Comparative Performance Analysis of Column Family Databases: Cassandra and HBase

by

VINAY SHETH
202111032

A Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of

MASTER OF TECHNOLOGY
in
INFORMATION AND COMMUNICATION TECHNOLOGY
to

DHIRUBHAI AMBANI INSTITUTE OF INFORMATION AND COMMUNICATION TECHNOLOGY

July, 2023

Declaration

I hereby declare that

i) the thesis comprises of my original work towards the degree of Master of Technology in Information and Communication Technology at Dhirubhai Ambani Institute of Information and Communication Technology and has not been submitted elsewhere for a degree,

ii) due acknowledgment has been made in the text to all the reference material used.

VinaySheth

Vinay Sheth

Certificate

This is to certify that the thesis work entitled "Comparative Performance Analysis of Column Family Databases: Cassandra and HBase" has been carried out by VINAY SHETH for the degree of Master of Technology in Information and Communication Technology at *Dhirubhai Ambani Institute of Information and Communication Technology* under my/our supervision.

Prof. PM Jat

Thesis Supervisor

Acknowledgments

I am grateful to my supervisor, Prof. PM Jat for his valuable guidance and encouragement throughout the M. Tech. thesis process. His extensive knowledge and expertise have been instrumental in shaping my research and improving the quality of my thesis.

I would also like to express my sincere gratitude to Prof. Minal Bhise, Prof. Kalyan Sasidhar, Prof. Supantha Pandit and Prof. Gopinath Panda for their valuable suggestions and comments during both the stage presentations.

I am thankful to the IT Help Desk staff of Dhirubhai Ambani Institute of Information and Communication Technology, for providing technical support and prompt assistance whenever required.

I would like to extend my heartfelt gratitude to my family and friends, who have provided me with unwavering encouragement and support throughout my thesis journey. Their constant love and encouragement have kept me motivated and inspired.

Lastly, I would like to thank Dhirubhai Ambani Institute of Information and Communication Technology, whose support has made it possible for me to pursue my M.Tech. program and carry out this thesis.

Contents

Abstract

Up until now, relational databases have been unquestionably the most prevalent type of databases used to handle data. The advent of cloud computing and big data has underlined the need for databases that are capable of managing and analyzing big data. By allowing storage and retrieval of structured as well as unstructured data, NoSQL databases circumvent the limitations of relational databases. Because of their support for schema flexibility, rapid data access and potential to scale up quickly, they have emerged as the favored choice for big data processing.

These systems have several properties/parameters which can be tuned to achieve specific performance goals based on business needs. Having well-defined performance objectives assist us in articulating the acceptable trade-offs for our application. This motivates us to evaluate the performance of one such frequently used NoSQL system: Cassandra. Apache Cassandra is an open-source, decentralized, distributed, fault-tolerant, highly available, elastically scalable, tunably consistent, row-oriented database. In order to accomplish the performance evaluation, we use the Yahoo! Cloud Serving Benchmark (YCSB) for benchmarking efforts. Our findings highlight that increasing thread count initially improves throughput and CPU utilization but later decreases it. Higher record count, consistency level, and dataset size lead to decreased throughput and increased latency. Stronger consistency level also increases the CPU utilization. Increasing operation count improves throughput but increases latency as well. These findings provide guidance for optimizing Cassandra's performance by adjusting these parameters.

We also assess Apache HBase, another well-known NoSQL database, using YCSB. The relative performance of these databases under analytical as well as update-heavy workloads is the primary focus of our investigation. Our test results demonstrate that for both workloads, Cassandra outperforms HBase in read operations, whereas HBase excels in write operations. This research quantifies the performance traits of Cassandra and HBase, assisting developers and architects in choosing the best database system for their big data applications.

List of Principal Symbols and Acronyms

ACID	Atomicity, Consistency, Isolation, and Durability
AP	Availability and Partition Tolerance
BASE	Basically Available, Soft State, Eventually Consistent
CAP	Consistency, Availability and Partition Tolerance
CP	Consistency and Partition Tolerance
CPU	Central Processing Unit
CQL	Cassandra Query Language
DBMS	Database Management System
HDFS	Hadoop Distributed File System
IOT	Internet of Things
NO SQL	Not Only SQL
RAM	Random Access Memory
RF	Replication Factor
SQL	Structured Query Language
SSTABLE	Sorted Strings Table
WAL	Write Ahead Log
YCSB	Yahoo! Cloud Serving Benchmark

List of Tables

List of Figures

CHAPTER 1
Introduction

1.1 Background Information

NoSQL (Not Only SQL) databases have emerged as the data platform and a critical industrial solution for coping with data explosion. They are currently widely employed in various market segments, including business-critical systems, social networks, large-scale Internet applications, Internet of Things (IoT), and other industrial applications. The notion of NoSQL databases has been developed to successfully store and offer rapid access to Big Data sets whose volume, velocity, variety, veracity, and variability are challenging to cope with using standard Relational Database Management Systems [11]. Most NoSQL stores forego ACID (atomicity, consistency, isolation, and durability) properties in favour of BASE (basically available, soft state, eventually consistent) features as a cost of distributed data management, schema flexibility, and horizontal scalability [21].

Trade-offs between consistency, availability, and latency are inherent in NoSQL databases [16] [19]. Although the CAP theorem discovered these relationships qualitatively, it is still important to quantify how various system parameters/ properties (workload, consistency, threads, operation count, record count, request distribution, number of nodes, replication factor) impact system latency and throughput. Understanding this trade-off is critical for using NoSQL solutions effectively.

Although there are several NoSQL databases available in the market, majority of the industry trends indicate that Apache Cassandra is one of the most popular systems in use. Apache Cassandra delivers high availability with no single point of failure and a collection of unique characteristics (e.g., tunable consistency, high availability, data distribution using consistent hashing, flexibility to work across geographically distributed data centres, elastic scalability, etc.) that make it one of the most versatile and popular NoSQL systems.

Apache HBase is another popular column family database used in the industry. Having done the performance evaluation of Cassandra motivates us to explore the capabilities of HBase and consider it as a potential alternative for our data storage needs. Both Cassandra and HBase are prominently used because of their ability to handle enormous amount of data. Both offer high-performance capabilities while being highly scalable and fault-tolerant. In spite of these similarities, there are notable differences between Cassandra and HBase architectures which affect the read/write performance of these systems.

1.2 Objective

The objective of this thesis is (i) To conduct a thorough performance evaluation of Cassandra using the Yahoo! Cloud Serving Benchmark (YCSB) as the benchmarking tool, and (ii) To conduct a comprehensive comparative performance analysis of Cassandra and HBase, two popular column family databases, with the aim of assessing their performance characteristics for read-intensive/update-intensive workloads.

The study seeks to provide valuable insights and empirical evidence to aid decision-making in selecting the most appropriate database for specific use cases. The specific objectives of this thesis include:

1. To conduct a series of performance experiments using YCSB core workloads to measure the performance metrics of Cassandra.
2. To analyze and interpret the benchmarking results to identify the effect of system parameters on the performance of Cassandra.
3. To evaluate the trade-offs and performance implications of using Cassandra and HBase in different application contexts.
4. To contribute to the existing body of knowledge in the field of distributed databases by advancing the understanding of the performance characteristics and comparative analysis of Cassandra and HBase.

By achieving these objectives, this thesis aims to enhance the understanding of the performance capabilities of Cassandra and HBase and provide valuable guidance to researchers, practitioners, and organizations in making informed decisions regarding the selection of database systems.

1.3 Thesis Problem Statement

The goal of this thesis is to assess Cassandra's performance under various configurations when benchmarked with YCSB. We also aim to evaluate the relative performance of Cassandra and HBase for read-intensive and write-intensive workloads.

The following are the Research Questions:

1. What are the possible evaluation metric measurements (throughput, latency, CPU utilization) for various system configurations when running the Cassandra cluster, stressed by different workloads?
2. How can we find an ideal system configuration to achieve specific performance goals?
3. Which among Cassandra and HBase is best suited for execution of analytical and update-intensive workloads? Can we quantify the relative read/write performance of these databases?

1.4 Motivation

It is always desirable to experimentally understand and analyze the performance of a system.

Performance tuning of NoSQL database systems is an evolving topic with the advent of "Big Data". Answering research questions 1 and 2 will allow us to comprehend Cassandra's architecture and behaviour, and determine the average number of operations per second that a user may execute under different types of workloads for various configurations. The experimental results will help in proposing the most optimal configuration for running a particular type of workload.

Literature indicates that Cassandra exhibits relatively better read performance while HBase has a relatively better write performance in terms of throughput and latency [10] [12] [22] [23]. Answering research question 3 will help us in validating the same and deciding the selection of a particular column family database for a given use case.

1.5 Contribution

The research presented in this thesis makes the following key contributions to the field of distributed databases, specifically in the context of (i) Performance evaluation of Cassandra, and (ii) Performance comparison of column family databases-Cassandra and HBase:

1. Through rigorous experimentation and benchmarking, we have evaluated the performance of Cassandra across different workload configurations, record counts, operation counts, thread counts, consistency levels, and dataset sizes. By measuring key evaluation metrics such as throughput, read/update latency, and CPU utilization, we provide a detailed analysis of Cassandra's performance characteristics upon varying these parameters. We have confirmed that our findings are consistent with the results obtained in previous studies.

2. We analyze and interpret the benchmarking results to identify the effect of thread count on throughput; thread count, consistency level on CPU utilization and dataset size on latency.

3. We quantify the relative read/write performance of Cassandra and HBase for read-intensive and write-intensive workloads in terms of read latency, update latency and achieved throughput.

4. We analyze the effect of thread count and target throughput on latency and throughput while executing these workloads. By specifying different values of target throughput for a given value of thread count, we identify the threshold achieved throughput, which is helpful in indicating the scenario of a system overload.

1.6 Thesis Outline

The rest of the thesis is organized as follows: Chapter 2 describes Column Family databases and analyzes the data model, architectural implementation and query execution of Cassandra and HBase. Chapter 3 presents the literature review. Chapter 4 discusses the experiments conducted to evaluate the performance of Cassandra using YCSB. Chapter 5 presents the comparative performance analysis of Cassandra and HBase for read-intensive and update-intensive workloads, and the rationalization of the results. Chapter 6 concludes the thesis and suggests possible directions for future research.

CHAPTER 2
Column Family Databases

2.1 Introduction

Column Family databases enable storing data using a column-oriented model. They are also referred to as column store databases, column oriented databases, wide column stores and columnar databases. When compared to traditional relational databases which store data in rows and tables, the data in column families is arranged in a columnar manner, where related data is grouped and stored together according to the columns to which it belongs [21]. Each column family consists of a group of rows or records, where each row may have a distinct set of columns. As a result, it is possible to create flexible schema designs in which various rows of the same column family may include a variety of columns and the corresponding data.

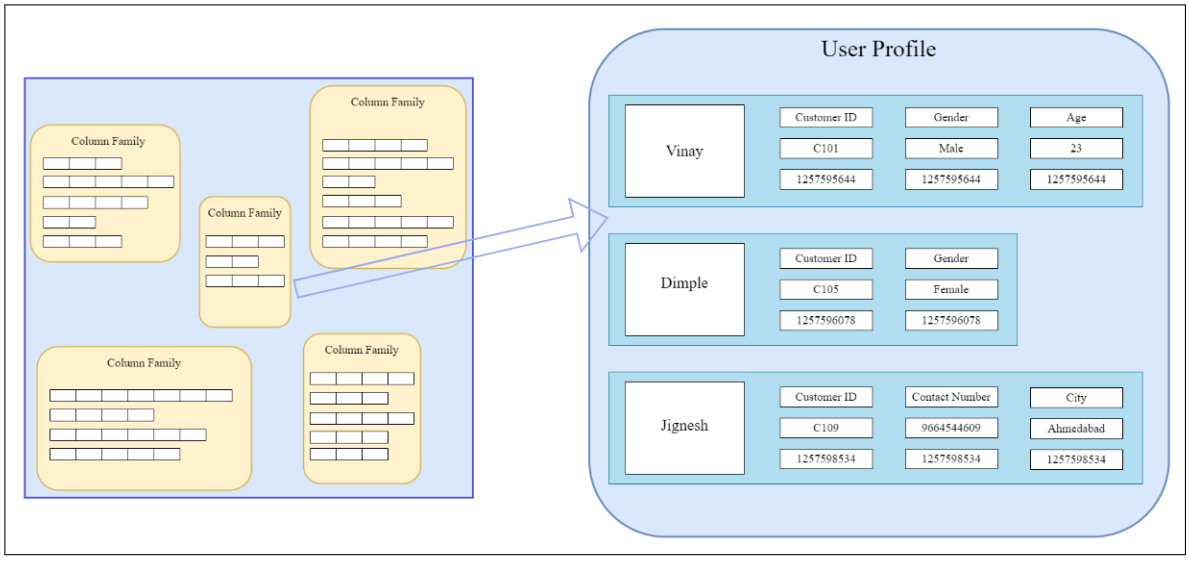


Figure 2.1: Database Schema containing Column Families

Fig. 2.1 shows a database schema containing 5 column families. It also provides a closer look to the column family named 'User Profile'. The figure illustrates the following:

- A column family can consist of multiple rows/records.
- The number of columns in different rows can be different. Additionally, the columns need not be the same in all the rows i.e., they can have different column names, column data types, etc.
- Each column is enclosed within its associated row. In contrast to a relational database, it does not span every row. Every column contains a name-value pair and a timestamp. The example used in the figure uses POSIX time for the timestamp.

