Pulls Docker Image**: Docker Engine checks if Ubuntu image is present in the repository. If the image exists it uses for building a new container or else, it pulls from Docker Hub into the Docker Host.
- **Creates New Container**: Once Docker Engine has the image, it is used to create the container.

- **Allocate File system and mount Read-Write layer**: Container is created in file system and a read-write layer is added to the image.
- **Allocate Network/Bridge interface**: A network interface that allows Docker container to interact with local host is created.
- **Sets up an IP address:** Finds and allots IP address from a pool.
- **Executes a specified process:** Runs application, and;
- **Captures and provides application output:** Connects and logs standard input, outputs and errors to see how application runs.

This now creates a running container. Now one can manage the container in terminal, interactive modes, interact with the application, and upon finishing usage stop the container and remove.

2.2.8 Underlying Technology

Docker is built on the following underlying technologies.

1. Name Spaces

Docker uses namespaces to provide isolated workspaces. When a container is run, docker creates a set of namespaces for that container. This provides an isolation layer i.e. each aspect of container runs its own namespace and doesn't have external access outside it. Few commonly used namespaces for Linux are as follows:

- **pid namespace:** Process Isolation (PID: Process ID)
- **net namespace:** Managing network interfaces (NET: Networking)
- **ipc namespace:** Managing access to IPC resources (IPC: Inter Process Communication)
- **mnt namespace:** Managing mount-points (MNT: Mount)
- **uts namespace:** Isolating kernel and version identifiers. (UTS: Unix Timesharing System).

2. Control Groups

Docker Engine uses cgroups or control groups to share available hardware resources to containers and, if required, set up limits and constraints. Running applications in isolation requires containers to use only the resources they are intended to, which ensures them to be good multi-tenant citizens on a host. Ex: Limiting the disk usage of a specific container.

3. Union File systems

Union file systems, or UnionFS, are file systems that operate by creating layers, making docker very fast and lightweight environment. UFS is used by Docker engine and acts as building block for containers. Docker Engine utilizes various union file system variants of them which include: AUFS, btrfs, vfs, and DeviceMapper.

4. Container Format

Container format is a wrapper created by Docker Engine which combines all these components. Current default container format is libcontainer.

3 RELATED WORK

This chapter aims to present the major ideas in state of previous performance measurements in the field of NoSQL Databases, Bare Metal and Virtualization Technologies.

Datastax White paper on Why NoSQL [14], discusses the present day need for handling big data with the enormous growth in data. The paper discusses six reasons for the world to move towards NoSQL database like Cassandra from traditional RDBMS databases. The paper highlights the usage of Cassandra that enables to achieve Continuous Availability, Location Independence, Modern transactional capabilities, with having a better architecture and a Flexible Data Model.

Datastax White Paper on Introduction to Cassandra [15] provides a brief overview of the NoSQL data base, its underlying architecture and discusses about the distribution and replication strategies, read and write operations, cloud support, performance and data consistency. The paper highlights the linear scalability feature of cassandra that keeps it to the top among other databases and the easy to use data model that enables developers to use and develop applications.

Avinash Lakshman and Prashanth Malik [16] in their present their model of a NoSQL database Cassandra, which was developed to handle the facebook inbox search problem which involved high write throughputs. The paper presents an overview of the client API, system design and the various distributed algorithms that enable the database to run. The authors have discussed about performance improvement strategies and give a use case of cassandra in an application.

Vishal Dilipbhai Jogi and Ashay Sinha in their work [17] focused on the evaluating the performance of tradional RDBMS and No-SQL databases MySQL, Cassandra and HBase for Heavy Write Operations by a web based REST application. The authors measured that Cassandra scaled up scaled up enormously with fast write operations about ten times while HBase that had almost twice the speed as traditional databases.

Vladimir Jurenka in his paper [18], presents an overview of virtualization technologies and discusses about the container virtualization using docker. The author presents an overview of docker, its architecture, docker images and orchestration tools. The author highlights the docker API and presents a comparative study with other container technologies and finally ends with a real time implementation of file transfer between docker containers running on different hosts and developing a new docker update command that simplifies detection of out-dated images.

Wes Felter, Alexander Ferreira, Ram Rajamony, Juan Rubio in their paper [5] outline the usage of traditional virtual machine and container hypervisors, in cloud computing applications. The authors make a comparative study of native, container and virtual machine environments using hardware and software across a cross-section of benchmarks and workloads relevant to the cloud. The authors identify the performance impact of using virtualized environments and bring into notice issues that affect their performance. Finally, they come up with docker showing sometimes negligible or outperforming performance when compared to Virtual Machines for various test scenarios.

Preeth E N*, Fr.Jaison, Biju and Yedu in their paper [19], evaluated the usage of container technology with bare-metal and hypervisor technologies based on their hardware resources utilization. The others use benchmarking tools like Bonnie+ and benchmarking code psutil to evaluate the performance of file system and CPU, Memory Utilizations. Their research also focused on CPU count, CPU times, Disk Partition, Network I/O counter in

Docker and Host OS. Their research found out that the promising nature of docker in its near native performance.

Roberto Morabito, Jimmy and Miika, in their paper [20] investigated on the performance traditional hypervisor to lightweight virtualization solutions using various benchmarking tools to better understand the platforms in terms of processing, storage, memory and network. The authors results showed the level of overhead introduced by containers to be almost negligible for linux and docker containers which enables dense deployment of services.

4 METHODOLOGY

METHOD

This section describes the research methodology and the experimental test-bed followed in our thesis work:

4.1 Ways to study a system

There are few different ways to study a system and understand its operations, relationship with resources, and measure and analyze its performance.

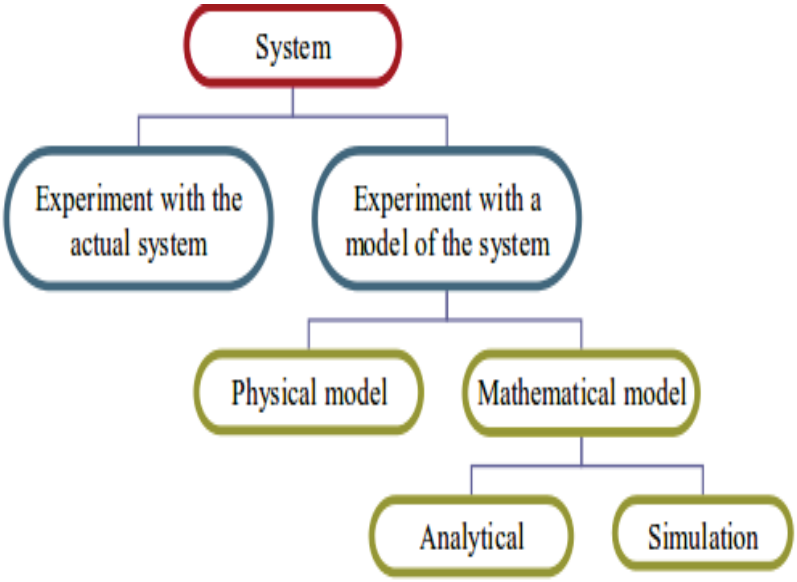

Figure 4:1Classification in Performance Analysis of a System

Experiment with the actual system and Experiment with the model of a system are two ways to do it. Experiment on a physical model of a system is considered for this thesis since our goal is to evaluate the performance of the Cassandra database in a physical server and comparing that to Cassandra implemented in docker containers, a physical model will show the performance overhead without having the need to have production servers. Moreover, a physical model of a system allows us to make alterations to the data-model and schema allowing us to make a more detailed analysis of the performance overhead in two different scenarios.

Furthermore, we consider the physical model over the mathematical model since a mathematical model is an abstract version of the system and shows the logical relationships between components. As our primary goal is to evaluate the performance of the database in various load cases in different virtualized environments, a physical model of a system is considered to be best suited for our analysis.

4.2 Methodology for analyzing a system

This sections details the Tools, Experimental Test beds and the methodology used for analyzing the considered system in our thesis work.

4.2.1 Tools

This section describes the various tools that were used to calculate the parameters required for this thesis work. List of tools along with their usage are detailed below:

1 SAR

- IOSTAT
- Cassandra Stress Tool

4.2.1.1 SAR Tool

The sysstat package provides the sar and iostat system performance utilities. SAR - Collect, report, or save system activity information. Sar is used to collect the CPU usage of the system. Sar takes a snapshot of the system at periodic intervals, gathering performance data such as CPU utilization, memory usage, interrupt rate, context switches, and process creation. When using sar CPU utilization is shown in different fields. **%user** gives the CPU utilization that occurred while executing at the application level. This field includes the time spent running virtual processors. **%usr** gives the CPU utilization that occurred while executing at the application level as well but it does not include time spent running virtual processors. **%nice** gives the utilization while executing at the user level with nice priority. **%idle** gives the percentage of the time the CPU or CPU's were idle and the system did not have an outstanding dick I/O request. %idle gives the summation of all the others and hence is the absolute CPU utilization [21].

4.2.1.2 Iostat Tool

Iostat reports CPU statistics and input/output statistics for devices and partitions. The Iostat tool [22] is used for monitoring system input/output device loading by observing the time the devices are active in relation to their average transfer rates. The Iostat reports can be used to change the system configuration to better balance the input/output load between physical disks.

Iostat tool generates two kinds of reports, the CPU utilization report and the Device utilization report. We are primarily concerned with the disk utilization report of the iostat tool. It provides statistics per physical device or on partition basis. It provides the following lines.

Tps- Indicate the number of transfers per second that were issued to the device. A transfer is an I/O request to the device. Multiple logical requests can be combined into a single I/O request to the device. A transfer is of indeterminate size.

kB_read/s- Indicate the amount of data read from the drive expressed in kilobytes per second. Data displayed are valid only with kernels 2.4 and newer.

kB_wrtn/s- Indicate the amount of data written to the drive expressed in kilobytes per second. Data displayed are valid only with kernels 2.4 and newer.

kB_read- The total number of kilobytes read. Data displayed are valid only with kernels 2.4 and newer.

kB_wrtn

The total number of kilobytes written. Data displayed are valid only with

kernels 2.4 and newer.

4.2.1.3 CASSANDRA-STRESS TOOL

The Cassandra-stress tool is a Java-based stress utility for basic benchmarking and load testing a Cassandra cluster. We are using the stress tool to create a key space called keyspace1 which has the table standard1 and in each of the servers and we observe the CPU and Disk utilization of the respective servers using the sar and iostat tools. Creating the best data model requires significant load testing and multiple iterations. The Cassandra-stress tool helps us in this endeavor by populating our cluster and supporting stress testing of arbitrary CQL tables and arbitrary queries on tables [23].

4.3 Experimental Test-Beds

There are two experimental test-beds in our thesis because we aim to investigate the performance of the database in a native bare-metal server and Cassandra running in docker containers. One test-bed is running the Cassandra database in native bare-metal servers and the second case is running the container in Docker containers. Both the test-beds are in the same servers and hence use the same physical resources.

4.3.1 Test-bed 1: Cassandra on Native Bare-metal Server

This is our first test-bed which is running Cassandra data-base in a native bare-metal server. The physical testbed topology is shown below:

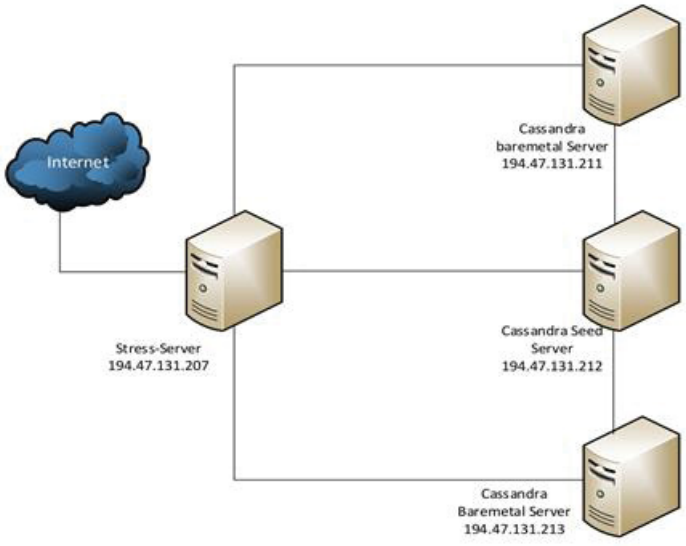

Figure 4:2 Cassandra in native bare-metal server

4.3.1.1 Test-bed details

The Test bed details are as follows:

Table 4:1 Bare-Metal Test Bed

Operating system	Ubuntu 14.04 LTS (GNU/Linux 3.19.0-49-generic x86_64)
RAM	23 GB
Hard-disk	279.4 GB
Processor	12 cores, 2 threads per core ---> 24 theoretical cores
Cassandra	3.0.8
Cassandra-stress tool	2.1

4.3.1.2 Performance evaluation of Cassandra in native Bare-metal servers

In our experimental test-bed, we have four hosts (one source and three destination hosts) with Cassandra 3.0.8 installed in each of them. Cassandra package comes with a command-line stress tool (Cassandra-stress tool) to generate load on the cluster of servers, cqlsh utility, a python-based command line client for executing Cassandra Query Language (CQL) commands and nodetool utility for managing a cluster. These tools are used to stress the servers from the client and manage the data in the servers.

In order to do a performance analysis of Cassandra database we consider three modes of operation of the database - read, write and mixed load operation. Each of the servers have the same configuration of hardware, software and network used. We ensure that the RAM and Hard-disk of each of the servers are equivalent after each iteration of the experiment to ensure the integrity of the results.

Cassandra database has access to full hardware resources of the servers as there is no other competing application for the resources. The Cassandra-stress tool creates a keyspace called keyspace1 and within that, tables named standard1 or counter1 in each of the nodes. These are automatically created the first time you run the stress test and are reused on subsequent runs unless you drop the keyspace using CQL. A write operation inserts data into the database and is done prior to the load testing of the database. We use standard tools sar and iostat to measure the load on each server.

To form the cluster in the three nodes, we install the database in each of the servers. To form the cluster, we set the listen address, rpc address and broadcast address in the Cassandra yaml file to the ip address of the node. The IP address of the one of the nodes is set as the seeds ip address of the cassandra cluster. Setting the seeds in the cassandra.yaml file allows the nodes to communicate with each other and form the cluster.

Sar takes a snapshot of the system at periodic intervals, gathering performance data such as CPU utilization, memory usage, interrupt rate, context switches, and process creation. %idle value of the sar tool gives the total percentage of the time the CPU's were idle which in turn gives us the percentage of CPU's total usage. iostat reports CPU statistics and input/output statistics for devices and partitions. The iostat tool is used for monitoring system input/output device loading by observing the time the devices are active in relation to their average transfer rates. The amount of disk resources utilized is collected with iostat. It

monitors the system by observing the time devices are active in relation to their average transfer rates. We are primarily interested in the **kB_wrtn/s** value of the iostat tool which indicates the amount of data written per second to the disk.

Using the stress tool, we generate write data on the cluster. This is to have a preloaded data in all the nodes. On this data set we have three cases -mixed operation, read operation and write operation for a duration of 20 minutes and record the values of CPU and Disk utilized. Average values of CPU utilized and Disk utilized for an interval of 30 seconds is taken for the entire duration of 20 minutes for the servers in the cluster. Latency is measured in the stress server and is total time taken by the write or read request from the stress server to the time an acknowledgement is received to the server.

4.3.2 Test-bed 2: Cassandra in Docker

This is our second test-bed which is running Cassandra data-base in a docker container. The physical testbed topology is shown below:

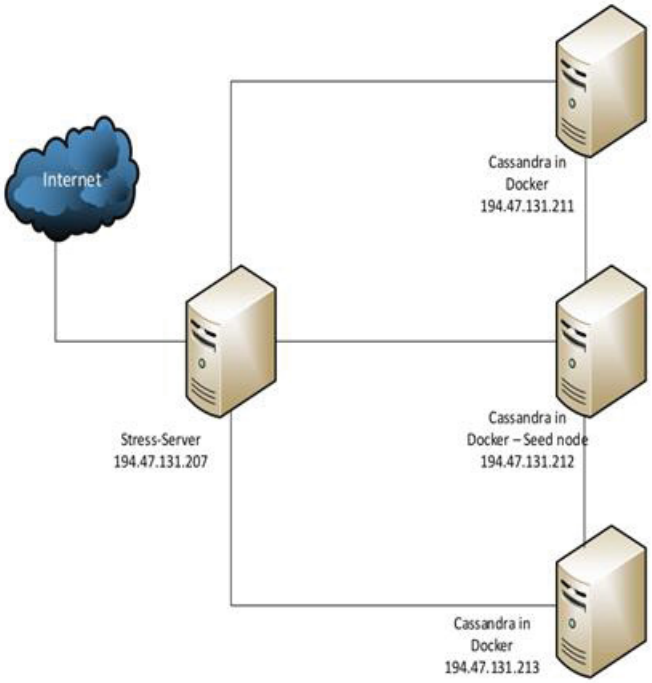

Figure 4:3 Cassandra in Docker

4.3.2.1 Test-bed details

The configuration is as follows

Table 4:2 Docker Test Bed Details

Operating system	Ubuntu 14.04 LTS (GNU/Linux 3.19.0-49-generic x86_64)
RAM	23 GB
Hard-disk	279.4 GB
Processor	12 cores, 2 threads per core ---> 24 theoretical cores
Cassandra	3.0.8
docker	1.11.2
Cassandra-stress tool	2.1

4.3.2.2 Cassandra in Docker

We use the docker image from the Cassandra docker hub. The image is built from a docker file and is available at the docker hub page for Cassandra. The image comes with a default script to run when the Cassandra container is started. This default script allows us to set the environment variables in the cassandra.yaml file directly.

To run cassandra in the seed node we use the command:

$docker run –name some-cassandra -d -e CASSANDRA_BROADCAST_ADDRESS=194.47.131.212 -p 7199:7199 -p 9042:9042 -p 7000:7000 cassandra:3.0.8

This starts the container with the name some-cassandra. -d option runs the container in the background, -e option is to set the environment variable in the cassandra yaml file, -p option forwards the traffic from the respective port of the host to the container and finally cassandra version that we use is 3.0.8 which is same as the one in bare-metal server.

In the other two nodes we start cassandra with

$ docker run –name some-cassandra -d -e CASSANDRA_BROADCAST_ADDRESS=194.47.131.211 -e CASSANDRA_SEEDS=194.47.131.212 -p 7199:7199 -p 9042:9042 -p 7000:7000 cassandra:3.0.8

$ docker run –name some-cassandra -d -e CASSANDRA_BROADCAST_ADDRESS=194.47.131.212 -e CASSANDRA_SEEDS=194.47.131.212 -p 7199:7199 -p 9042:9042 -p 7000:7000 cassandra:3.0.8

CASSANDRA_SEEDS value points to the ip address of the seed node which forms the cluster and communicates with the other nodes.

4.3.2.3 Performance evaluation of Cassandra in docker

Once the Cassandra cluster is formed in the docker container on the host OS. We consider the same three modes of operation of the database -read, write and mixed load operation. Cassandra in docker has access to all the hardware resources of the servers again as there is no competing application for the resources.

We use sar and the iostat tools on the host operating system while running the database in the container. This gives us the overhead of the CPU and disk resources used by deploying the database in a container. We consider the same parameters of CPU utilization from %idle value of sar and Disk utilization from kB_wrtn/s of iostat. We follow the same procedure as we did for bare-metal evaluation.

4.4 Statistical and measurement based system analysis

Statistical analysis involves collecting and inspecting every data sample in a set of items from which samples, which are a representative selection from a total population can be drawn.

4.4.1 Confidence Intervals

Taking a random sample from a population to compute mean is to find the approximation of mean of a sample. In order to see how well the mean estimates the underlying population can be taken care of by building Confidence Intervals. Confidence Interval gives the range of values within which there is a specified probability that the value of a parameter lies in it.

95% confidence Interval means if we used the same sample to compute to compute the estimate on various occasions and interval estimates are made on each occasion, then we can expect the true population to fall within the interval 95% of the time.

Confidence Intervals are calculated based on standard deviation (deviation from mean value), population size and confidence level.

Standard Deviation is calculated using the below formula.

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

Where μ = mean

x_i= Value of 'i'th element in the sample

N = Total number of elements in the sample

σ = Standard Deviation

Then calculate the z value which is based on z table for 95% confidence interval the value is 1.96.

Population Mean is given by the following formula.

$$Population\ Mean\ (\mu) = Sample\ Mean \pm sample\ error$$

Where, Sample error = $(z * \sigma)/\sqrt{n}$

The confidence interval is the difference between upper and lower bounds of population mean.

4.5 Experiment Scenarios

This section details the different types of loads generated onto the cluster that is used in this thesis work using the test beds described in section 4.3. The experiment scenarios are formulated based on the aims of this thesis work and aid in proving solution to the proposed research questions.

The load scenarios generated by the load generator node in order to populate the cluster using the Cassandra-stress tool are described below:

- Mixed Load
- Read Load
- Write Load

We conducted the experiment by using running 100%load which is the maximum load that can be generated onto the cluster and 66% load which is 66% of the maximum load The maximum throughput generated while running the Cassandra stress tool has been noted during the experiment and set as the maximum load which is in the case of 450threads and we have also considered the 66% of maximum load i.e. 150 threads with the above load types for our experiment. Before running the above case scenarios, the cluster is populated with approximately 11GB (Giga Bytes) of blob data that enables the experiment to be run in on a stable system. All the scenarios with the load types are shown in the figure.