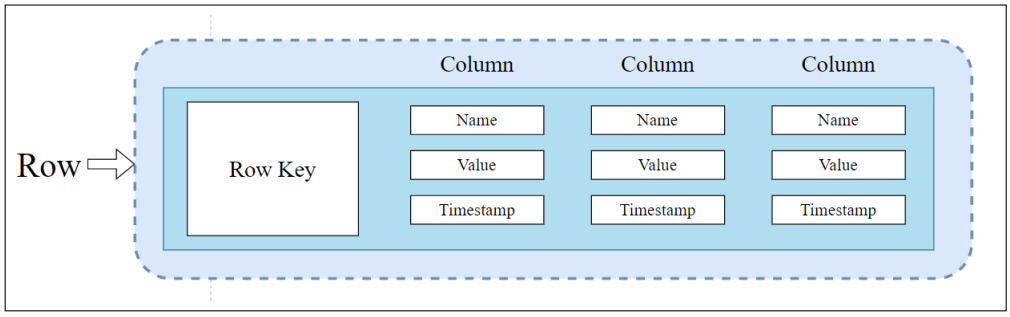


Figure 2.2: Column Family Row Structure

The description of each component of a column family row as illustrated in Fig. 2.2 is as follows:

1. Row Key: Every row has a distinct key, which serves as the row's unique identifier.
2. Column: Each column has a name, a value and a timestamp.
3. Name: It is the name of the column.
4. Value: It is the value of the column for the given row.
5. Timestamp: It specifies the date and time when the data was inserted. This field is used to determine the most recent version of the data.

2.2 Cassandra

Cassandra is an open-source NoSQL database management system created by Facebook. It was built to manage massive amounts of data spread over several commodity servers [17]. Cassandra has various advantages over other NoSQL databases, including elastic scalability, quick linear-scale performance, flexibility in data storage, simplicity of data dissemination, transactional support, fault tolerance and high availability.

2.2.1 Data Model

As opposed to conventional relational databases, Cassandra uses a column-family data model. In this model, data is arranged in tables, but the structure of these tables is flexible, enabling dynamic column addition or deletion without altering the existing structure [19].

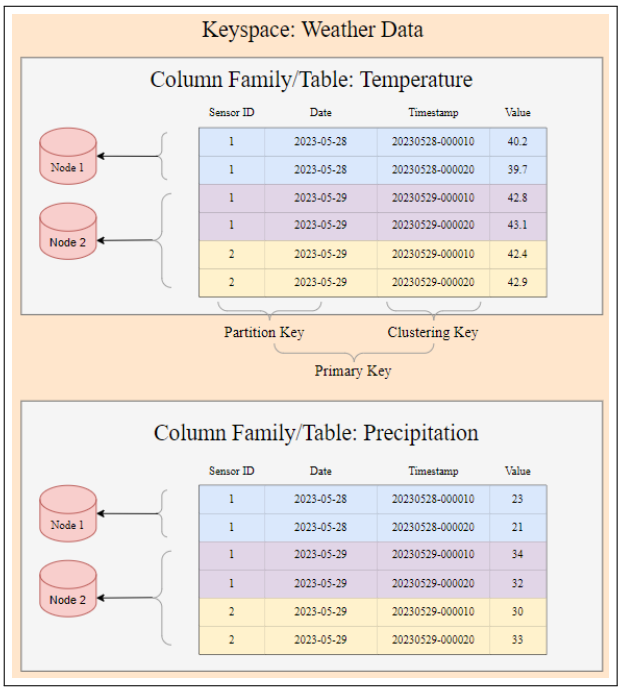


Figure 2.3: Cassandra Data Model

The following are the key components of the Cassandra data model as illustrated in Fig. 2.3:

1. Cluster: Cassandra database is distributed over a number of interconnected computers operating together. The outermost container is referred to as the cluster. Cassandra arranges the cluster nodes in a ring format and assigns data to them.

2. Keyspace: A keyspace serves as the top-level data container. It represents a namespace that stores related column families together.

3. Column Family: A column family is the container of a set of rows. Each row contains ordered columns. The column families help us to specify the structure of the data.

4. Column: Cassandra columns have three components: a name, a value, and a timestamp. In contrast to relational databases which have a fixed number of columns for each row, Cassandra allows adding or removing columns dynamically without impacting the schema, and hence, each row can have a different set of columns.

5. Partition Key: The partition key is a subset of the primary key and is used to identify the partition on which the data must be stored. Cassandra has a distributed architecture and the data is distributed among cluster nodes on the basis of the partition key.

6. Clustering Key: The clustering key is a subset of the primary key that specifies the sorting order within a partition. It helps determine how the data is stored physically in a partition.

2.2.2 Consistency Model

Cassandra is categorized as an AP (Availability and Partition Tolerance) system according to the CAP theorem, which means it prioritizes availability and partition tolerance over strong consistency. Cassandra's consistency model allows for tunable consistency levels, giving users flexibility in trading off between data consistency and system availability. Here are some key characteristics of Cassandra's consistency model:

1. Tunable Consistency:
   Cassandra provides tunable consistency levels that allow users to control the level of consistency they require for read and write operations. Users can choose from consistency levels such as "ONE", "QUORUM", or "ALL" among others. The consistency level determines the number of replicas that must respond for a read or write operation to be considered successful. Higher consistency levels provide stronger consistency guarantees but may sacrifice system availability.

2. Read and Write Consistency:
   In Cassandra, read and write operations can have different consistency levels. For example, a user can configure a write operation to require a consistency level of "QUORUM", meaning that a majority of replicas must acknowledge the write before it is considered successful. On the other hand, a read operation can have a lower consistency level, such as "ONE", where only a single replica needs to respond to fulfil the read request.

It is important to note that while Cassandra provides availability and allows continued operation during network partitions or failures, it may exhibit inconsistent behavior.

Cassandra's AP characteristics make it well-suited for use cases where high availability and scalability are crucial, such as content delivery networks, real-time analytics, and social media platforms. The ability to adjust consistency levels allows users to strike a balance between data consistency and system responsiveness, tailoring the database behavior to their specific requirements.

2.2.3 Architecture Implementation

The distinctive design of Cassandra is influenced by the Dynamo and BigTable data models, combining their advantages to offer high availability, linear scalability, and data distribution across numerous nodes [2]. This section discusses the practical aspects of Cassandra's architecture implementation, taking into account design considerations and components which promote its successful deployment.

Partitioning is a fundamental aspect of Cassandra's architecture, which enables efficient data distribution and horizontal scalability. A partitioner is used to partition the data in Cassandra, by generating a token for each partition based on the partition key. The partitioner uses a special hashing algorithm called consistent hashing that guarantees that data is distributed equally throughout the cluster, and provides efficient means to locate data based on its key. The same is illustrated in Fig. 2.4.

The peer-to-peer, decentralised architecture of Cassandra allows nodes to connect with one another in order to exchange metadata and coordinate operations. Each individual cluster node is in charge of storing and managing one or more partitions. As new nodes are added to the cluster, Cassandra can scale linearly by

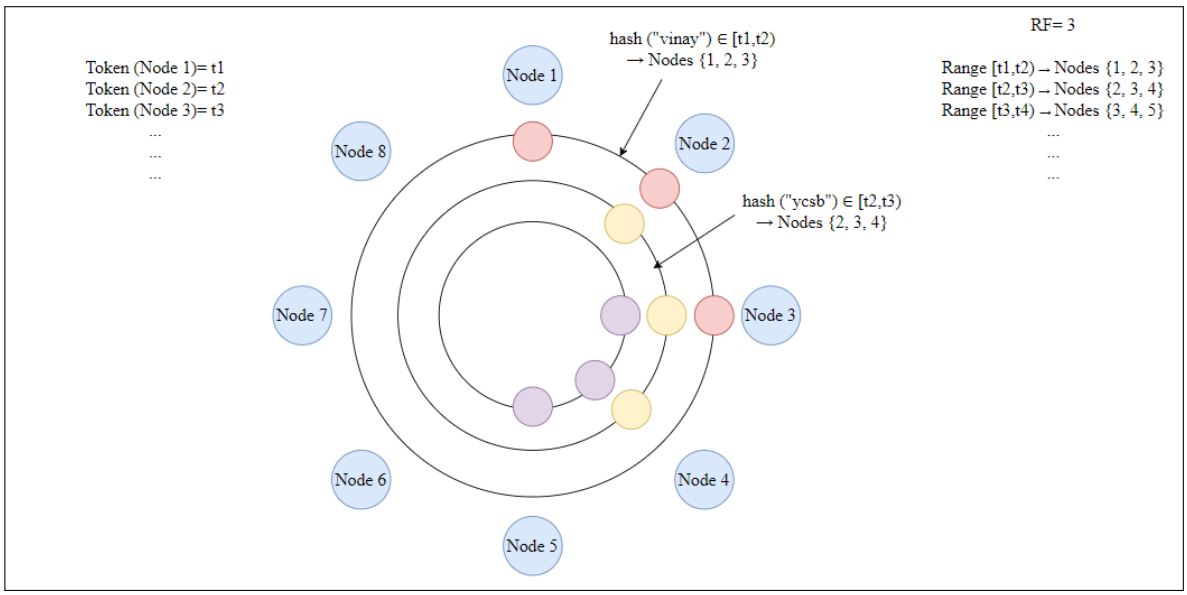


Figure 2.4: Consistent Hashing using a Token Ring

automatically redistributing data among the nodes and ensuring that the workload is divided equally. Scaling enables greater storage capacity and increased read and write throughput. Cassandra supports smooth addition and removal of nodes, allowing clusters to scale up or down with zero downtime.

Replication is a crucial component of the Cassandra architecture which guarantees data availability, load balancing, durability and fault tolerance. Cassandra users can configure the replication factor, which specifies how many replicas of the data are stored across the cluster nodes. Replication provides redundancy and ensures data durability in the event of node failures. Cassandra supports different replication strategies like Simple Strategy and Network Topology strategy.

Cassandra employs several mechanisms to provide fault tolerance. Cassandra utilises a gossip protocol to detect and disseminate information about the state of the cluster nodes in event of node failures. The system automatically promotes one of the replicas to handle read and write operations when a node goes down, maintaining continuous availability. Cassandra's architecture allows seamless recovery and ensures that data is accessible even in the face of failures.

By comprehending and effectively using these architectural components, developers can build robust and scalable applications using Cassandra.

2.2.4 Cassandra Access Path for Write Operation

The write path in Apache Cassandra refers to the procedure by which data is written to and stored in the database. In order to guarantee durability, availability, and high performance for write operations, it involves a number of steps and components that interact with one another [6].

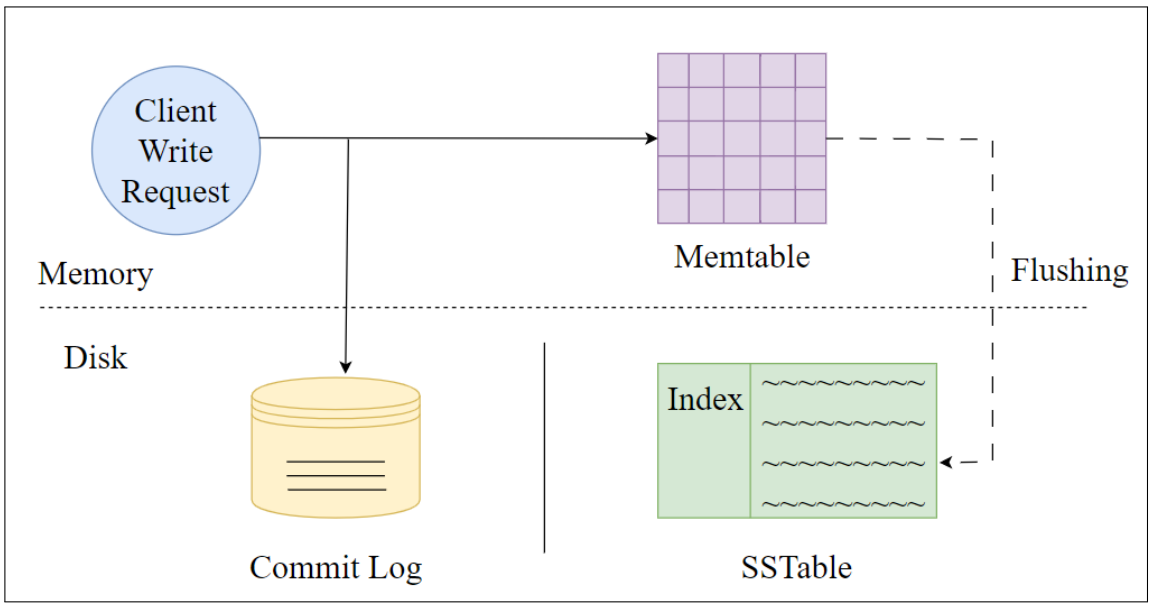


Figure 2.5: Cassandra Write Path

The steps listed below make up the typical write path in Cassandra as shown in Fig. 2.5:

1. Client Request: A client sends a write request to a Cassandra node to initiate a write operation. The client specifies the keyspace, the table, and data to be written.

2. Partitioning: Based on the partition key, Cassandra utilises a distributed hash function to identify the node that is responsible for storing the data. The partition key, which is generally a component of the primary key, aids in determining how data is distributed across the cluster nodes.

3. Replication: Cassandra offers high availability and is designed to be distributed. Using the replication strategy and the replication factor specified for the keyspace, Cassandra duplicates the data across multiple cluster nodes after identifying the coordinator node. Each replica node holds a copy of the data to offer fault tolerance and data durability.

4. Commit Log: After data replication, the involved nodes store the data to their commit log on disk. The append-only commit log ensures durability in the event of node failures.

5. Memtable: Cassandra uses an in-memory structure called the memtable to temporarily store data. The involved nodes update the memtable with the new data, and subsequent writes with the same partition key are added to the memtable as well. This lowers disk I/O and enables efficient writes.

6. SSTables: The memtable is periodically flushed to disk as an SSTable (Sorted String Table). The SSTable is an immutable file that contains sorted data for a certain range of partition keys. To optimize disk space and read performance, SSTables are compacted periodically.

7. Acknowledgement: An acknowledgment is given back to the client when the data has been successfully written to the commit log and memtable of the appropriate number of nodes as specified by the consistency level. The data can be now regarded as durable.

By adhering to this write path, Cassandra guarantees durability, fault tolerance and efficiency for write operations. It makes use of distributed data storage, replication, and memory-based structures to offer high-performance writing capabilities, even in large-scale distributed systems.

2.2.5 Cassandra Access Path for Read Operation

The read path in Apache Cassandra refers to how data is retrieved from the database. Cassandra performs a series of steps to effectively locate and retrieve the data whenever a client submits a read request [6].