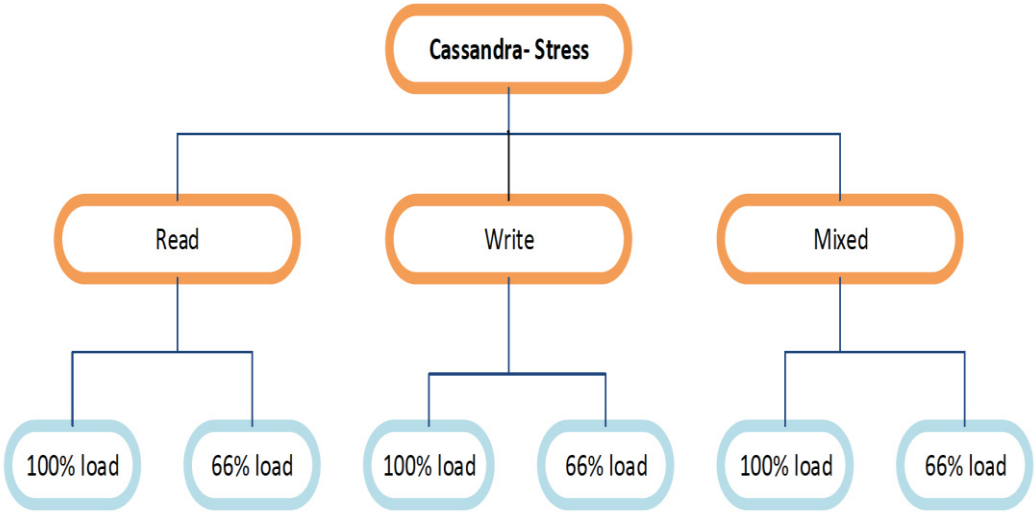

Figure 4:4 Scenarios

4.5.1 Mixed Load

To analyze the operation of a database while running both read and write operations during one operation, a mixed load case is used to populate the cluster. A mixed load operation of 3 reads and 1 write generated for a duration of 20minutes onto the cluster is used in this thesis work. The commands used for creating 100% and 66% mixed load operations respectively are described below:

$./cassandra-stress mixed ratio\(write=1,read=3\) duration=20m cl=ONE -pop dist=UNIFORM\(1..50000000\) -rate threads\=450 -node 194.47.131.212;

$./cassandra-stress mixed ratio\(write=1,read=3\) duration=20m cl=ONE -pop dist=UNIFORM\(1..50000000\) -rate threads\=150 -node 194.47.131.212;

A short description of the terms in the above command are noted below:

- duration: It specifies the time in minutes to run the load.
- cl: It indicates Cassandra consistency level to be used
- rate threads: It is used to specify the thread count
- node: To specify the address of the node to which data is to be populated

A 100% load is generated by setting the thread count parameter to 450 while a 66% load is generated by setting the parameter to 150.

4.5.2 Write Load

To analyze the operation of a database in the case of write operations, a write load type is used to populate the cluster. A write load operation for a duration of 20minutes onto the cluster is used in this thesis work. The commands used for creating 100% and 66% write load operations respectively are described below:

$./cassandra-stress write duration=20m cl=ONE -pop dist=UNIFORM\(1..50000000\) -rate threads\=450 -node 194.47.131.212;

$./cassandra-stress write duration=20m cl=ONE -pop dist=UNIFORM\(1..50000000\) -rate threads\=150 -node 194.47.131.212;

4.5.3 Read Load

To analyze the operation of a database in the case of read operations, a read load type is used to populate the cluster. A read load operation for a duration of 20minutes onto the cluster is used in this thesis work. The commands used for creating 100% and 66% read load operations respectively are described below:

$./cassandra-stress read duration=20m cl=ONE -pop dist=UNIFORM\(1..50000000\) -rate threads\=450 -node 194.47.131.212;

$./cassandra-stress read duration=20m cl=ONE -pop dist=UNIFORM\(1..50000000\) -rate threads\=150 -node 194.47.131.212;

4.6 Metrics

The following metrics are to be known to understand the monitoring the system performance while running the Cassandra database.

- **Read requests:** The data stored in the database is read through read requests by the client.
- Write requests: The data is written to the database using write requests by the client.

- **Write request latency:** The total time taken by the nodes to accept a write request and send a response back to the client in the case of write operations is the write request latency.

- **Read request latency:** The total time taken by the nodes to accept a read request and send a response back to the client (i.e., load generator) is the Read request latency.

- **Operations per second:** The total number of write and/or read requests generated by the client per second is the operations per second.

- **CPU utilization:** It is the number of processor cycles that are utilized by the process. It is monitored at the servers running the database while running for mixed, read and write workloads

5 RESULTS AND ANALYSIS

This Chapter presents the individual, common results, and a detailed analysis of the obtained results of the experiment. This includes the performance metric values of latency, cpu and disk utilizations on individual node while stressing the cluster with a load generator for different workloads and their measurements in graphical manner along with their analysis.

5.1 Individual Results

This section presents the individual experimental results and their analysis performed which involves with the CPU Utilization and Read Latency Parameters.

5.1.1 Experimental Description for CPU Utilization

Cpu Utilization is the number of processor cycles that are utilized by the process. In our experiment we only run one process inside the servers to evaluate the performance of the database. In our experiment we use the source traffic model where in order to evaluate any parameter, we initially create a cassandra cluster of three nodes then from another load which is the load generator, populate the cluster with some BLOB (Binary Large Object) data into the cluster using cassandra-stress tool so we can perform the experiments on a preloaded cluster that ensures there is no random behavior while experimenting.

Once a preloaded cluster is setup, we populate the cluster with the desired load scenario as described in section 4.6 for a duration of 20 minutes and with the desired load either 100% load or 66% of the maximum load.

Sar tool is executed the same time when the desired load scenario is generated by the load generator which collects the values of different parameters of cpu utilization while running Cassandra in both Bare Metal and Docker testbeds. The value for Idle Cpu usage is obtained by the following sar command executed in the server at the time of loading the cluster:

```
sar -u 30 40 | awk '{print $8 "\t" $9 }' > cpu
```

In this thesis work, the values of Idle CPU percentage which is the amount of time the CPU is idle are considered in order to calculate the CPU Utilization. The actual Cpu utilization of the server is found by the following formula:

$$\%\ \boldsymbol{Cpu\ Utilization} = \mathbf{100} - \%\ \boldsymbol{Cpu\ Idle}$$

The above sar command sets the values to be taken for every 30seconds and thus outputs a total of 40values as the experiment is run for a duration of 20minutes. Every value in the output means it's an average of 30seconds from the previous output while the first output is an average of first 30 seconds.

Graphical Representation of obtained results and furthermore, 95% Confidence Intervals are plotted to verify if the obtained results from the experimental process can be considered to make out some analysis. A Linearity in the confidence intervals and overlapping means the results obtained are genuine or else they cannot be considered for making out statements. We perform 10 such iterations and the average values obtained for these iterations is taken as the result.

The same experimental procedure is repeated for both the test beds and the values obtained after the analysis are shown graphically in the following sub sections based on the type of load scenario (mixed/read/write) and load type (100% or 66% load).

5.1.1.1 Results from Mixed Load Operations

This section presents the results obtained after running mixed load operation on a loaded cluster with BLOB data.

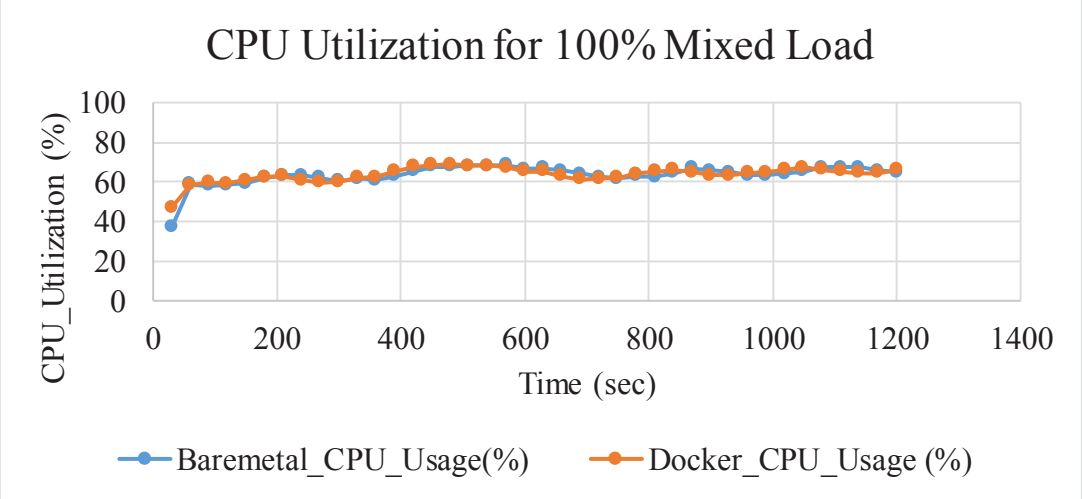

Figure 5:1 CPU Utilization for 100% Mixed Load

The results in the fig show the variation of Cpu Utilization with respect to time duration. Here every point on the graph is an average of cpu utilizations of 30 seconds duration. Further the Cpu Utilization of running cassandra with maximum load using 450 threads in a docker container environment is similar to that of native bare-metal servers.

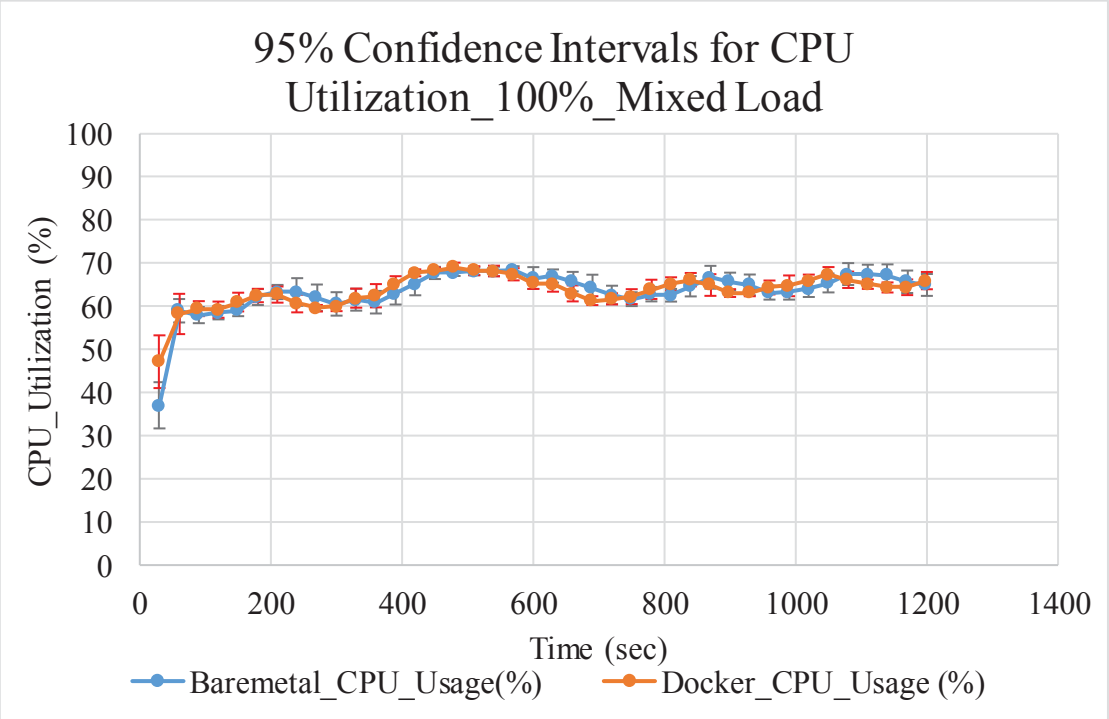

Figure 5:2 95% Confidence Interval for CPU Utilization for 100% Mixed Load

95% Confidence Intervals for Mixed load scenario with 450threads show most of the values to be overlapping that makes it to verify that the experimentation was rightly done. Furthermore, average values for mixed load with 2 and 8 intervals is shown in Fig 5.3 below:

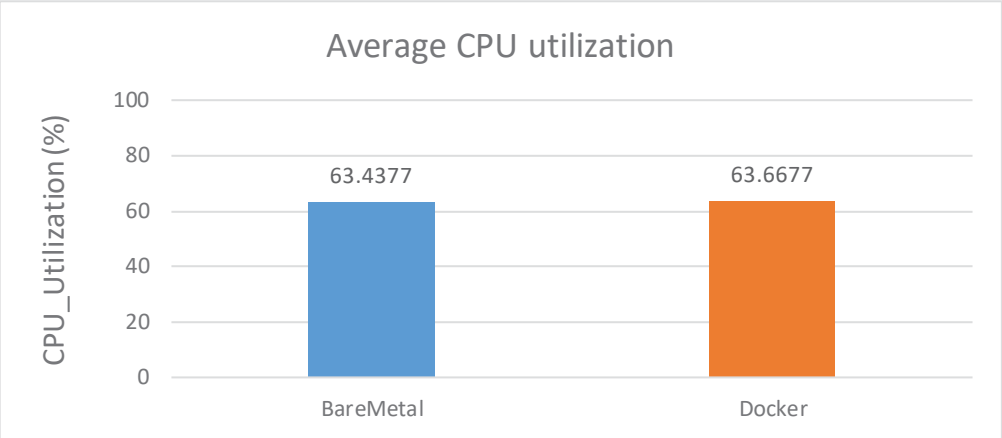

Figure 5:3 Average CPU utilization

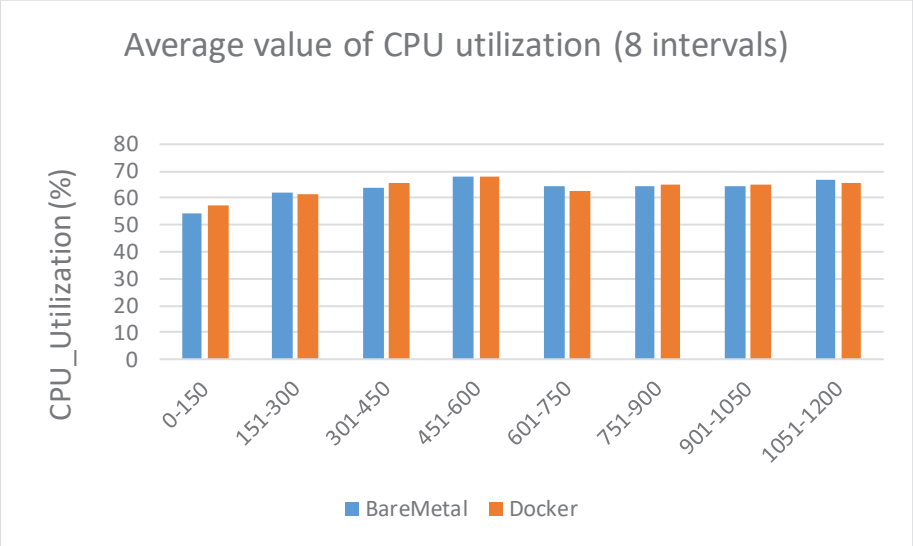

Figure 5:4 Average value of CPU utilization (8 intervals)

5.1.1.2 Write Operation

This section presents the results obtained after running write load operation on a loaded cluster with BLOB data.

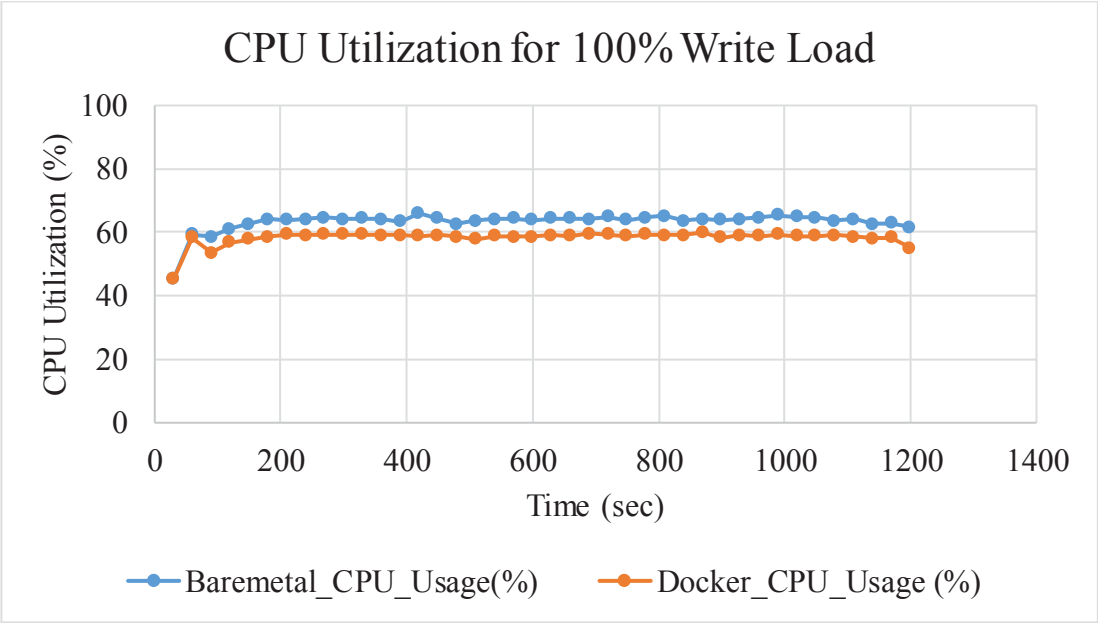

Figure 5:5 CPU Utilization for 100% Write Load

The results in the fig show the variation of Cpu Utilization with respect to time duration. Here every point on the graph is an average of cpu utilizations of 30 seconds duration. Further the Cpu Utilization of running cassandra with maximum load using 450 threads in a docker container environment is similar to that of native bare-metal servers and Bare Metals to be having a slightly more Cpu Utilization.

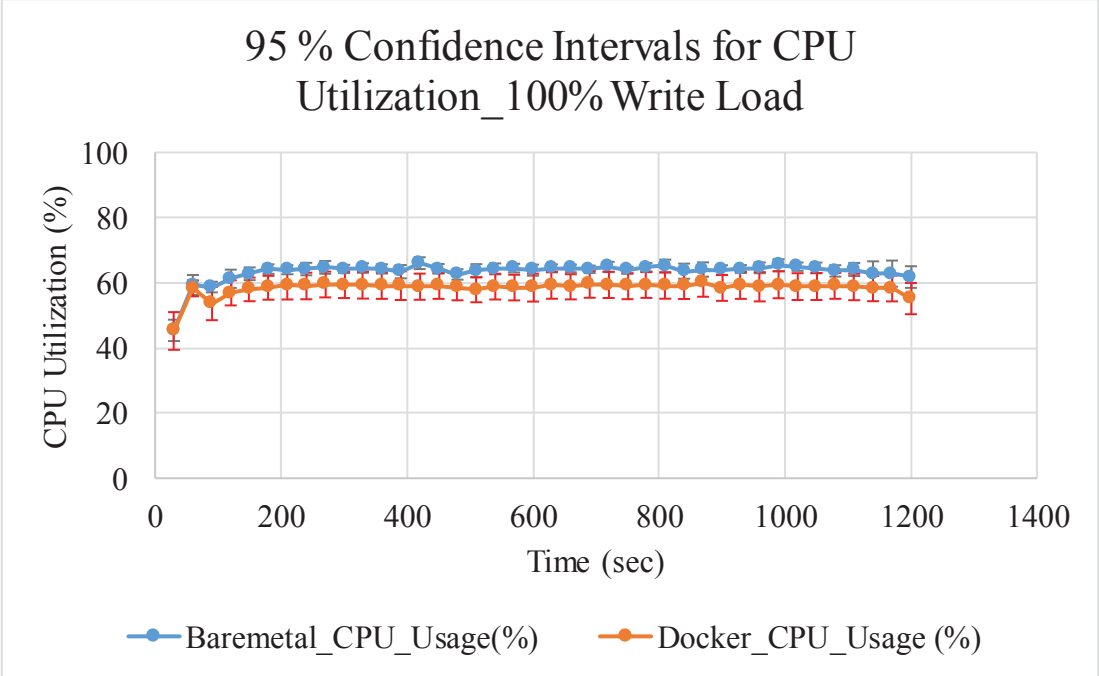

Figure 5:6 95% Confidence Interval for CPU Utilization for 100% Write Load

95% Confidence Intervals for Write load scenario with 450threads show most of the values to be overlapping that makes it to verify that the experimentation was rightly done.

5.1.1.3 Read Load

This section presents the results obtained after running read load operation on a loaded cluster with BLOB data.

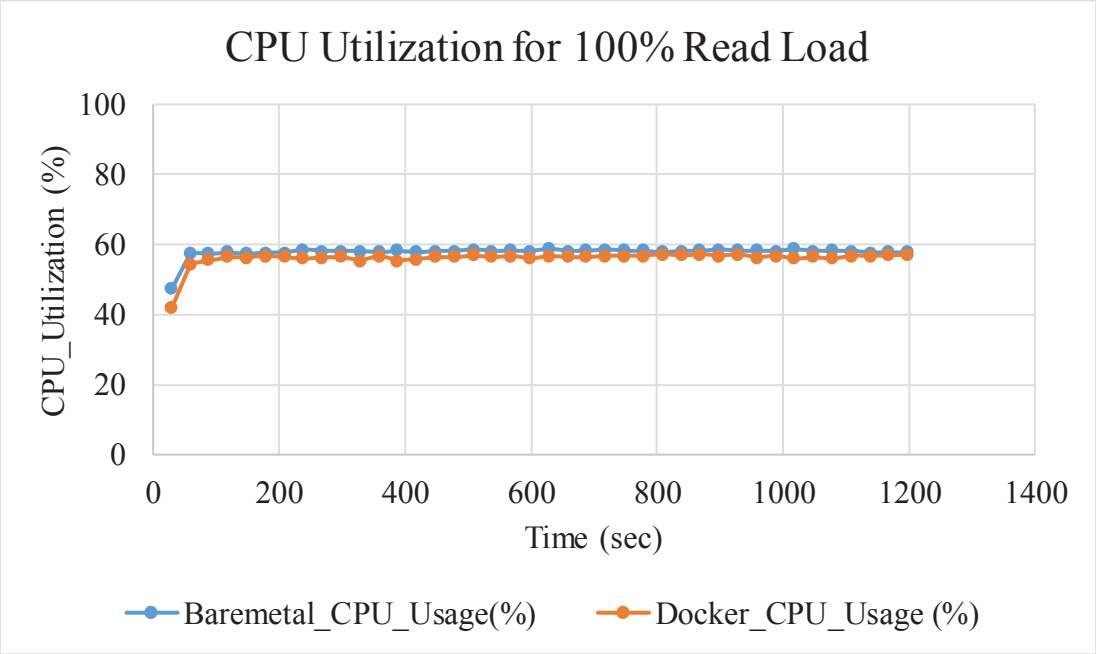

Figure 5:7 CPU Utilization for 100% Read Load

The results in the fig show the variation of Cpu Utilization with respect to time duration. Here every point on the graph is an average of cpu utilizations of 30 seconds duration. Further the Cpu Utilization of running cassandra with maximum load using 450 threads in a docker container environment is similar to that of native bare-metal servers and Bare Metals to be having a slightly more Cpu Utilization.