Here is a summary of the typical read path in Cassandra as shown in Fig. 2.6:

1. Client Request: A read operation starts when a client submits a read request to a Cassandra coordinator node. The request specifies the keyspace, the table and the criteria for the data to be retrieved.

2. Partitioning: The coordinator node uses the partitioner to identify the replica nodes and ensures that there are enough replicas to satisfy the specified consistency setting.

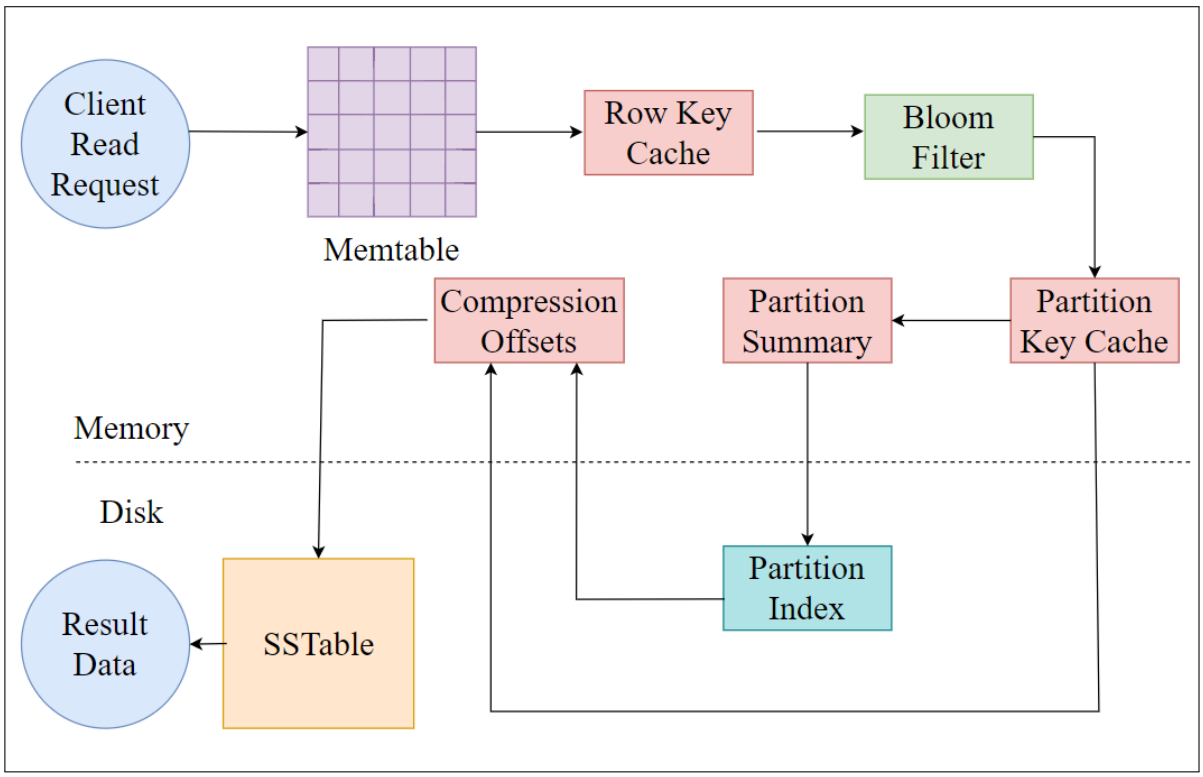


Figure 2.6: Cassandra Read Path

3. Node Coordination: If the desired data is found on the coordinator node, data retrieval happens instantly. If not, the coordinating node makes contact with the relevant replica nodes to obtain the data. The data from the fastest replica can be returned if the replicas are consistent and the desired consistency level has been achieved. Otherwise, the coordinator node must carry out a read-repair.

4. Bloom Filters, Caches and Indexes: Cassandra makes use of bloom filters, caches and indexes to optimize the search process. If the data resides in the cache, it can be returned immediately. Bloom filters are probabilistic data structures that can determine whether the SSTable could contain the required data. Data within the SSTables may be efficiently located with the aid of indexes, such as the partition index and secondary indexes.

5. SSTable Lookup: Based on the partition key and any further criteria specified in the read request, the coordinating node conducts a lookup in the SSTables. To locate the necessary data, it scans all the relevant SSTables. The primary key is used in SSTables to order the data, which facilitates quick lookups.

6. Data Retrieval and Merging: After determining the relevant SSTables, the

coordinator node obtains the data and, if required, merges it by selecting the value with the latest timestamp for each requested column. Cassandra uses a procedure known as read repair to maintain consistency when data from several replica nodes has to be combined.

7. Response to the Client: As a last step, the coordinator node replies to the client's read request by sending the data it has just retrieved.

By adhering to this read path, Cassandra optimizes data retrieval by leveraging its distributed architecture, data partitioning, replication, bloom filters, caches and indexes. This approach enables Cassandra to handle read operations efficiently, even in large-scale distributed systems.

2.3 HBase

HBase is an Apache Hadoop-based open-source NoSQL database management system. It is intended to store and handle massive volumes of structured and semi-structured data in a distributed fashion. HBase is built on the Google BigTable paradigm, which enables automated data replication and sharding across several nodes of a cluster [13]. The key characteristics of HBase include high write throughput, low latency and a fault-tolerant design. Since HBase is a column-oriented database, it stores and accesses data by columns rather than by rows. This facilitates storage and retrieval of massive volumes of data.

2.3.1 Data Model

Data storage in HBase is column oriented, and takes the form of a multi-hierarchical Key-Value map [3]. The HBase data model is extremely adaptable, and one of its best features is the ability to add or remove column data as per requirement without affecting the performance. It is possible to process semi-structured data using HBase. There are no specific data types since the data is stored in bytes.

The following are the key components of the HBase data model as illustrated in Fig. 2.7:

1. Table: In HBase, a table is mostly logical rather than physical. A collection of rows forms an HBase table. The table's data is distributed across multiple regions using the range of rowkey.

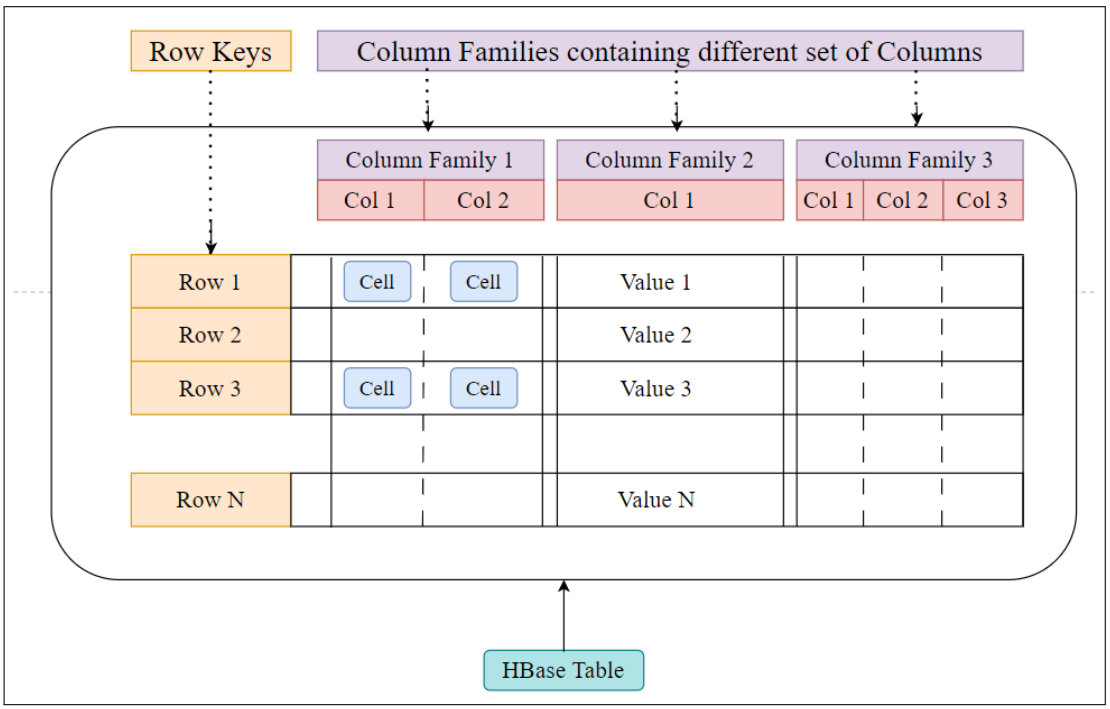


Figure 2.7: HBase Data Model

2. Row: In HBase, a row is only a logical representation. Physically, the data is stored in columns rather than rows. Rows are combinations of column families. A rowkey serves as a primary key index that is used to uniquely identify each row.

3. Column Family: Columns in HBase are organised into column families, which are collections of related columns. Column families are created during schema definition and are defined at the table level. Each row within a column family may have a different set of columns, but all rows in a table share the same set of column families.

4. Column: In HBase, columns are defined within column families. Each column consists of a name, value, and timestamp. A column can be accessed by using a qualifier as column_family-name:column-name.

5. Cell: In HBase, data is written into cells. In an HBase table, the combination of rowkey, row, column, and version can be used to define a cell. The data will be saved as cell value and will be of the byte[] data type.

6. Version: In HBase, a rowkey (row, column, version) contains a cell. To allow holding more than one cell for the same row and column, we need to specify a version value. To ensure that the most recent cell values are retrieved first, HBase stores the versions in descending order.

2.3.2 Consistency Model

HBase is categorized as a CP (Consistency and Partition Tolerance) system according to the CAP theorem, which states that in the event of a network partition, a distributed system must choose between consistency and availability. HBase prioritizes consistency over availability, making it a CP system.

As a CP system, HBase focuses on ensuring strong consistency and data integrity across replicas, even in the presence of network failures or partitions. Here are some key characteristics of HBase as a CP system:

1. Row-Level Strong Consistency:
   HBase provides strong consistency guarantees for single-row operations. When a client performs a read or write operation on a single row, HBase ensures that subsequent read operations from the same row will reflect the effects of the previous write. This guarantees that clients always observe a consistent view of the data within a row.

2. Consistent Replication:
   HBase employs replication to achieve fault tolerance and data durability. It maintains multiple replicas of data across different nodes in the cluster. While replication is typically asynchronous, HBase ensures that all replicas eventually catch up and maintain consistency. It employs mechanisms like write-ahead logs (WALs) and distributed commit logs to synchronize replicas and maintain data consistency.

HBase is designed to handle network partitions and failures without experiencing significant unavailability. It automatically recovers and ensures strong consistency across replicas once the network issues are resolved. This robustness makes HBase suitable for applications where continuous availability and data integrity are crucial.

HBase's CP characteristics make it suitable for use cases where data consistency and integrity are of utmost importance, such as financial systems and e-commerce platforms. However, it's worth noting that achieving strong consistency may introduce some latency overhead compared to systems with weaker consistency models.

2.3.3 Architecture Implementation

HBase's architecture has its foundation laid in its close integration with HDFS, a distributed file system intended for storing massive amounts of data across a cluster of commodity hardware. HDFS serves as the foundational storage layer for HBase, guaranteeing fault tolerance, high availability, effective data distribution and strong consistency. This section examines the practical aspects of implementing the HBase architecture, with a focus on the design considerations and components necessary for successful deployment.

HBase uses a partitioning scheme called sharding to distribute data horizontally across the cluster [13]. In HBase, data is organised into tables that are divided into regions. Each region is stored on a particular region server and consists of a range of row keys. Row keys are used for partitioning of data, and HBase uses a hashing algorithm to map row keys to relevant regions and region servers. The same is illustrated in Fig. 2.8.

Partitioning allows HBase to scale well by distributing the workload and data across several region servers. It enables read and write operations to be processed in parallel, enabling efficient data retrieval and updates. Additional region servers may be added to the cluster to accommodate the growth as the dataset or the workload increases, maintaining linear scalability. Whenever a new region server is added, regions are automatically split and distributed throughout the expanded cluster to ensure effective resource utilization and data balance.

Replication is another critical feature of HBase's architecture, which offers data redundancy, load balancing, and enhanced performance. HBase offers asynchronous, cross-datacenter replication, enabling users to replicate data to remote clusters for disaster recovery or geographically dispersed deployments. The replication mechanism in HBase makes sure that modifications made to the primary regions are propagated to the replica regions. Replication makes it possible to serve read operations from the local replica, lowering network latency and enhancing read performance. Users can specify which tables and column families are replicated, and to which remote clusters, by configuring the replication scope.