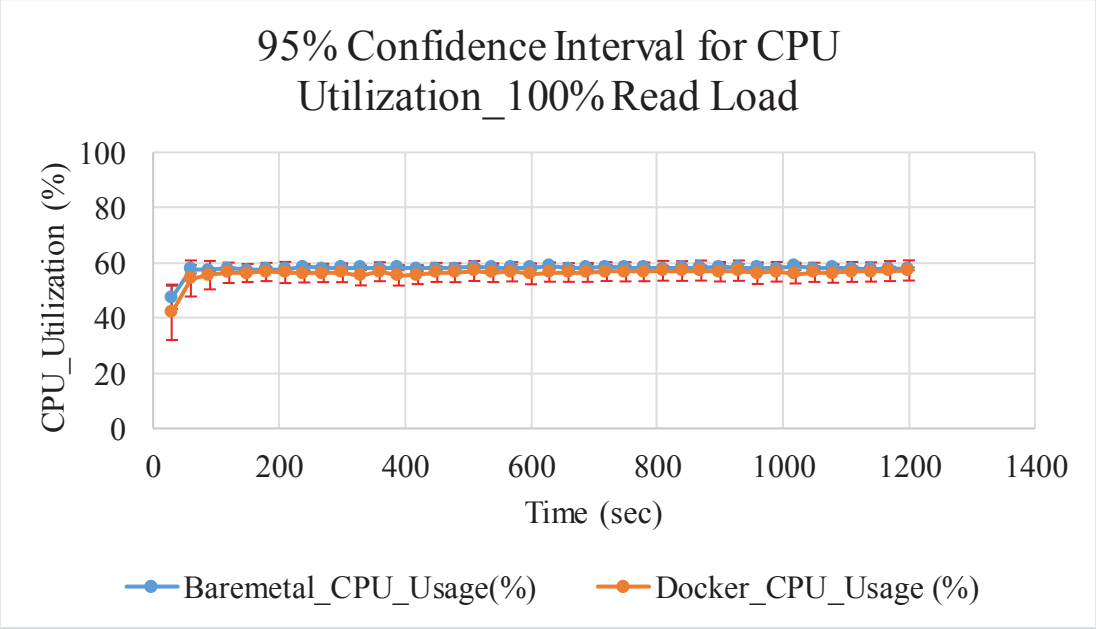

Figure 5:8 95% Confidence Interval for CPU Utilization for 100% Read Load

95% Confidence Intervals for Read load scenario with 450threads show most of the values to be overlapping that makes it to verify that the experimentation was rightly done.

5.1.2 Experimental Description for Latency

The latency of the operations is the time taken for the operations (either read or write) to be executed and generate a response from the cluster running Cassandra. The latency values are noted at the load generator server.

The latency of operations is noted while the load is generated on the cluster for read, write and mixed operations. The results for different load cases and operations are shown below.

As the load increases the latency of the operations increases because there are more number of responses for the requests per second.

Scenario 1 Mixed load operations: We generate a mixed load operation of 3 reads and 1 write on a cluster having 11GB of data for a duration of 20 minutes. The average latency of the operations is taken for one iteration. We perform 10 such iterations and the average value of the latency for these iterations is taken as the result.

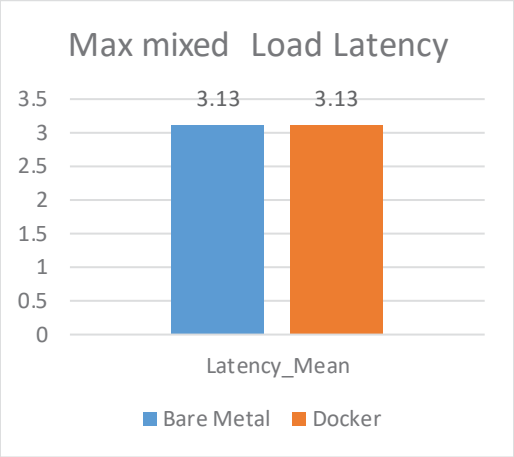

Figure 5:9 Max mixed load latency

Write operations: We generate a write load case on the cluster having 11GB of data. The write load is run for a duration of 20 minutes. We take 10 such iterations and perform mean value analysis.

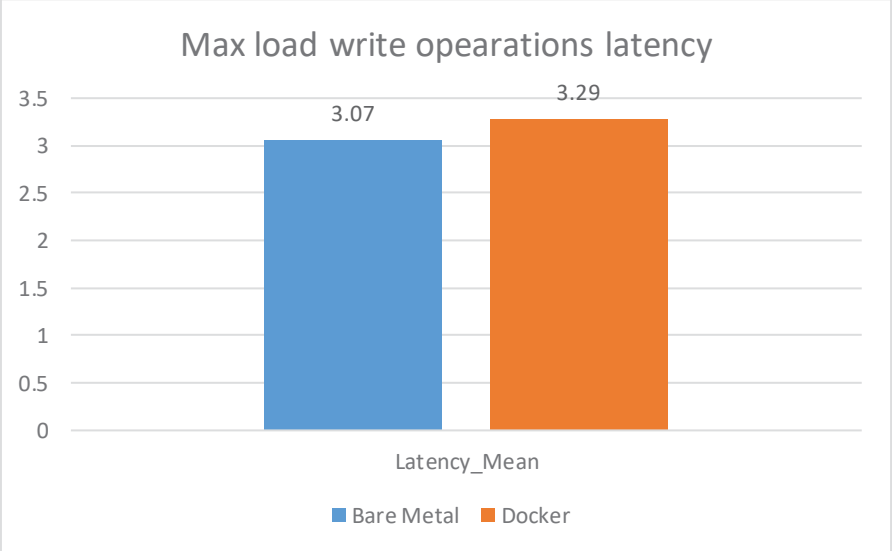

Figure 5:10 Max load write operations latency

5.2 Common Results

These are the results of Disk Utilization and Latency experiment performed by Rajeev. The methodology is the same as in the case of Cpu Utilization.

5.2.1 Experimental Description for Disk Utilization

Disk utilization of a server for a process is the amount of disk space occupied by the process during the run time of the process. In our experiment to evaluate the disk utilization of the servers we generate the data-set from load generator into the cluster having three Cassandra nodes. Each node is an individual server running Cassandra. The Cassandra cluster is created by configuring the Cassandra.yaml file in each of the servers. The disk utilization is measured on the servers running the Cassandra database

The disk usage of each of the servers while the load is generated on the cluster for read, write and mixed operations is considered for this thesis work. The results for different load cases and operations are shown below.

The amount of disk space used depends on the data stored in the physical memory before it is flushed to the disk. If for some reason the memTables are not flushed to the disk. Then there will be no disk utilization. The disk utilized is a collection of memTables and compaction together.

Disk throughput depends on the rate of requests sent to the cluster. As the number of requests increases the load increases as observed when comparing the maximum load case against the 66% of the maximum load case. This is because more data is flushed to the disk and compactions occur quickly.

Cassandra 3.0.8 is installed in four servers (one as load generator and three servers forming the Cassandra cluster). The load generator runs for the different load cases of read, write and mixed load and The write Kbps of iostat tool gives the disk utilization. While the load is generated at the client, we run the iostat tool on the servers running the database. The values of disk utilization are collected using the bash command:

iostat -d 30 40 | grep sda | awk '{print $4}' > disk

The same experiment is repeated for docker by installing Cassandra inside a docker container, forming a cluster by configuring the cassandra.yaml, running the load generator for read, write and mixed load cases and collecting the values of disk throughput in the servers.

5.2.1.1 Mixed Load

Mixed load of three reads and one write are taken for a duration of 20 minutes to observe the server's disk utilization. Disk throughput for 100% load: The total op/s sec pushed to the cluster is around 150000. We use the default Cassandra configurations for memory and other parameters. The stress is generated over a cluster that has 11 GB of data. On this cluster we perform the mixed load operation of 3 reads and 1 write. The maximum Cassandra in docker disk utilization is 21795.61 Kbps and the maximum Cassandra bare-metal disk utilization is 21051.96 Kbps.

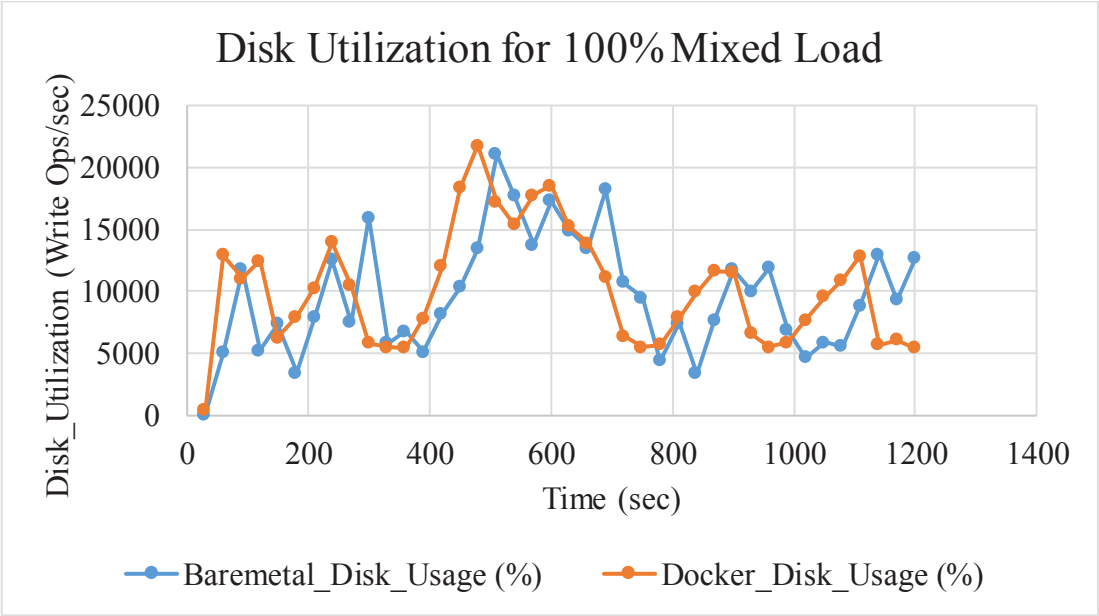

Figure 5:11 Disk Utilization for 100% Mixed load

- In the above graph, the disk utilization is plotted for a duration of 20 minutes. Each value specifies the average value for 30 seconds.
- Values indicate that there are more compactions happening on the disk of the server because memTables are flushed
- The disk utilization is around 5000 - 22000 op/s in both the cases of Docker and bare-metal servers.