Replication of data across multiple region servers provides fault tolerance and high availability. The number of replicas stored in the cluster is determined by the replication factor configured for each region. Replication ensures data acces-

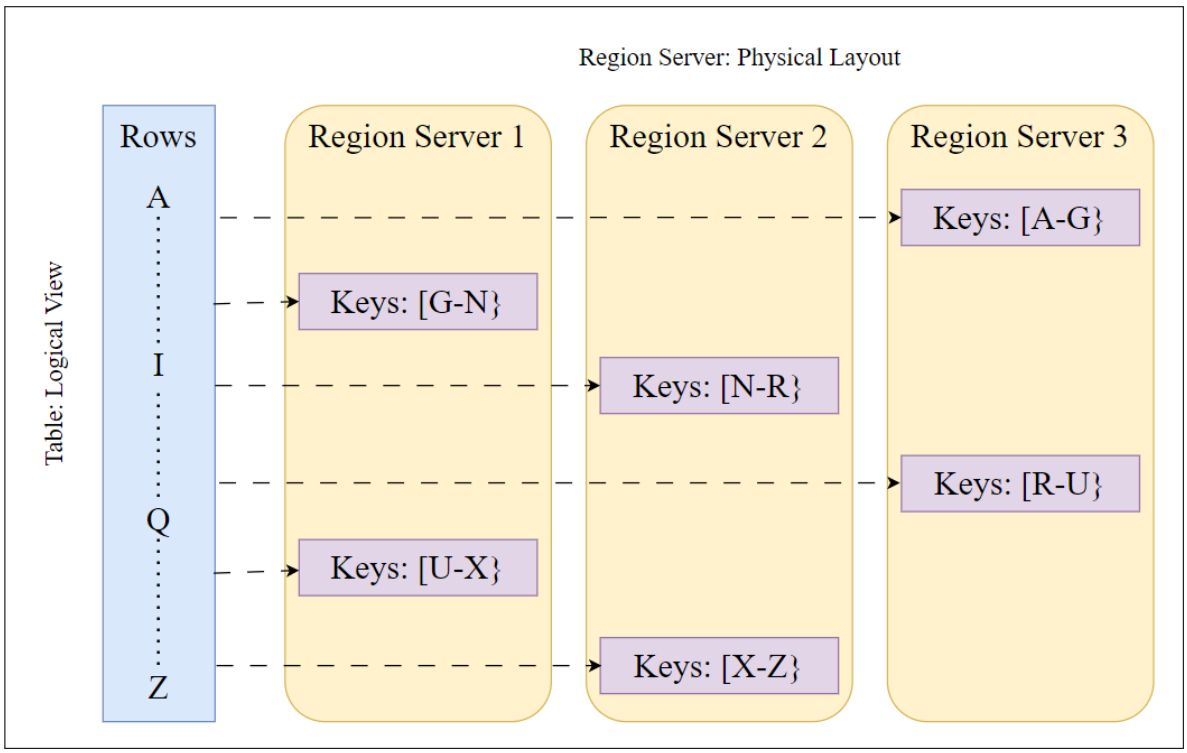


Figure 2.8: Rows grouped into Regions and managed by different Region Servers

sibility and durability in case of region server failures. HBase automatically promotes one of the replica regions to replace the failed region server in the event of a failure. This failover mechanism ensures high availability and seamless recovery. Furthermore, HBase also uses a distributed coordination service like Apache ZooKeeper to manage metadata and enable coordination among the distributed components.

A comprehensive understanding of these architectural elements empowers developers to harness the full potential of HBase.

2.3.4 HBase Access Path for Write Operation

The write path in Apache HBase refers to the procedure by which data is written to and stored in the database. When a client sends a write request to HBase, it goes through a series of steps to ensure data durability and availability [5].

The steps listed below make up the typical write path in HBase as shown in Fig. 2.9:

1. Client Request: A write operation starts when a client sends a write request to HBase.

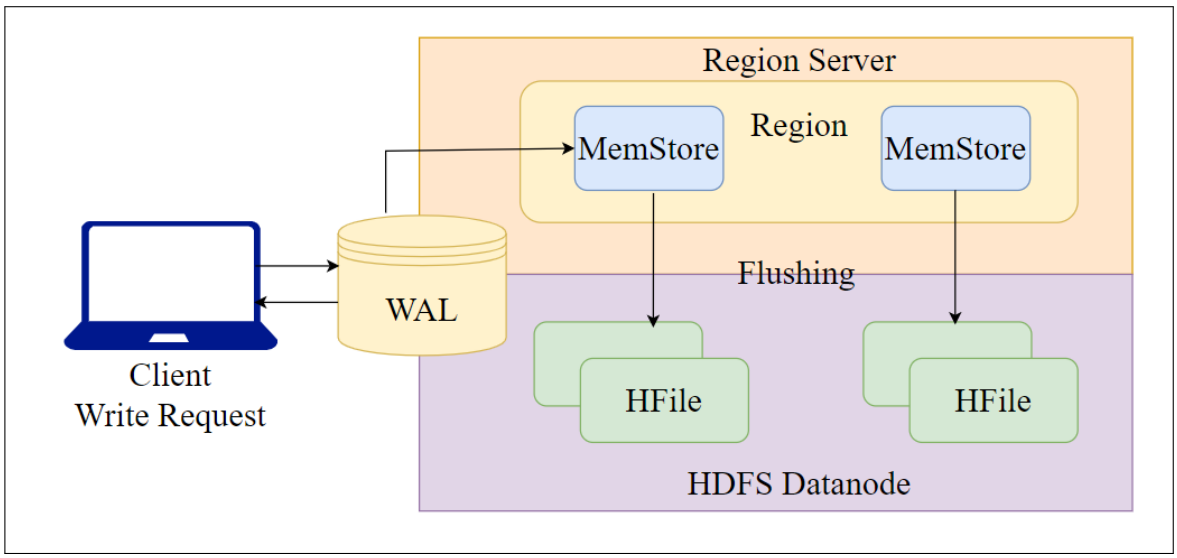


Figure 2.9: HBase Write Path

2. Zookeeper: In order to find the location of the region server(s) in charge of the data being written, the client contacts ZooKeeper.

3. Region Server: The client communicates directly with the region server(s) responsible to handle the data.

4. Write-Ahead Log (WAL): The region server initially writes the data to the append-only Write-Ahead Log (WAL), which is stored on the disk. The WAL assures durability by recording all changes before they are applied to the data files.

5. MemStore: After writing to WAL, the region server writes the data to an in-memory data structure called the MemStore, which allows for quick access to newly written data.

6. HFile Flush: The region server periodically flushes the MemStore to disk as a new HFile when it fills up in memory. This ensures durability and also makes room for the subsequent writes by freeing up memory in the MemStore.

7. Compaction: HFiles accumulate on disk over time, and HBase periodically performs compaction to improve performance. Compaction merges many HFiles, discards expired or deleted data, and improves storage.

8. Replication and WAL Replay: To ensure fault tolerance, the changes written to the WAL are replicated to other region servers. In the event of failures, the data can be recovered by replaying the WAL.

9. Acknowledgement to the Client: Once the data has been written to the WAL and MemStore, and the necessary replication and flushing operations have been finished, the region server sends an acknowledgment back to the client, indicating that the write operation was successful.

By adhering to this write path, HBase ensures durability, fault tolerance, and efficient data storage. The use of WAL, MemStore, flushing, compaction, and replication mechanisms helps HBase provide reliable and high-performance write operations in distributed environments.

2.3.5 HBase Access Path for Read Operation

The read path in Apache HBase refers to how data is retrieved from the database. HBase performs a series of steps to effectively locate and retrieve the data whenever a client submits a read request [5].

The high level read architecture of HBase is shown in Fig. 2.10. Here is a summary of the typical read path in HBase:

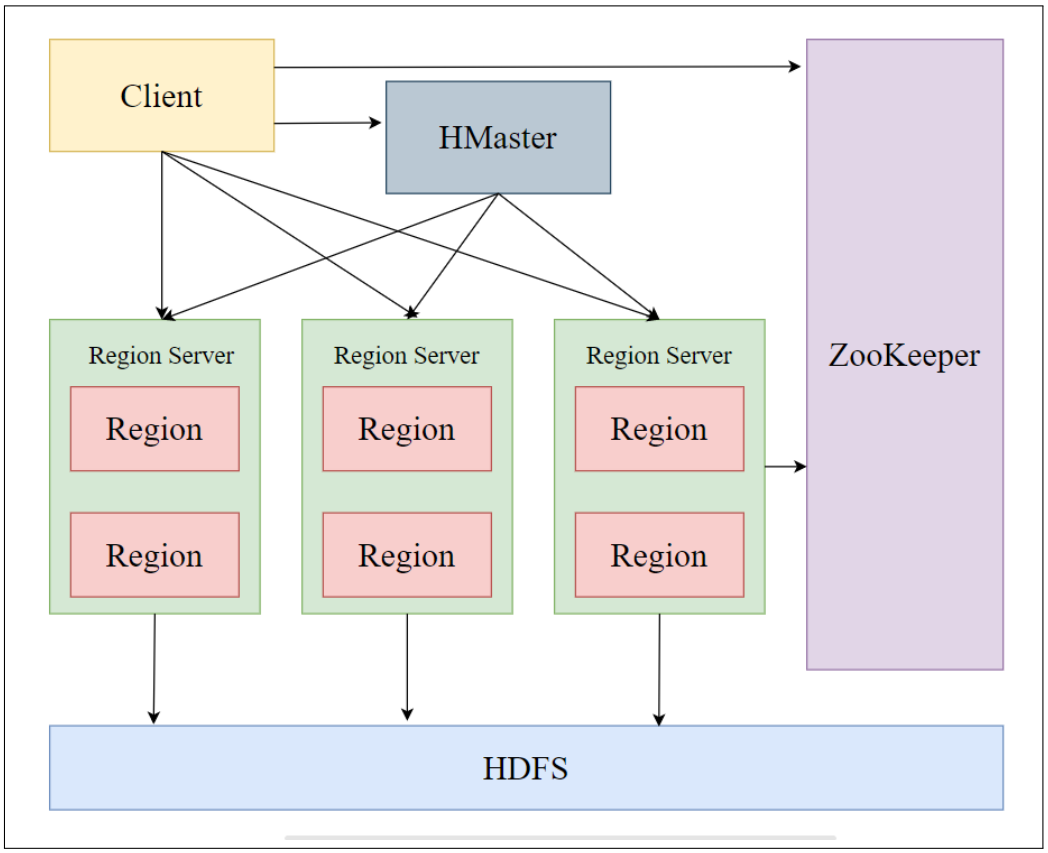


Figure 2.10: HBase Read Path

1. Client Request: A read operation gets initiated when a client submits a read request to an HBase region server. The request specifies the table name, the row key, and possibly any particular columns or filters for the data to be retrieved.

2. Region Assignment: HBase employs a distributed architecture and assigns the data to different regions. Based on the row key, the region server in charge of the requested data's region is identified.

3. Block Cache: The region server initially looks for frequently accessed data blocks in the block cache, a memory-based cache. If the required data is present in the cache, it can be sent to the client right away.

4. MemStore: If data is not available in the block cache, the region server searches the in-memory data structure called MemStore that contains the recently written data. The MemStore is set up as a sorted map of key-value pairs.

5. HFile Lookup: If the data is not found in the MemStore, the region server searches the HFiles. HFiles are persistent storage files where the data is stored on disk. They are organized as data blocks sorted by row key ranges.

6. Bloom Filters and Indexes: HBase makes use of Bloom filters and indexes to optimize the search process. Bloom Filter is a probabilistic data structure that is used to determine whether an HFile contains the requested data. Indexes, such the row index and block index, make it easier to locate the data within the HFiles.

7. Block Cache Population: If the requested data is found in the HFiles, the region server populates the block cache with the relevant data blocks for serving future read requests.

8. Data Retrieval: After locating the requested data, the region server retrieves it and assembles it into a response.

9. Response to the Client: Finally, the region server sends the retrieved data to the client as a response to the read request.

By adhering to this read path, HBase optimizes data retrieval by utilizing block caching, in-memory storage (MemStore), HFile lookup, Bloom Filters, and indexes. This approach enables HBase to handle read operations efficiently and provide low-latency access to the requested data.

2.4 Comparative Summary

Table 2.1 provides key differences between Cassandra and HBase.

Aspect	Cassandra	HBase
Base of Database	Amazon DynamoDB	Google BigTable
Architecture	Peer-to-Peer	Master-Slave
Single Point of Failure	No	Master Node
Disaster Recovery	Possible	Possible only if more than one master node exist
CAP Theorem	AP model	CP model
Consistency	Tunable	Strong
Partitioning	Consistent Hashing	Range-Based
Concurrency Control using Locks	No	Yes
Internode Communication	Gossip Protocol	ZooKeeper protocol
Query Language	SQL like language called CQL	API based approach for data access and manipulation
Cluster Setup	Easy	Difficult
HDFS Compatibility	No	Yes
Use by Companies	eBay, GitHub, Netflix, Facebook	Adobe, Flipkart, Spotify, Twitter

Table 2.1: Cassandra vs HBase

CHAPTER 3
Literature Survey

Column Family databases have emerged as one of the most effective NoSQL databases for managing large-scale data storage and retrieval. By offering high scalability, fault tolerance, and distributed architecture, column family databases are designed to overcome the drawbacks of conventional relational databases. Apache Cassandra and Apache HBase are two well-known column family databases that have gained considerable attention recently.

To fulfil the thesis objective, this literature survey will explore existing research and publications related to Cassandra and HBase. We will examine studies that have utilized YCSB to assess Cassandra's performance and understand how various system parameters affect the performance metrics. We will also examine studies that have compared the performances of Cassandra and HBase, to uncover any gaps or limitations in the existing research. By integrating and analyzing the available literature, this thesis intends to contribute to the corpus of knowledge regarding the performance evaluation of Cassandra using YCSB and the comparative performance analysis of Cassandra and HBase.

3.1 Yahoo! Cloud Serving Benchmark (YCSB)

One of the most often used benchmarks is the Yahoo! Cloud Serving Benchmark (YCSB), which provides benchmarking for the basis of apples-to-apples comparison across NoSQL systems [7]. It was originally developed by Yahoo! research and has since gained popularity in industry as well as academic research.

The YCSB framework is made up of a workload generator and a set of predefined workloads. The workload generator mimics the behaviour of a client application by producing a mix of read and write requests in accordance to the specified workload. The predefined workloads reflect diverse application scenarios and

access patterns by including varying distributions of read and write operations. When running a benchmark, YCSB will record the latency in conducting these operations as well as the throughput in operations per second. The primary purpose of the Yahoo! Cloud Serving Benchmark is to create tiers to assess scalability, availability, elasticity, replication and performance of cloud serving systems.

3.1.1 YCSB Workloads

Table 3.1 illustrates the YCSB preconfigured workloads [9].

Workload	Operations	Record Selection	Applications
A-Update Heavy	Read: 50% Update: 50%	Zipfian	Session store recording recent actions in a user session
B- Read Mostly	Read: 95% Update: 5%	Zipfian	Photo tagging: adding a tag is an update, but most operations are to read tags
C- Read Only	Read: 100%	Zipfian	User profile cache, where profiles are constructed elsewhere (Ex: Hadoop)
D-Read Latest	Read: 95% Insert: 5%	Latest	User status updates: people want to read the latest statuses
E- Short Ranges	Scan: 95% Insert: 5%	Zipfian/ Uniform	Threaded Conversations, where each Scan is for the post in a given thread (assumed to be clustered by thread id)
F- Read-Modify-Write	Read: 50% Read/Modify/Write: 50%	Zipfian	User Database, where user records are read and modified by the user, or to record user activity

Table 3.1: Workload Characterization for YCSB

The 'Record Selection' column indicates the distribution strategy used for determining which records from the dataset will be accessed or operated upon during the execution of the workload. The record selection strategies available in YCSB include:

- Uniform Distribution: This strategy selects records uniformly at random from the dataset. Each record has an equal probability of being accessed, resulting in a flat access pattern.
- Zipfian Distribution: This strategy is used to mimic real-world scenarios where a small number of records are accessed frequently, while the majority of records are accessed less frequently. It follows a power-law distribution,

where a few records have high probabilities of being accessed, while the rest have decreasing probabilities.

- Latest Distribution: This strategy focuses on accessing the most recently added or updated records in the dataset. It is often used to simulate scenarios where the most recent data is of higher interest or relevance.

The 'Applications' column indicates the specific use case based on the type of the application pattern the workload simulates.

3.1.2 System Core Properties

The following is the list of properties that may be tuned to improve system performance [8]:

1. Workload: It specifies the YCSB Workload class to be used.
2. Record Count: It specifies the number of records in the dataset when running the workload.
3. Operation Count: It specifies the number of operations to be performed in the workload.
4. Thread Count: It specifies the number of YCSB client threads involved in running the workload.
5. Consistency Level: It specifies the minimum number of cluster nodes that must acknowledge a read/write operation before it is considered as succeeded.
6. Field Count: It specifies the number of fields in the database record.
7. Field Size: It specifies the size of each field in the database record.
8. Replication Factor: It specifies the number of replicas of the data across the cluster.
9. Request Distribution: It specifies the distribution strategy that should be used to select the records for performing the CRUD operations.
10. Cluster Size: It specifies the number of nodes in the database cluster whose performance is to be assessed.

3.2 Performance Evaluation of Cassandra

3.2.1 Benchmarking Replication and Consistency strategies in cloud serving databases: HBase and Cassandra.

Wang et al. were motivated by the trade-off between consistency and latency to investigate how the latency varies when the replication factor and consistency level are modified [24]. In order to accomplish this task, the authors used YCSB for conducting benchmarking efforts for two most frequently used systems: HBase and Cassandra. The experiments used 16 server-class machines of the same rack as the testbed. The machines had the following specifications: two Xeon L5640 64 bit processors (each processor owned 6 cores and each core owned 2 threads), 32 GB of RAM, one hard drive and gigabit ethernet connection. The following conclusions were obtained in regard to Cassandra: (i) More number of replicas can deteriorate read performance when using a lower consistency level. (ii) Higher consistency levels can dramatically increase the write latency. (iii) Higher consistency levels are not suitable for write heavy and read latest workloads.

3.2.2 Interplaying Cassandra NoSQL consistency and performance: A benchmarking approach

The primary goal of Gorbenko et al. was to investigate the relationship between Cassandra performance (response time) and consistency settings [14]. The paper describes the read and write performance benchmarking findings for a replicated Cassandra cluster installed in the Amazon EC2 Cloud. The authors set up a 3-replicated Cassandra 2.1 cluster in the Amazon EC2 cloud as a testbed. The cluster was deployed with the following specifications: AWS US-West-2 (Oregon) region, c3.xlarge instances (vCPUs– 4, RAM– 7.5 GB, SSD– 2x40 GB, OS– Ubuntu Server 16.04 LTS). For experimentation, the authors used the YCSB read-only Workload C and the Workload A, which had been configured to do write-only operations. One of the key results obtained by the authors is that by optimally coordinating consistency settings for both read and write requests, we may reduce Cassandra delays while maintaining strong data consistency. Their experiments demonstrate that: (i) The performance metrics throughput and read/update latency increase with an increase in number of threads. (ii) Strong consistency costs up to 25% of performance. (iii) The ideal option for strong consistency is dependent on the read/write operation ratio.

3.2.3 Automatic configuration of the Cassandra database using Irace

By utilizing the YCSB benchmark under various configurations, authors Silva-Munoz et al. employed the Irace configurator to automatically determine the optimal parameter setup for the Cassandra NoSQL database [20]. The authors performed the experiments with the help of Google Cloud infrastructure using n1-standard-8 machines, with eight virtual CPUs, 30 GB of memory and a 20GB persistent disk. The version of Cassandra benchmarked was 3.6. The authors made the following observations: (i) The throughput increases with an increase in thread count. (ii) Upon increasing the record count, the throughput decreases and the read/update latency increases. (iii) Upon increasing the operation count, the throughput as well as the read/update latency increases. With a budget of 2000 experiments, the Irace configurator serves as a straightforward general-purpose tool that can achieve throughput speedups of up to 30% compared to the default configuration.

3.3 Comparative Performance Analysis of Cassandra and HBase

3.3.1 Benchmarking Cloud Serving Systems with YCSB

Cooper et al. have presented the Yahoo! Cloud Serving Benchmark. This benchmark is designed to offer resources for direct comparisons of various cloud serving data stores [10]. With the use of this tool, they were able to compare the performance of four cloud-serving systems: Cassandra, HBase, PNUTS, and sharded MySQL. The following were the specifications used for the experimentation: six server-class machines with dual 64-bit quad core 2.5 GHz Intel Xeon CPUs, 8 GB of RAM, 6 disk RAID-10 array and gigabit ethernet. The authors found that each system's architectural choices had a clear impact on the read and write performance tradeoffs. The following were the observations in regard to the comparison of Cassandra and HBase: (i) For update-heavy workloads: Read Latency $_{\text{(Cassandra)}} \approx 0.4$ Read Latency $_{\text{(HBase)}}$ and Update Latency $_{\text{(Cassandra)}} \approx 7$ Update Latency $_{\text{(HBase)}}$. (ii) For read-heavy workloads: Read Latency $_{\text{(Cassandra)}} \approx 0.5$ Read Latency $_{\text{(HBase)}}$ and Update Latency $_{\text{(Cassandra)}} \approx 5$ Update Latency $_{\text{(HBase)}}$.

3.3.2 A comparison between several NoSQL databases with comments and notes

Authors Tudorica et.al. have tried to make a comparison between Cassandra, HBase and MySQL using qualitative and quantitative measures [23]. The qualitative comparison is performed by considering features such as persistence, replication, availability, transactions, etc. For quantitative evaluation, the authors have considered system parameters- dataset size and workload. The experiments were conducted by running YCSB benchmark on 120 million records of small size (1kB) on a 6-node cluster with 0.12 TB equivalent of installations of the three systems. The following were the observations in regard to the comparison of Cassandra and HBase: (i) For update-heavy workloads: Read Latency $_{(Cassandra)} \approx$ 0.7 Read Latency $_{(HBase)}$ and Update Latency $_{(Cassandra)} \approx$ 7 Update Latency $_{(HBase)}$. (ii) For read-heavy workloads: Read Latency $_{(Cassandra)} \approx$ 0.64 Read Latency $_{(HBase)}$ and Update Latency $_{(Cassandra)} \approx$ 7 Update Latency $_{(HBase)}$. This is indicative of the fact that they cannot be used interchangeably for any given situation, but one must rather decide between these systems for ensuring optimal performance.

3.3.3 Quantitative Analysis of scalable NoSQL databases

Authors Swaminathan et.al. have discussed that NoSQL databases are quickly replacing the relational databases as the go-to databases for big data applications [22]. They have evaluated three most commonly used NoSQL databases: Cassandra, HBase and MongoDB using the YCSB benchmark. The experiments were conducted using the following specifications: 14 servers connected through 1 Gigabit Ethernet Switch where each server consists of Xeon CPU, 4 GB RAM, 1.4 TB hard disk and CentOS operating system. The node acting as master had a 2.7 TB hard disk. The configurations of the databases were as mentioned: Cassandra (2.0.16), sharding: Murmur3Partitioner, key cache: 100 MB, row cache: None; HBase (1.1.0), sharding: auto, Hadoop (2.2.0). A total of 6 (Cluster sizes) * 5 (Workloads) * 4 (Dataset sizes) * 4 (Operation counts) = 480 experiments were conducted for each database. The authors concluded that for update-heavy workloads, Throughput $_{(Cassandra)} \approx$ 1.4 Throughput $_{(HBase)}$ and for read-heavy workloads, Throughput $_{(Cassandra)} \approx$ 1.6 Throughput $_{(HBase)}$.

3.3.4 Experimental Evaluation of NoSQL Databases

Authors Abramova et.al. aimed to compare different NoSQL databases and evaluate their performance in accordance to the data storage and retrieval use cases [12]. The authors tested 10 NoSQL databases using YCSB. The experimental setup had the following specifications: virtual machine Ubuntu Server (32 bit) with 2GB RAM available, hosted on a computer with Windows 7 and a total of 4GB RAM. The version of Cassandra used was 1.2.1, and that of HBase was 0.94.10. 600,000 records, each with 10 fields of 100 bytes were inserted during the load phase. The execution of workloads was done using 1000 operations. The following were the results regarding the comparison of Cassandra and HBase: (i) For workload A, Cassandra exhibited a performance 2.7 times faster than HBase. (ii) For workload B, Cassandra exhibited a performance 1.42 times faster than HBase. Hence, Cassandra provided better throughput for both update-heavy and read-heavy workloads.

3.3.5 NoSQL evaluation: A use case oriented survey

Authors Hecht et.al. have tried to compare several NoSQL databases by their data models, query possibilities, concurrency controls, partitioning and replication opportunities [15]. The following were the observations in regard to the comparison of Cassandra and HBase: (i) Cassandra is more effective in handling complex data structures since it can have an additional dimension of super columns that can contain multiple columns and be stored within column families. (ii) Both Cassandra and HBase offer query functionalities in the form of JAVA API, query language, and Map Reduce support. HBase additionally offers REST API querying. (iii) HBase is the only column family database that supports a limited variant of locks and transactions, since the concurrency control techniques can be applied only on single rows. (iv) Cassandra datasets are partitioned horizontally using consistent hashing, whereas HBase uses range based partitioning. (v) Cassandra offers optimistic replication since it is able to configure the level of consistency that can be provided. On the other hand, HBase offers pessimistic replication, wherefore it provides strong consistency.

CHAPTER 4
Performance Evaluation of Cassandra using YCSB

The experiments were carried out on a 3-node Cassandra 3.11.13 cluster. The replication strategy used was 'Simple Strategy' and the replication factor of the cluster was set to 3. The durable writes property was set to 'True'. This ensures that data is written to the commit log. A series of YCSB load and run tests were carried out by varying the system core properties: record count, operation count, thread count, consistency level and dataset size. We have used the YCSB preconfigured Workloads A and C, and Workload G (Workload A parameterized to execute write-only operations) for performing the experiments. 10 trials of the same run test were performed and the results were aggregated to avoid any anomalies.

4.1 Experimental Setup

The specifications of the experimental setup are as follows:

- YCSB Version: 0.17.0
- Operating System: Ubuntu 18.04.3 - 64 bit
- Memory: 12 GB
- Hardware Model: Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz

We make the assumption that benchmarking is carried out on specifically designated resources and that the nodes are not performing any compute-intensive operations simultaneously at the time of our testing.

4.2 Results and Discussion

From the experiments performed, the following result graphs were obtained.

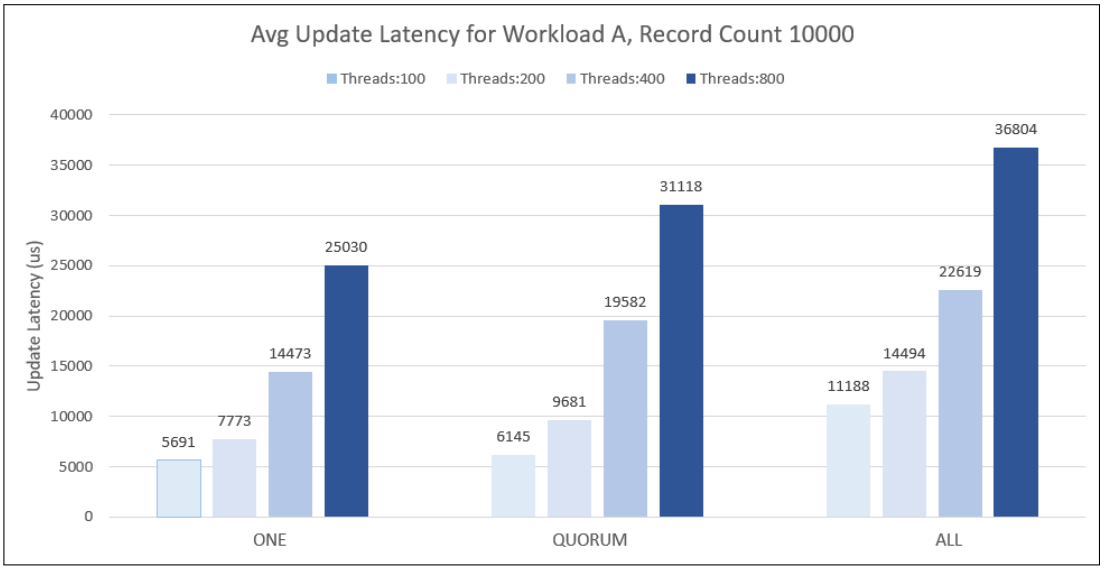


Figure 4.1: {Threads, Consistency} vs Latency

1. Observation:
   From Fig. 4.1 we can notice that for a given cluster configuration, if the number of threads is increased without varying any other parameters, then the read/update latency increases linearly with the number of threads.

   Reason:
   This is because as the number of threads increases, the system has to manage and schedule more requests concurrently, which can lead to higher contention for shared resources. This contention can cause increased waiting time for requests and result in higher read/update latency. Additionally, as the number of threads increases, the likelihood of thread synchronization overhead also increases, which can further impact latency.

   The observation is inline with [14].

2. Observation:
   From Fig. 4.1 we can notice that for a given cluster configuration, if the consistency level is strengthened without varying any other parameters, then the read/update latency increases.

   Reason:
   For a read/update operation to be considered as successful, a stronger consistency level requires data across more number of replicas to be read/updated.

Hence, the read/update latency in such a configuration is more as compared to a weaker consistency level.

The observation is inline with [14].