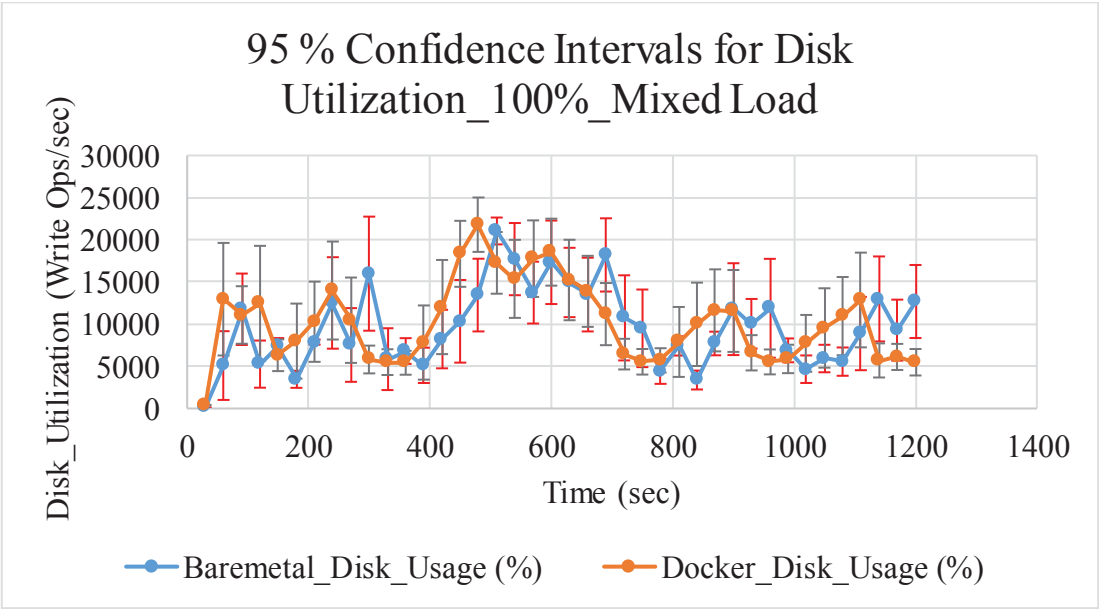

Figure 5:12 95 % Confidence Intervals for Disk Utilization_100%_Mixed Load

- This is the same graph as above with the confidence intervals. There is high standard deviation in the results because compaction can occur at any time and this is true both in the case if bare-metal servers and Docker. Although this shows ambiguity in the results because of compactions the overall average indicating similar performance for both Docker and bare-metal servers.

- 95% Confidence Intervals for Read load scenario with 450threads show most of the values to be overlapping that makes it to verify that the experimentation was rightly done.
- Average value of Disk Utilization for 100% Mixed Load for 2 and 8 time intervals are shown below:

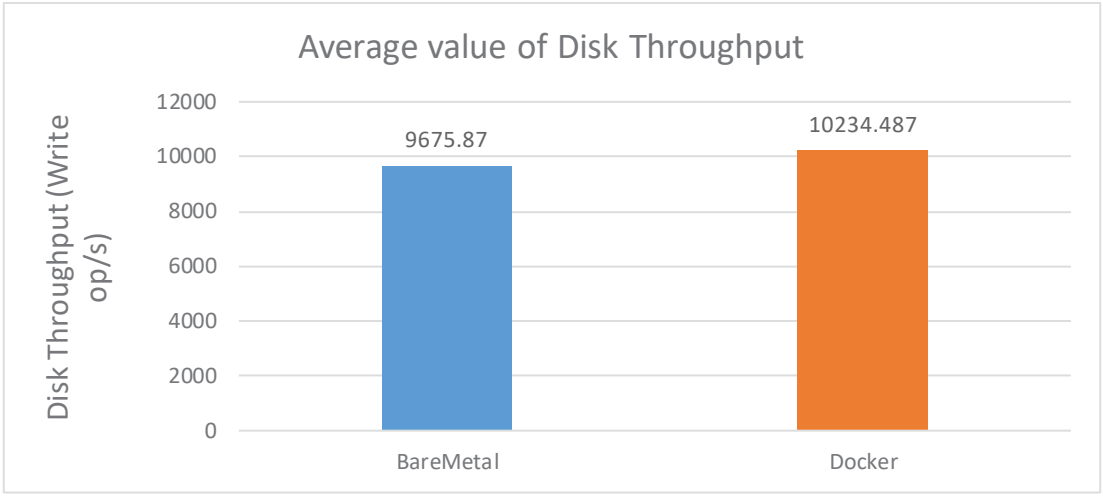

Figure 5:13 Average Disk Utilization

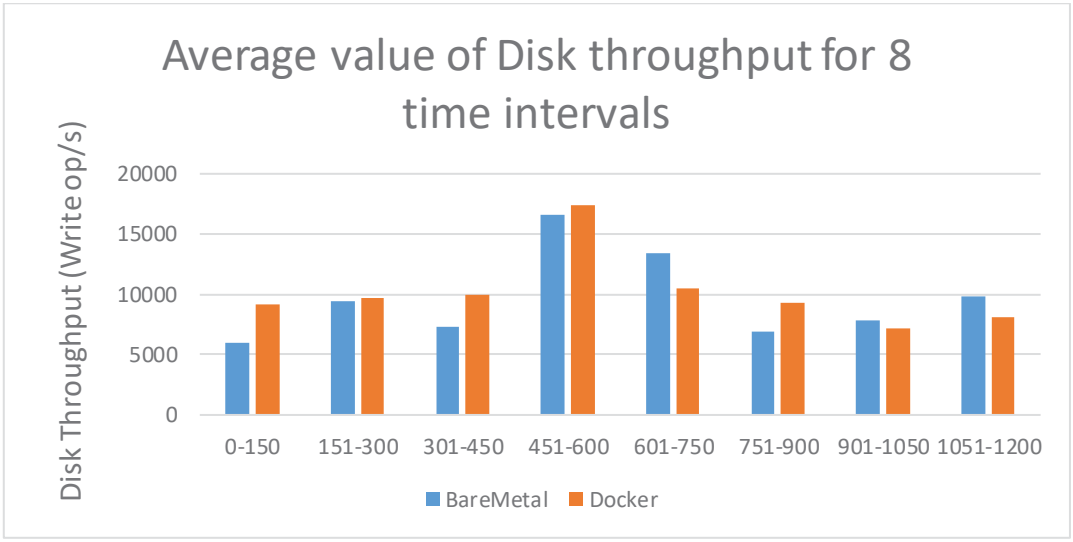

Figure 5:14 Average disk utilization (8 intervals)

- The average value for 10 iterations for the case of bare-metal and docker is taken.
- Also, the entire duration is divided into multiple intervals of 2 halves, 4 halves, and 8 halves. Cassandra in Docker shows better performance in the first 4 halves and in the second half bare-metal shows better disk throughput

5.2.2 Experiment description a for Latency

The latency of the operations is the time taken for the operations (either read or write) to be executed and generate a response from the cluster running Cassandra. The latency values are noted at the load generator server.

The latency of operations is noted while the load is generated on the cluster for read, write and mixed operations. The results for different load cases and operations are shown below.

As the load increases the latency of the operations increases because there are more number of responses for the requests per second.

5.2.2.1 Mixed load operations

We generate a mixed load operation of 3 reads and 1 write on a cluster having 11GB of data for a duration of 20 minutes. The average latency of the operations is taken for one iteration. We perform 10 such iterations and the average value of the latency for these iterations is taken as the result.

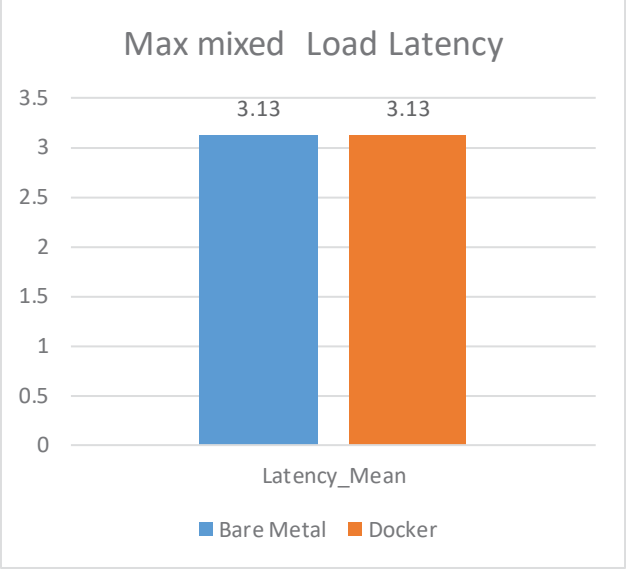

Figure 5:15 Max Mixed Load Latency

5.2.3 Discussion

CPU Utilization: In the case of mixed load operations, the cpu utilization in bare-metal server running Cassandra is 63.43 and docker running Cassandra is 63.66. The CPU utilization is slightly higher in the case of write workloads and read workloads but the confidence interval values largely overlap showing that the values are almost the same.

Latency: The average latency in the case of mixed load operations is the same in both docker and bare metal for maximum load case. In the 66% case we see that the average latency of 10 iterations for bare metal is 1.4 and 1.45 for docker. The average latency for 10 iterations in the case of max load write operations is 3.07 ms for bare metal and 3.29 in the case 66% load docker 1.38 ms for bare metal, 1.49ms for docker. this means that docker adds about 6-7% network overhead in the case of write operations. The latency in the case of reads is 3.07 in the case of bare-metal and 3.29 in the case of docker. In the case of 66% load the latency is 1.38 in bare metal and 1.49 in the case of docker.

Disk utilization: In the case of mixed load operation, the average disk utilization of the servers while running the Cassandra database in Docker was higher. The mean average value in the case of bare-metal is 9675.87 and in the case of docker is 10234.487. Though this is an anomaly, there is a possibility that running the experiment for a longer time would show us that the disk utilization is the equivalent for both Cassandra in Docker and in bare metal as the average values for 2 halves, 4 halves and 8 halves show that the bare-metal performance is better later on.

From the mean value analysis, we speculate that the performance will be the same for both Cassandra in Docker and bare-metal. There is high standard deviation in the results

because compaction can occur at any time and this is true both in the case if bare-metal servers and Docker.

In the case write work load disk utilization is greater in the case of bare metal and the time series analysis shows that it is consistently better in both the cases of maximum load and 66% of the maximum load. Although this is the case the standard deviation is quite high for both the scenarios. Hence it is not possible to predict the exact overhead of running the database in containers. Though the overhead might be there, it seems that the overall overhead would be small.

6 CONCLUSION AND FUTURE WORK

The purpose of this thesis was to monitor the disk performance and latency of the Cassandra database in the case of bare-metal servers and compare its performance with the case of containers. Standard tools for monitoring the performance of servers – sar for cpu utilization and iostat for disk utilization were used. The overhead in the case of containers would decide the feasibility of using containers for running the database.

Firstly, from the results, server disk utilization in the case of bare-metal servers and docker have shown equivalent values in the case of mixed load, Cassandra in bare-metal slightly outperform. In the case of latency of operations there is overhead in the case of Cassandra in docker for write and mixed load cases. Different cases of operations of the database and load cases to analyze the performance of the database are shown and time series values for each case is graphed. This shows a variability in the performance of the database with time and the average values are compared for docker and bare-metal cases. The limitation of this methodology is that we do not how compactions affect the disk utilization from our results by themselves. One way for a deeper analysis of the database would be to check the nodetool compaction history values to see the number of compactions happening and see if it corresponds to the disk utilization results to show that the disk utilization at a particular instance is due to compaction and explain the anomalies in instances where Cassandra in docker has outperformed Cassandra in bare-metal. By combining the system analysis results with the results from observing the cassandra based metrics like compaction history, we can create a more wholesome methodology for performance analysis of the database.