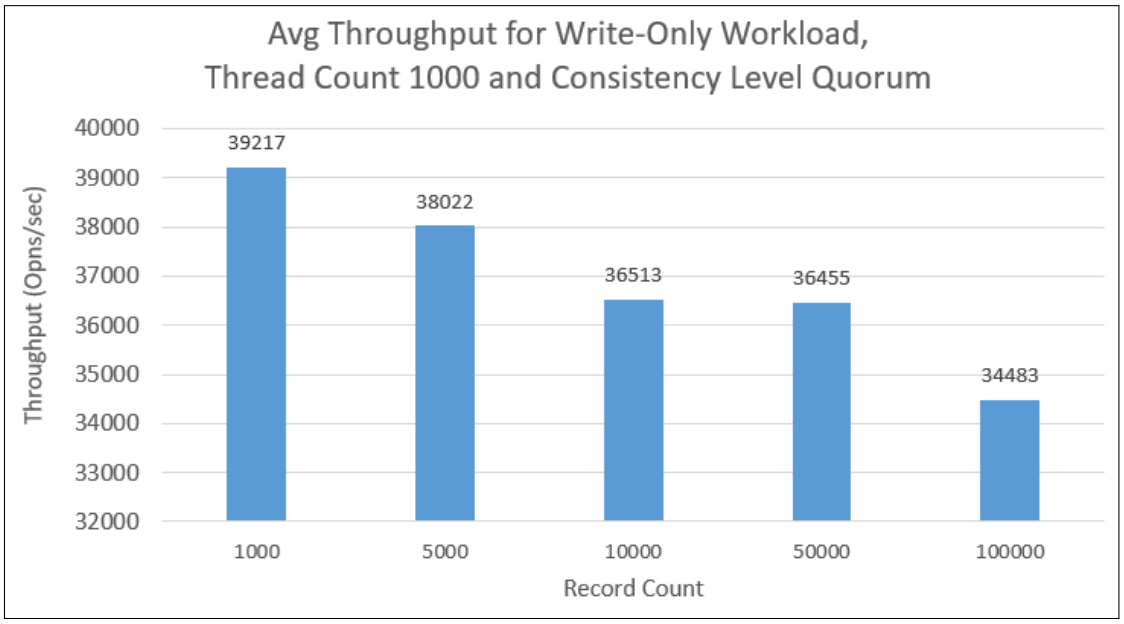


Figure 4.2: Record Count vs Throughput

3. Observation:
   From Fig. 4.2 we can notice that for a given cluster configuration, if the record count is increased without varying any other parameters, then the throughput decreases.

   Reason:
   Increasing the record count means that the total number of records that need to be searched for each read/update operation also increases. This leads to an increase in the time it takes to complete each read/update operation, which can ultimately result in decreased throughput.

   The observation is inline with [20].

4. Observation:
   From Fig. 4.3 we can notice that for a given cluster configuration, if the record count is increased without varying any other parameters, then the

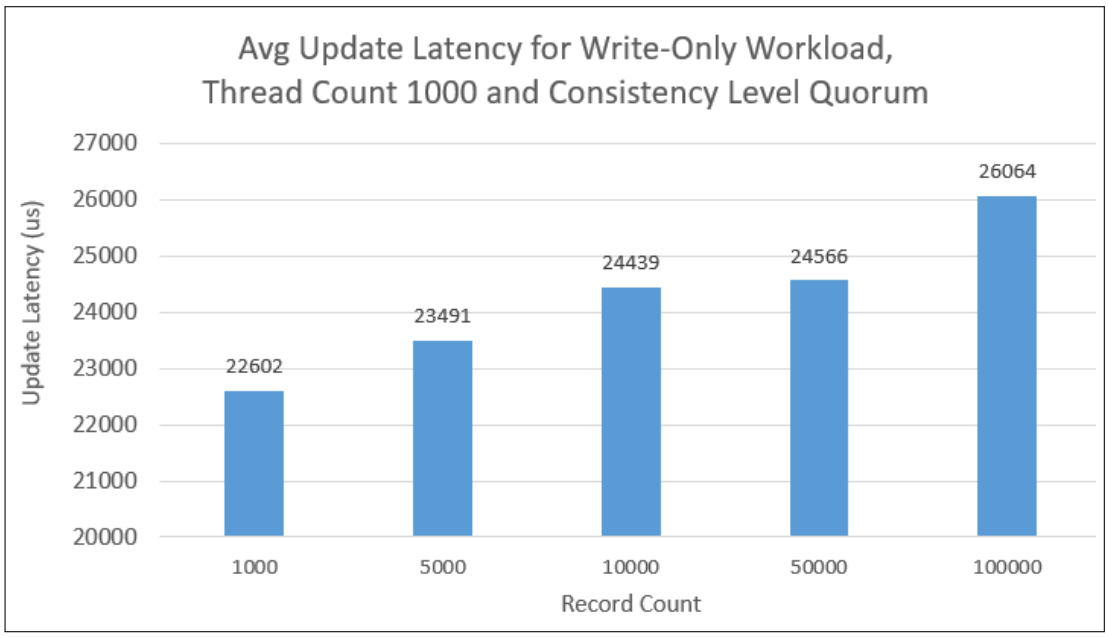


Figure 4.3: Record Count vs Latency

read/update latency increases.

Reason:
As the record count increases, the system needs to perform more disk reads to search for the required records, which can lead to increased disk I/O and, thus, longer read/update latencies.

The observation is inline with [20].

5. Observation:
From Fig. 4.4 we can notice that for a given cluster configuration, if the consistency level is strengthened without varying any other parameters, then the throughput decreases.

Reason:
This is because for a weaker consistency level, a greater number of read/write operations can be performed per unit of time as compared to a stronger consistency level.

The observation is inline with [14] and [24].

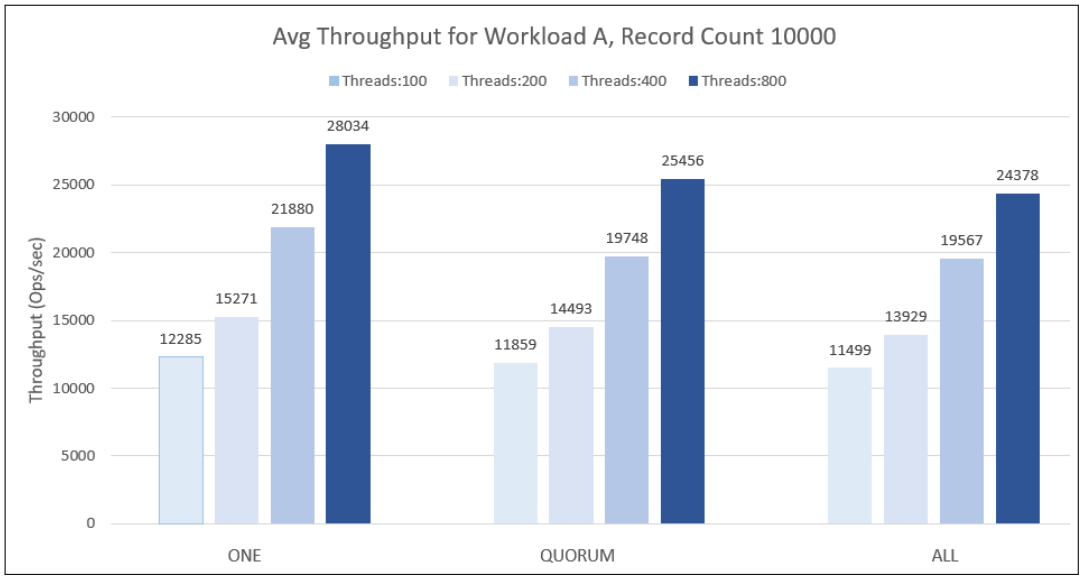


Figure 4.4: Consistency vs Throughput

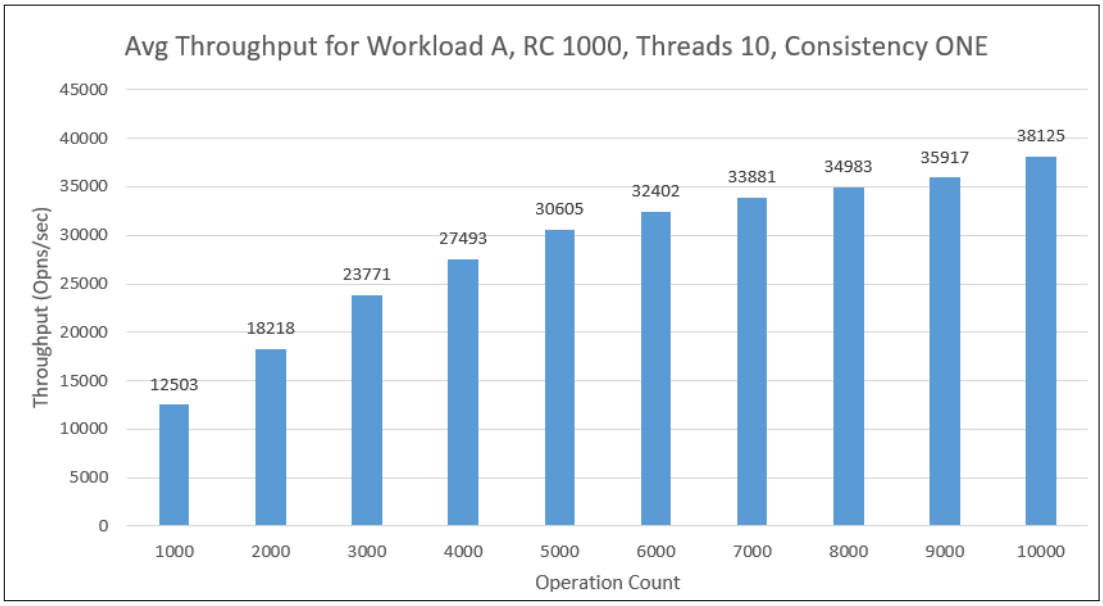


Figure 4.5: Operation Count vs Throughput

6. Observation:
   From Fig. 4.5 we can notice that for a given cluster configuration, if the operation count is increased without varying any other parameters, the throughput increases.

   Reason:
   When the operation count is increased, it can potentially lead to higher throughput, especially if the system has spare resources like CPU, network bandwidth and memory. Cassandra can effectively manage a large number of concurrent requests, and underutilized resources can be used to effectively process additional operations. So, increasing the number of operations may outcome in increased throughput.

   The observation is inline with [20].

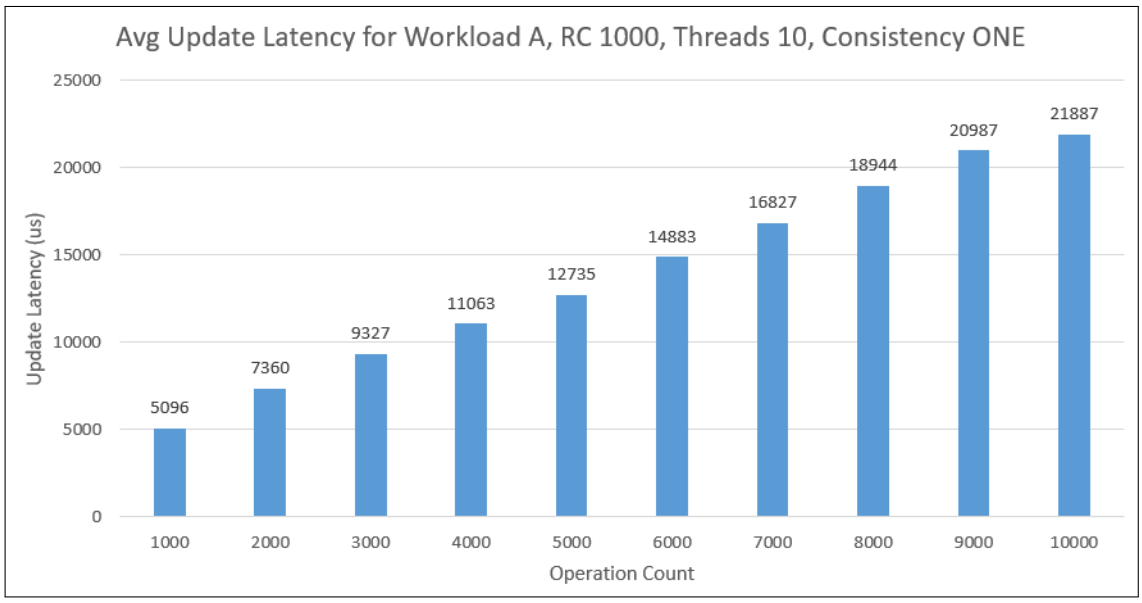


Figure 4.6: Operation Count vs Latency

7. Observation:
   From Fig. 4.6 we can notice that for a given cluster configuration, if the operation count is increased without varying any other parameters, then the read/update latency increases.

   Reason:
   The system may encounter increased latencies for individual requests as the

operation count increases. This is because the system has to handle more concurrent operations, which might result in resource contention and queuing delays. When the system is already working at or near its capacity limit, the increased workload may result in degraded response times and increased latency for both read and write operations.

The observation is inline with [20].