6.1 Answers to Research Questions

RQ1. What is the methodology for analyzing the performance of databases in virtual machines?
We implement the physical model of the Cassandra cluster based on realistic and commonly used scenarios or database analysis for our experiment. We generate different load cases on the cluster for Bare-Metal and Docker and see the values of CPU utilization, Disk throughput and latency using standard tools like sar and iostat. Statistical analysis (Mean value analysis, higher moment analysis and confidence intervals) are done on measurements on specific interfaces in order to show the reliability of the results.

RQ2. What is the CPU utilization of the server while running the database in bare-metal case and in containers?
In the case of mixed load operations, the cpu utilization in bare-metal server running Cassandra is 63.43 and docker running Cassandra is 63.66. The CPU utilization is slightly higher in the case of write workloads and read workloads but the confidence interval values largely overlap showing that the values are almost the same which makes it to be a suitable alternative to other virtualized platforms.

RQ3. How does this performance vary with different load scenarios?
The results of docker container with regards to Cpu Utilization show similar and very close behavior to bare-metal. 95% Confidence Intervals for Mixed load scenario with 450threads show most of the values to be overlapping that makes it to verify that the experimentation was rightly done.

6.2 Future Work

This thesis aims focused on the investigating the trade-off in performance while loading a Cassandra cluster in bare-metal and containerized environments. A detailed study of

the effect of loading the cluster in an individual node in terms of Latency, CPU and Disk throughput have been analyzed with the default cassandra configurations.

Future Work in this area can be done by making configurations changes in compaction strategies, memory, read and write rates in the cassandra yaml file. In docker containers instead of using a prebuild image from the docker hub, a customized cassandra image based on our requirements can be built and tested with various configurations. This would pave a future for a faster and quick deployment of applications on a wider scale.

7 REFERENCES

[1] R. Lawrence, "Integration and Virtualization of Relational SQL and NoSQL Systems Including MySQL and MongoDB," in *Proceedings of the 2014 International Conference on Computational Science and Computational Intelligence - Volume 01*, Washington, DC, USA, 2014, pp. 285–290.

[2] E. Casalicchio, L. Lundberg, and S. Shirinbad, "An Energy-Aware Adaptation Model for Big Data Platforms," in *2016 IEEE International Conference on Autonomic Computing (ICAC)*, 2016, pp. 349–350.

[3] "Docker: Containers for the Masses." [Online]. Available: http://patg.net/containers,virtualization,docker/2014/06/05/docker-intro/. [Accessed: 09-Oct-2016].

[4] "NoSQL Comparison Benchmarks," *DataStax*. [Online]. Available: http://www.datastax.com/nosql-databases/benchmarks-cassandra-vs-mongodb-vs-hbase. [Accessed: 14-Sep-2016].

[5] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 171–172.

[6] Intellipaat, "Brief Architecture of Cassandra - Cassandra Tutorial | Intellipaat.com," *Intellipaat*. [Online]. Available: https://intellipaat.com/tutorial/cassandra-tutorial/brief-architecture-of-cassandra/. [Accessed: 09-Oct-2016].

[7] "The write path to compaction." [Online]. Available: https://docs.datastax.com/en/cassandra/2.0/cassandra/dml/dml_write_path_c.html. [Accessed: 14-Sep-2016].

[8] "About reads." [Online]. Available: https://docs.datastax.com/en/cassandra/2.0/cassandra/dml/dml_about_reads_c.html. [Accessed: 09-Oct-2016].

[9] "About hinted handoff writes." [Online]. Available: https://docs.datastax.com/en/cassandra/2.0/cassandra/dml/dml_about_hh_c.html?hl=hinted,handoff. [Accessed: 09-Oct-2016].

[10] "Docker Overview," *Docker*, 30-May-2016. [Online]. Available: https://docs.docker.com/engine/understanding-docker/. [Accessed: 14-Sep-2016].

[11] "Machine Overview." [Online]. Available: http://54.71.194.30:4110/machine/overview/. [Accessed: 09-Oct-2016].

[12] "Understand the architecture." [Online]. Available: http://54.71.194.30:4019/engine/introduction/understanding-docker/. [Accessed: 09-Oct-2016].

[13] "Docker." [Online]. Available: https://delftswa.github.io/chapters/docker/#layered-structure. [Accessed: 09-Oct-2016].

[14] "Why NoSQL?," *DataStax*. [Online]. Available: http://www.datastax.com/resources/whitepapers/why-nosql. [Accessed: 14-Sep-2016].

[15] "Introduction to Apache Cassandra," *DataStax*. [Online]. Available: http://www.datastax.com/resources/whitepapers/intro-to-cassandra. [Accessed: 14-Sep-2016].

[16] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *SIGOPS Oper Syst Rev*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[17] V. D. Jogi and A. Sinha, "Performance evaluation of MySQL, Cassandra and HBase for heavy write operation," in *2016 3rd International Conference on Recent Advances in Information Technology (RAIT)*, 2016, pp. 586–590.

[18] V. Jurenka, "Virtualizace pomocí platformy Docker," Master's thesis, Masaryk University, Faculty of Informatics, 2015.

[19] P. E. N, F. J. P. Mulerickal, B. Paul, and Y. Sastri, “Evaluation of Docker containers based on hardware utilization,” in *2015 International Conference on Control Communication Computing India (ICCC)*, 2015, pp. 697–700.
[20] R. Morabito, J. Kjällman, and M. Komu, “Hypervisors vs. Lightweight Virtualization: A Performance Comparison,” in *2015 IEEE International Conference on Cloud Engineering (IC2E)*, 2015, pp. 386–393.
[21] “sar(1) - Linux man page.” [Online]. Available: http://linux.die.net/man/1/sar. [Accessed: 14-Sep-2016].
[22] “iostat(1) - Linux man page.” [Online]. Available: http://linux.die.net/man/1/iostat. [Accessed: 14-Sep-2016].
[23] “The cassandra-stress tool.” [Online]. Available: https://docs.datastax.com/en/cassandra/2.1/cassandra/tools/toolsCStress_t.html. [Accessed: 14-Sep-2016].

8 APPENDIX

This section provides the CPU results for 66% Load generation scenarios, Cassandra Cluster Formation steps and sample screenshots of Nodetool status that indicates cluster status, Latency value, command execution to obtain CPU utilization using sar tool.

8.1 CPU Results for 66% Load Scenarios

This section provides the CPU Utilization measurements in an individual node (194.47.131.212) while running 66% Mixed, Read and Write Loads on a cluster loaded with BLOB data.

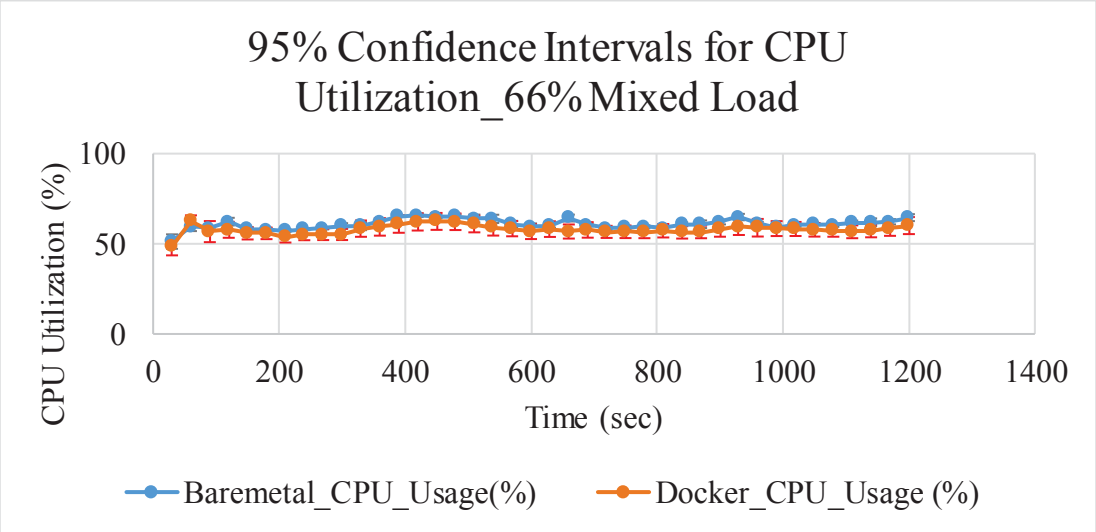

Figure 8:1 95% Confidence Interval for CPU Utilization for 66% Mixed Load

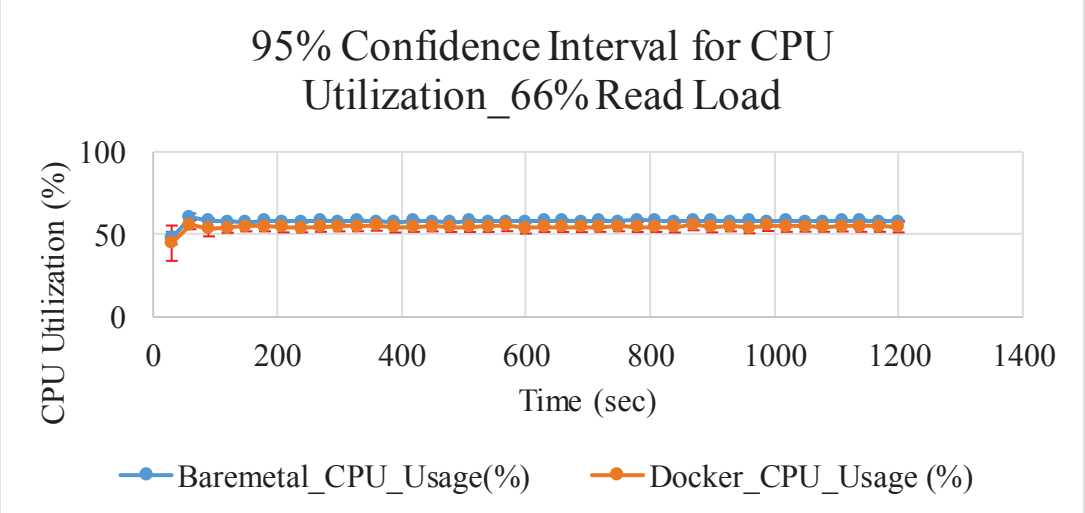

Figure 8:2 95% Confidence Interval for CPU Utilization for 66% Read Load

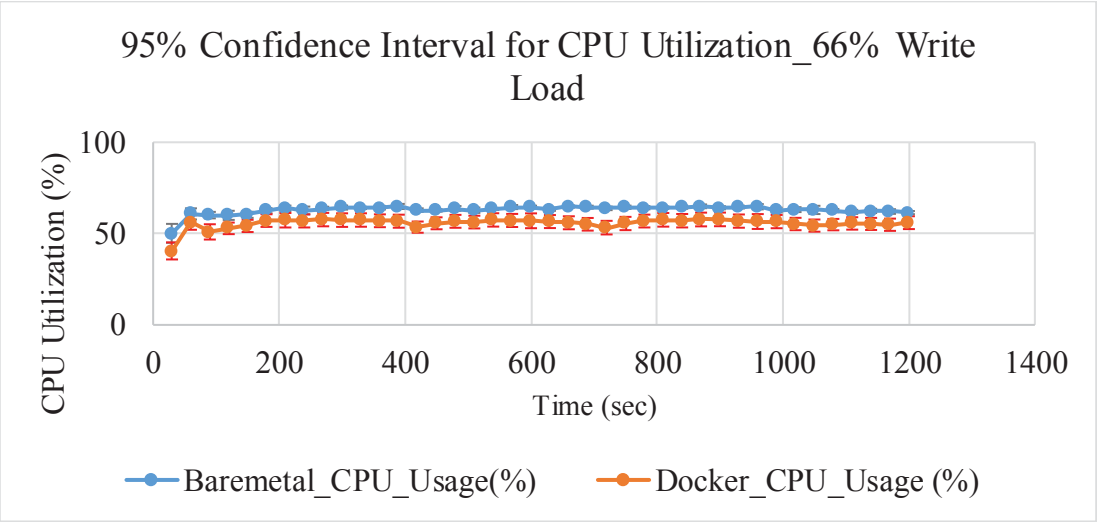

Figure 8:3 95% Confidence Interval for CPU Utilization for 66% Write Load

8.2 Cassandra Cluster

This section gives a brief introduction about Cassandra cluster formation in both bare metal and docker test beds.

8.2.1 Cassandra Cluster in Bare Metal

- Install Link to Cassandra Image in Bare Metal using linux command.

wget http://apache.mirrors.spacedump.net/cassandra/3.0.8/apache-cassandra-3.0.8-bin.tar.gz

- Extract tar file using the command
  tar zxvf apache-cassandra-3.0.8-bin.tar.gz
- Changes to be made in Cassandra yaml file locate in
  apache/conf/cassandra.yaml are:
  seeds:194.47.131.212
  rpc_address:node ip
  listen_address:node ip
- Starting Cassandra
  bin/cassandra -R
- Check cluster status
  bin/nodetool status
- Check Nodetool status values on individual node when:
  - Cluster is created
  - After writing BLOB data
  - After Load Generation

1. Operation on Load Generator Node
   a. Create data by going to install_location/tools/bin then
   ./cassandra-stress write n=50000000 -node 194.47.131.212
   b. Mixed LOAD
   cassandra-stress mixed ratio\(write=1,read=3\) duration=20m cl=ONE -pop dist=UNIFORM\(1..50000000\) -rate threads\=450 -node 194.47.131.212;
   c. WRITE LOAD
   ./cassandra-stress write duration=20m cl=ONE -pop dist=UNIFORM\(1..50000000\) -rate threads\=450 -node 194.47.131.212;
   d. READ LOAD
   ./cassandra-stress read duration=20m cl=ONE -pop dist=UNIFORM\(1..50000000\) -rate threads\=450 -node 194.47.131.212;

2. Checking CPU and Disk on Individual Node by
   sar -u 30 40 | awk '{print $8 "\t" $9 }' > cpu_baremetal
   iostat -d 30 40 | grep sda | awk '{print $4}' > disk_baremetal

   sar -u 30 40 | awk '{print $8 "\t" $9 }' > cpu_docker
   iostat -d 30 40 | grep sda | awk '{print $4}' > disk_docker

3. Check Nodetool status on node11

4. To clear data in cluster by dropping keyspace in Node 194.47.131.212
   bin/cqlsh 194.47.131.212
   desc keyspaces
   drop KEYSPACE keyspace1

5. Repeat the experiments.
   If you want to clear the data then do by issuing the following commands but if you just clear data/data, logs, commit log, you need to kill the cassandra using pid and start cassandra again with cassandra -R
   During this process if for example the data in the seed node is cleared,cassandra stopped. If you try to start the cluster, there is sometimes an error, nodes 10,12 form a cluster when you check nodetool status in wither 10 or 12,
   When you do nodetool status from 11, you will get just node 11 as up or nothing.
   The reason for this is, while you kill and start node 11, what we do in the cassandra yaml file is we assign the seed address as 194.47.131.212.
   When you try to do the same from 10 and 12 by killing the cassandra instance and starting, as the seed node is 11, they try to communicate with the seed and are up.
   But in case of Node 11, when it tries to communicate with seed which is 11, as the node is not up, it fails to join the cluster.
   Solution to the problem can be done with the following steps:
   1) Change the seed address to one of the nodes that formed the cluster other than 11
   2) Stop and start cassandra instances on all machines
   3) now a cluster with seeds address being 10 or 12 is up
   4) As in our experiment, we started to take the results in node 11, we changed the cluster seed to 11 again. Restarted the cassandra instances on each node, which helped in forming back the cluster.

   As an alternative, dropping the keyspace would be fine instead of clearing the whole data and performing the experiment as the load on inidividual nodes indicated using the nodetool status command, shows the same result even after clearning the data in data/data/*, data/commitlog, /log/* files.

6. Stopping Cassandra by clearing data first then stopping:
   Clear data by in the install_location
   rm -rf data/data/* data/commitlog/ logs/*

   Stop Cassandra
   ps aux | grep cass
   youll see the pid
   kill pid
   Start cassandra by using
   cassandra -R

   Nodetool status to check if the cluster was up.

7. If nodetool after dropping keyspaces isn't working, it could be the space is filled on the disk and data was not cleared properly which can be resolved by:
   Finding files with top 10 data usage in human readable format using
   du -sh * | sort -hr | head -n10

8. To replace dead nodes use the following command which has been used for 194.47.131.212 node.
   bin/cassandra -R -Dcassandra.replace_address=194.47.131.212

8.2.2 Cassandra Cluster in Docker

Cassandra Cluster in Docker is created by port forwarding and using the predefined Cassandra image in the public registry of Docker called Docker Hub and executing the following commands on individual nodes shown as follows:

- Node 194.47.131.212
  docker run --name some-cassandra-11 -d -e CASSANDRA_BROADCAST_ADDRESS=194.47.131.212 -p 7199:7199 -p 9042:9042 -p 7000:7000 cassandra:3.0.8
- Node 194.47.131.211
  docker run --name some-cassandra-11 -d -e CASSANDRA_BROADCAST_ADDRESS=194.47.131.211 -p 7199:7199 -p 9042:9042 -p 7000:7000 -e CASSANDRA_SEEDS=194.47.131.212 cassandra:3.0.8
- Node 194.47.131.213
  docker run --name some-cassandra-11 -d -e CASSANDRA_BROADCAST_ADDRESS=194.47.131.213 -p 7199:7199 -p 9042:9042 -p 7000:7000 -e CASSANDRA_SEEDS=194.47.131.212 cassandra:3.0.8

8.3 Screenshots from Experiments

A sample screenshot for one of the experimental operations on docker while running 100% Mixed Load on to the cluster.

```
total,      50000000,  130845,  130845,  130845,     1.5,     1.0,     3.4,     7.1,    44.7,
    47.9,  401.7,  0.00427,      0,       0,       0,       0,       0,       0


Results:
op rate                   : 124470 [WRITE:124470]
partition rate            : 124470 [WRITE:124470]
row rate                  : 124470 [WRITE:124470]
latency mean              : 1.6 [WRITE:1.6]
latency median            : 0.9 [WRITE:0.9]
latency 95th percentile   : 3.0 [WRITE:3.0]
latency 99th percentile   : 5.6 [WRITE:5.6]
latency 99.9th percentile : 59.3 [WRITE:59.3]
latency max               : 2562.5 [WRITE:2562.5]
Total partitions          : 50000000 [WRITE:50000000]
Total errors              : 0 [WRITE:0]
total gc count            : 0
total gc mb               : 0
total gc time (s)         : 0
avg gc time(ms)           : NaN
stdev gc time(ms)         : 0
Total operation time      : 00:06:41
END
root@diptprisrv06:/home/sogandgroup/apache-cassandra-3.0.8/tools/bin#
```

Figure 8:4 Latency for 100% Mixed Load on Docker

```
======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address         Load       Tokens       Owns (effective)  Host ID                               Rack
UN  194.47.131.213  233.57 KB  256          74.8%             49300369-a84d-4a42-8165-842a6007815b  rack1
UN  194.47.131.212  210.21 KB  256          63.0%             f3857e7f-26cd-42ea-95f2-8903bfd13dc2  rack1
UN  194.47.131.211  223.57 KB  256          62.1%             a9be9be8-0d09-433f-bc27-f8c80316f6a8  rack1

root@502322434c29:/# nodetool status
Datacenter: datacenter1
======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address         Load       Tokens       Owns (effective)  Host ID                               Rack
UN  194.47.131.213  4.07 GB    256          35.3%             49300369-a84d-4a42-8165-842a6007815b  rack1
UN  194.47.131.212  3.65 GB    256          31.9%             f3857e7f-26cd-42ea-95f2-8903bfd13dc2  rack1
UN  194.47.131.211  3.79 GB    256          32.8%             a9be9be8-0d09-433f-bc27-f8c80316f6a8  rack1

root@502322434c29:/# nodetool status
Datacenter: datacenter1
======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address         Load       Tokens       Owns (effective)  Host ID                               Rack
UN  194.47.131.213  6.7 GB     256          35.3%             49300369-a84d-4a42-8165-842a6007815b  rack1
UN  194.47.131.212  6.25 GB    256          31.9%             f3857e7f-26cd-42ea-95f2-8903bfd13dc2  rack1
UN  194.47.131.211  6.16 GB    256          32.8%             a9be9be8-0d09-433f-bc27-f8c80316f6a8  rack1

root@502322434c29:/#
```

Figure 8:5 Nodetool for 100% Mixed Load in Docker

```
Note: Non-system keyspaces don't have the same replication settings, effective own
ership information is meaningless
root@09aba499551f:/# exit
exit
root@diptprisrv11:/home/sogandgroup/ericsson/docker# ls
100%  66%
root@diptprisrv11:/home/sogandgroup/ericsson/docker# cd 100%/
root@diptprisrv11:/home/sogandgroup/ericsson/docker/100%# ls
root@diptprisrv11:/home/sogandgroup/ericsson/docker/100%# sar -u 30 40 | awk '{pri
nt $8 "\t" $9  }' > cpu_docker.txt
root@diptprisrv11:/home/sogandgroup/ericsson/docker/100%# ls
cpu_docker.txt
root@diptprisrv11:/home/sogandgroup/ericsson/docker/100%# ls
cpu_docker.txt
root@diptprisrv11:/home/sogandgroup/ericsson/docker/100%# ls
root@diptprisrv11:/home/sogandgroup/ericsson/docker/100%# sar -u 30 40 | awk '{pri
nt $8 "\t" $9  }' > cpu_docker.txt
root@diptprisrv11:/home/sogandgroup/ericsson/docker/100%# ls
cpu_docker.txt  disk_docker.txt
root@diptprisrv11:/home/sogandgroup/ericsson/docker/100%# sar -u 30 40 | awk '{pri
nt $8 "\t" $9  }' > cpu_read.txt
root@diptprisrv11:/home/sogandgroup/ericsson/docker/100%#
```

Figure 8:6 Sar Command Execution to calculate CPU Utilization