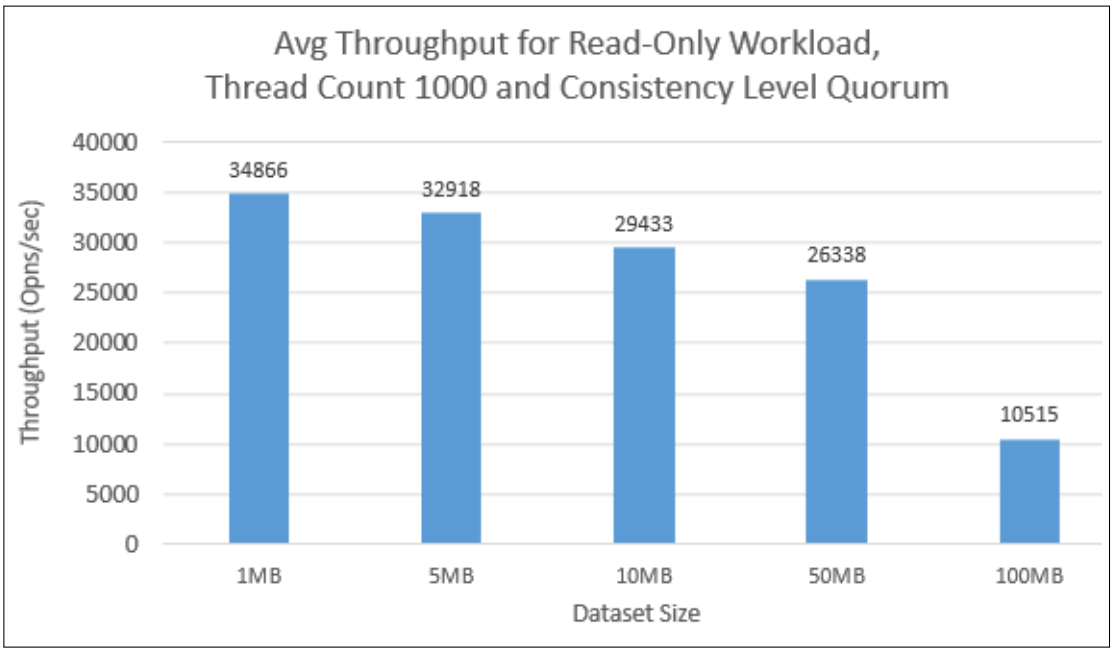


Figure 4.7: Dataset Size vs Throughput

8. Observation:
   From Fig. 4.7 we can notice that for a given cluster configuration, if the dataset size is increased without varying any other parameters, the throughput decreases.

   Reason:
   More data must be saved on the disk as the dataset gets larger. Due to more disk I/O activities caused by this increase in storage needs, the system's throughput may be impacted. Higher read and write latencies in the disk subsystem might result in slower data retrieval and storage, lowering the overall throughput. Also, more data must be transferred across the network during read and write operations as the dataset size grows. The system's performance may be impacted by this increasing network traffic, particu-

larly if the network bandwidth becomes a bottleneck. Throughput may be decreased as a result of network contention and increased latencies.

The observation is inline with [22].

4.2.1 Novel Findings

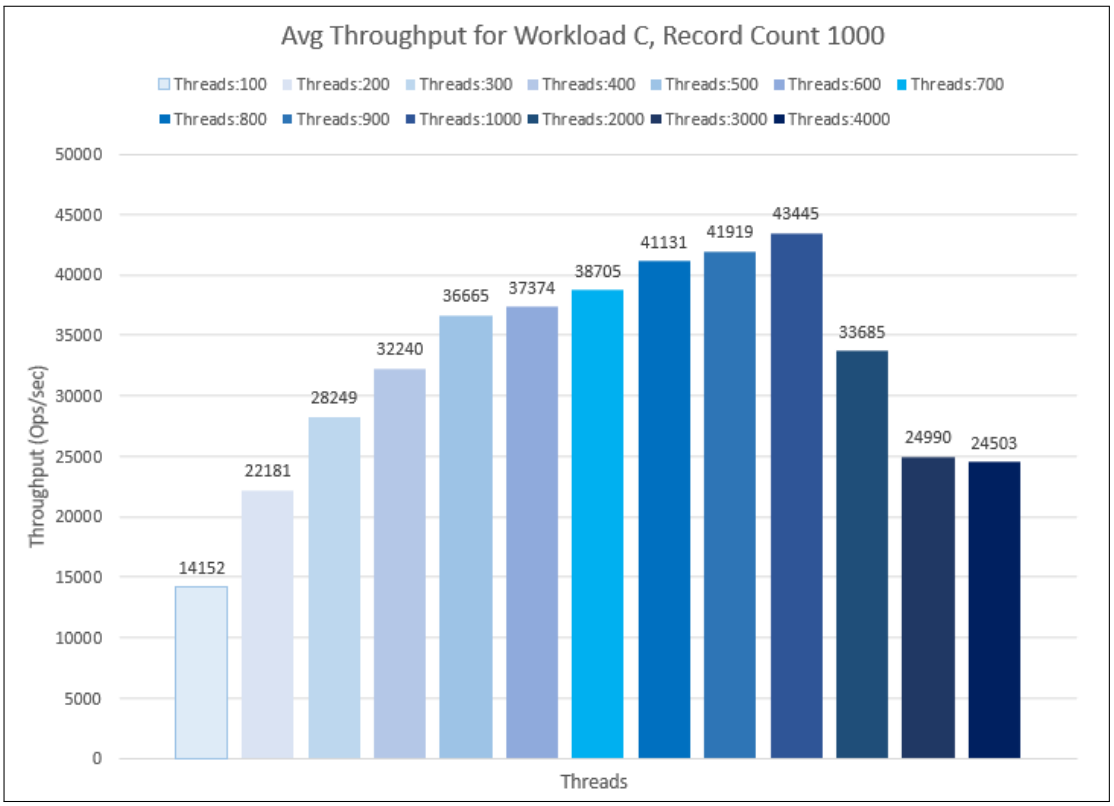


Figure 4.8: Threads vs Throughput

1. Observation:
   From Fig. 4.8 we can notice that for a given cluster configuration, if the number of threads is increased without varying any other parameters, the throughput increases logarithmically up to a certain limit and then decreases.

   Reason:
   This is because with more threads, more concurrent read/write requests can be sent to the Cassandra cluster, thus improving the number of read/write operations performed per unit of time. However, increasing the number of threads beyond a certain point can decrease throughput because it leads to

an increase in contention for resources such as CPU and memory, thereby reducing the system's efficiency in processing individual requests.

The observation of the initial increase in the throughput is inline with [14] and [20]. However, no previous works have considered the falling trend of the throughput with increase in thread count after reaching some threshold.

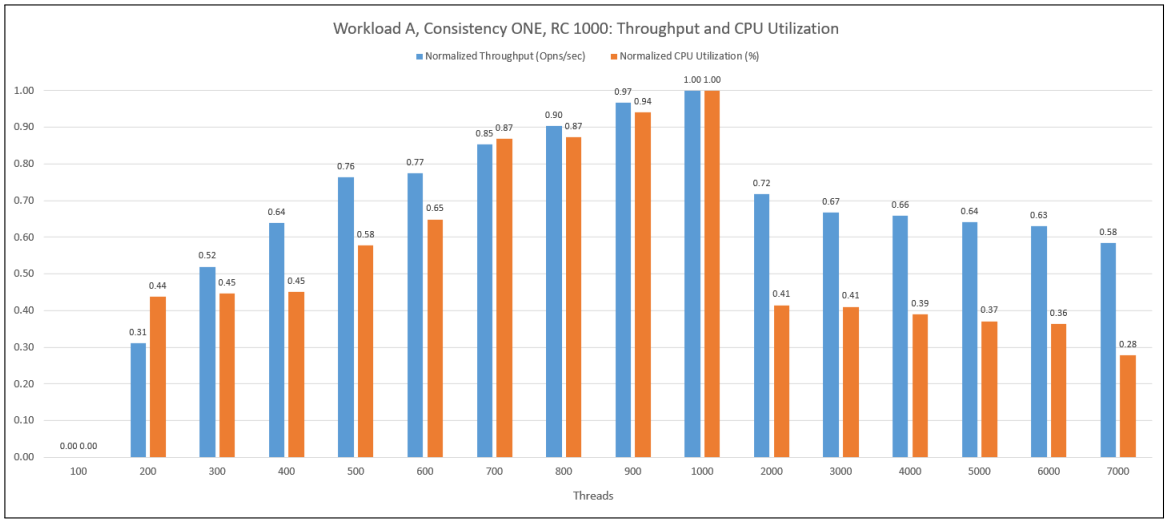


Figure 4.9: Threads vs CPU Utilization

2. Observation:
   From Fig. 4.9 we can notice that for a given cluster configuration, if the number of threads is increased without varying any other parameters, then the CPU utilization increases up to a certain limit and then decreases.

   Reason:
   The behaviour of CPU utilization increasing and then decreasing with an increase in the number of threads can be explained by the behaviour of CPU utilization during parallel processing. When a workload is executed by a single thread, the CPU utilization is likely to be low. However, as the number of threads increases, the CPU utilization also increases as the CPU is tasked with managing multiple threads in parallel. At some point, adding more threads will not result in an increase in CPU utilization as the CPU becomes saturated and the number of available processing cycles per thread decreases. In such cases, adding more threads can actually result in a decrease in CPU utilization, as the CPU spends more time switching between threads and less time executing actual processing tasks.

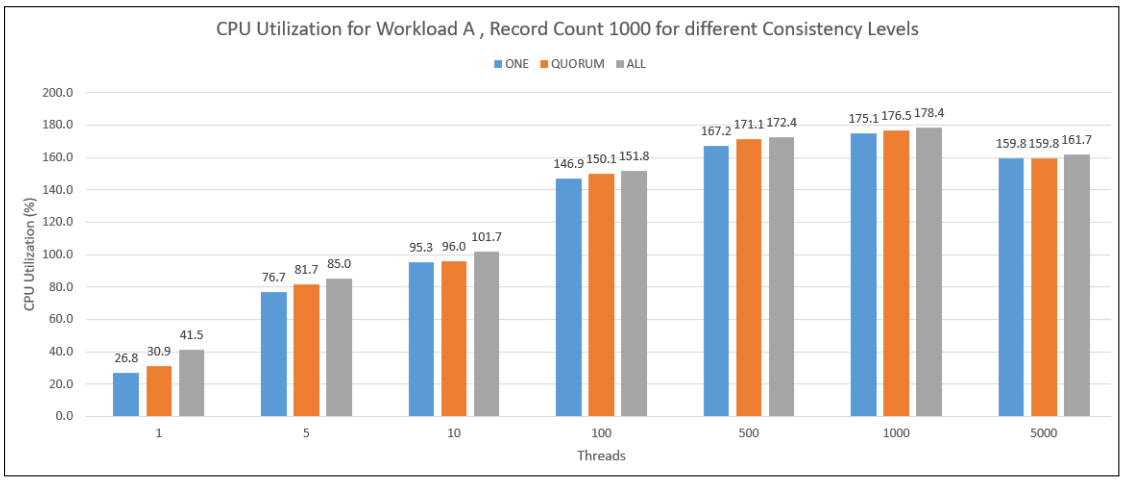


Figure 4.10: Consistency vs CPU Utilization

3. Observation:
   From Fig. 4.10 we can notice that for a given cluster configuration, if the consistency level is strengthened without varying any other parameters, then the CPU utilization increases for majority of the experiments.

   Reason:
   It is generally expected that a weaker consistency level would have lower CPU utilization compared to a stronger consistency level. This is because the former requires lesser replica nodes to respond to a request than the latter. Therefore, the coordination and communication overhead among replica nodes is reduced, resulting in lower CPU utilization.

4. Observation:
   From Fig. 4.11 we can notice that for a given cluster configuration, if the dataset size is increased without varying any other parameters, then the read/update latency increases.

   Reason:
   As the dataset size increases, the time required to retrieve records from Cassandra can increase, which can result in higher latency for operations Additionally, as the dataset size grows, more data needs to be transferred over the network to the client, which can also increase latency.

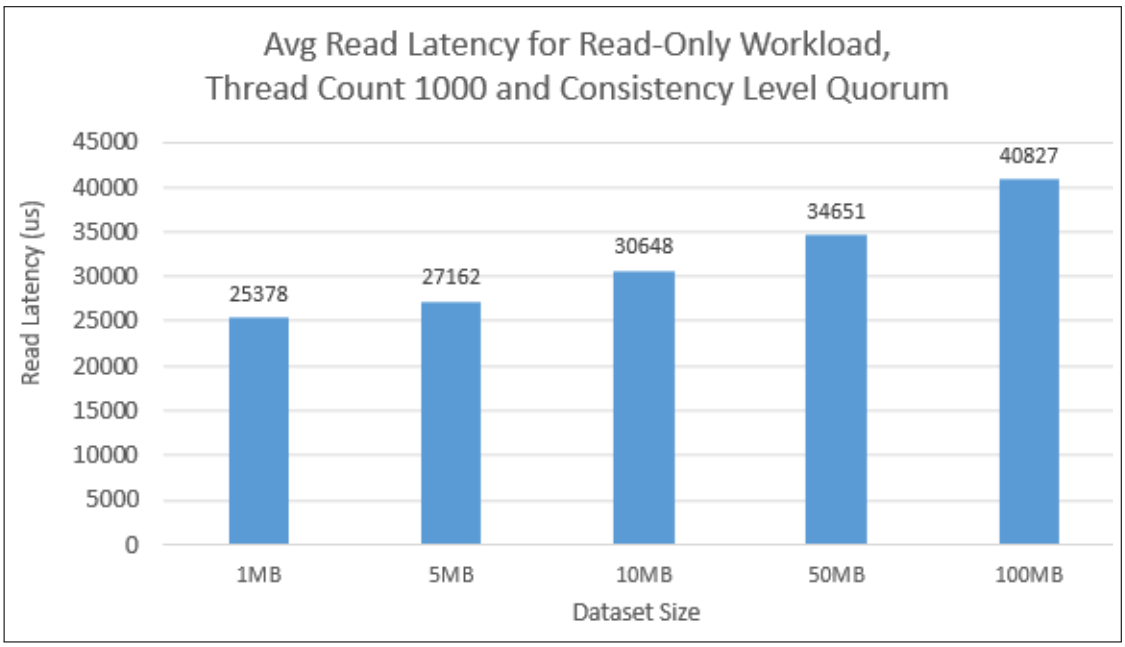


Figure 4.11: Dataset Size vs Latency

4.3 Summary of Experimental Results

Table 4.1 summarizes the effect of the YCSB core properties on the performance metrics.

Property	Variation	Throughput	Latency	CPU Util.
Thread Count	Increased {1, 2, . . . , 10, 100, 200,. . . , 1000, 2000, . . . , 8000}	Logarithmically increases and then decreases	Linearly increases	Increases and then decreases
Record Count	Increased {1k, 5k, 10k, 50k, 100k}	Decreases	Increases	NA
Consistency Level	Strengthened {ONE, QUORUM, ALL}	Decreases	Increases	Increases
Operation Count	Increased {1k, 2k, . . . , 10k}	Increases	Increases	NA
Dataset Size	Increased {1MB, 5MB, 10MB, 50MB, 100MB}	Decreases	Increases	NA

Table 4.1: YCSB Core Properties and effect on Performance Metrics

CHAPTER 5
Comparative Performance Analysis of Cassandra and HBase

The performance benchmarking of Cassandra and HBase was done using YCSB workload A (Update-heavy workload) and workload B (Read-heavy workload). The experiments were performed on a 3-node cluster with record count= 10^6 and operation count= 10^5. To identify the effect on latency and throughput, various thread counts were used for executing the workloads. For each thread count, different values of target throughput were specified in order to identify the threshold throughput achievable by that thread count.

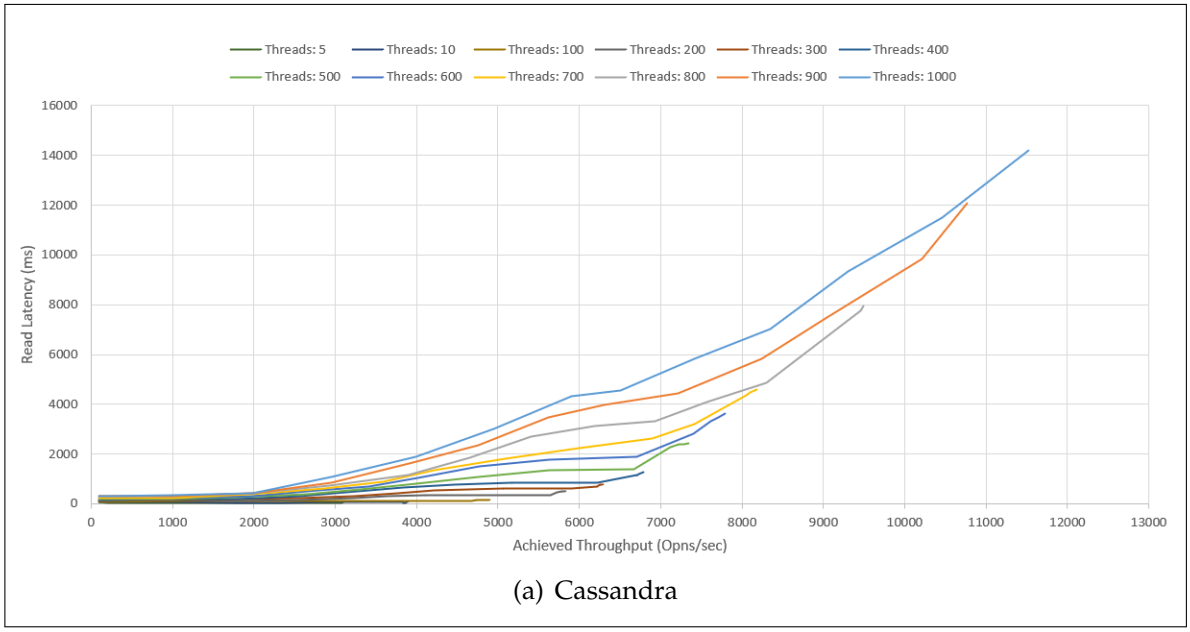
5.1 Experimental Setup

The specifications of the experimental setup are as follows:

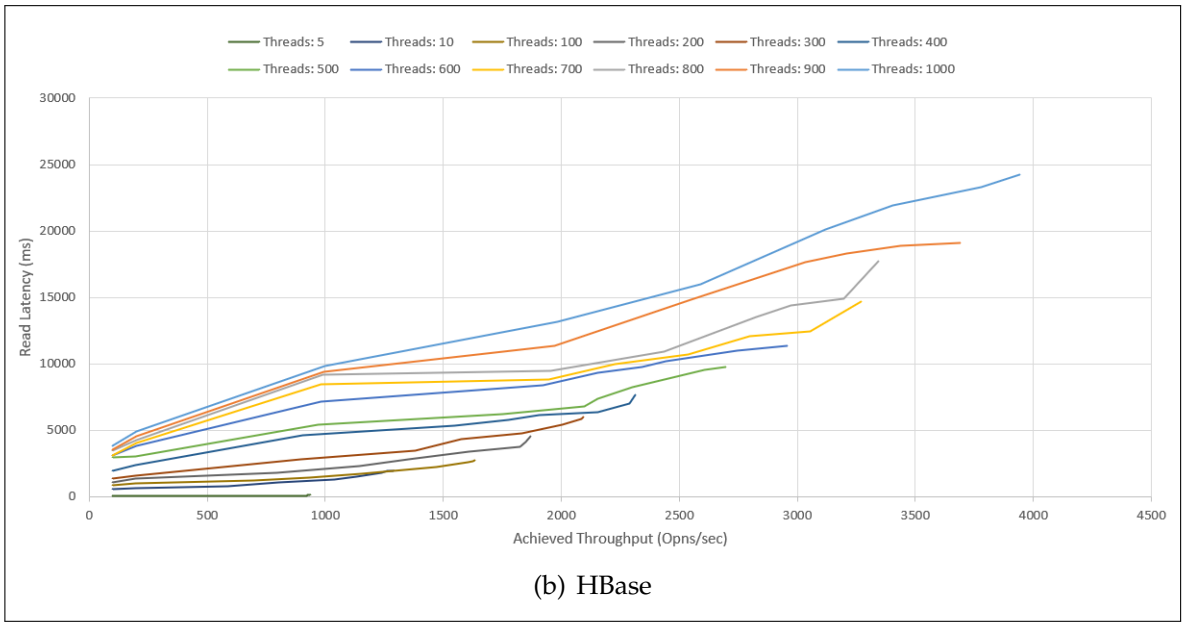
- YCSB Version: 0.17.0
- Cassandra Version: 3.11.13
- HBase Version: 2.4.16
- Hadoop Version: 2.7.2
- ZooKeeper Version: 3.4.10
- Operating System: Ubuntu 18.04.3 - 64 bit
- Memory: 12 GB
- Hardware Model: Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz

We make the assumption that benchmarking is carried out on specifically designated resources and that the nodes are not performing any compute-intensive operations simultaneously at the time of our testing.

5.2 Analysis for Update-Heavy workload (YCSB Workload A)

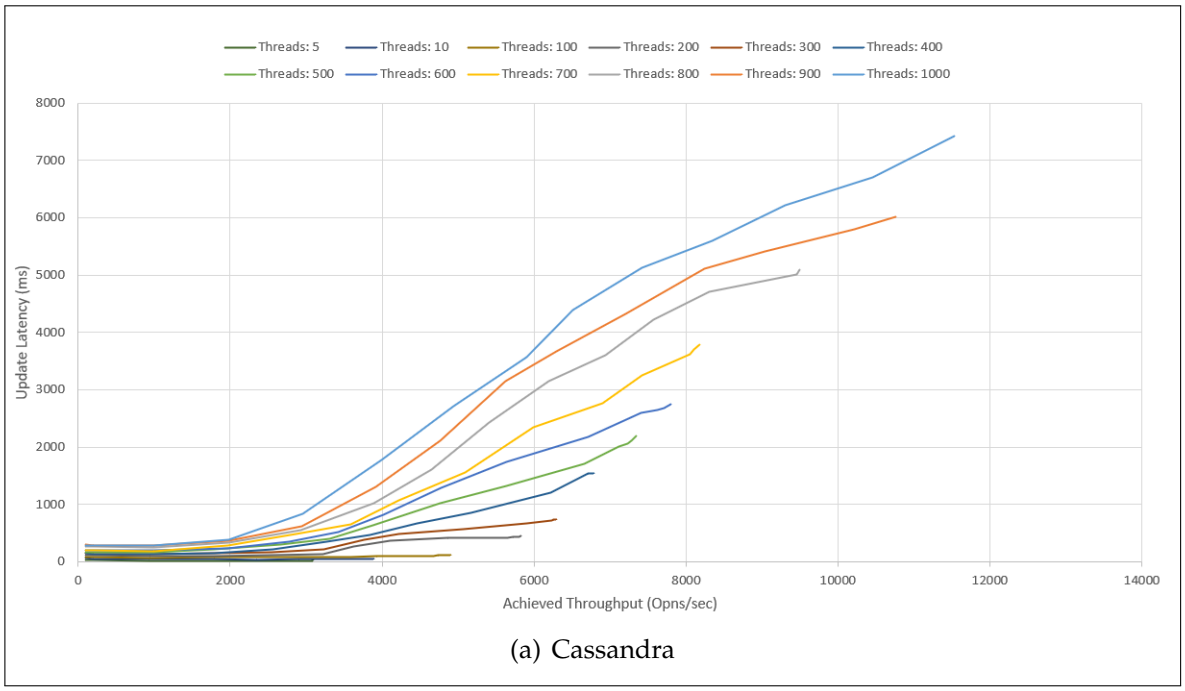


(a) Cassandra

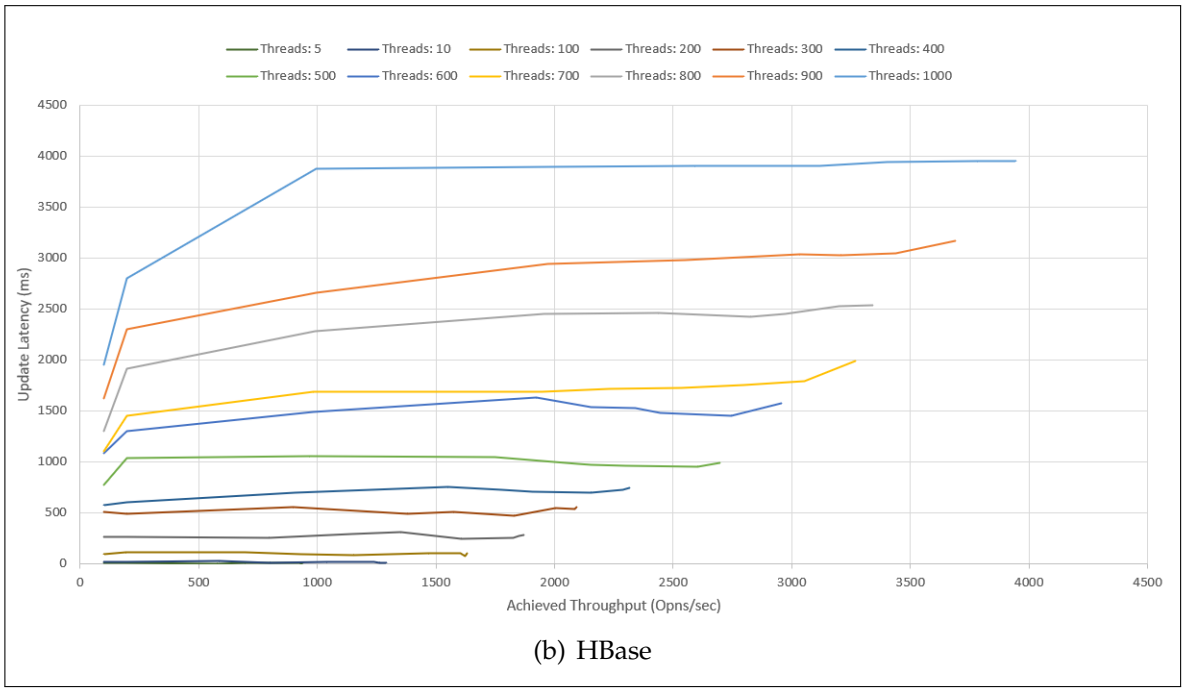


(b) HBase

Figure 5.1: Read Latency vs Achieved Throughput for Workload A

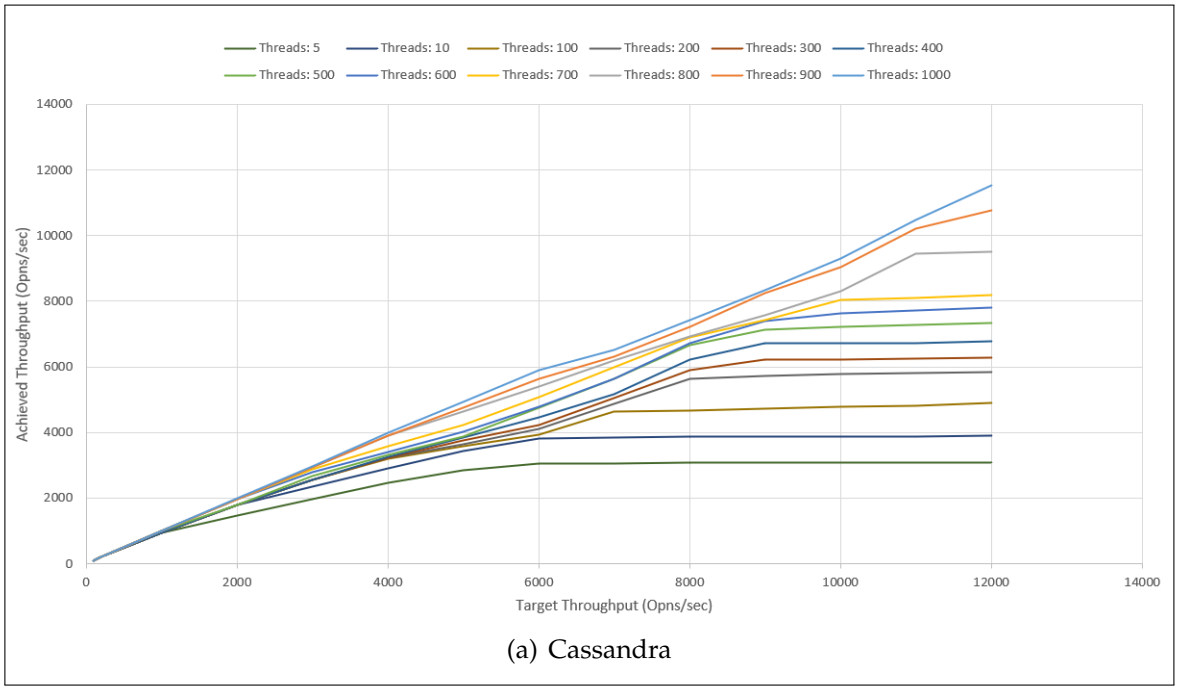


(a) Cassandra

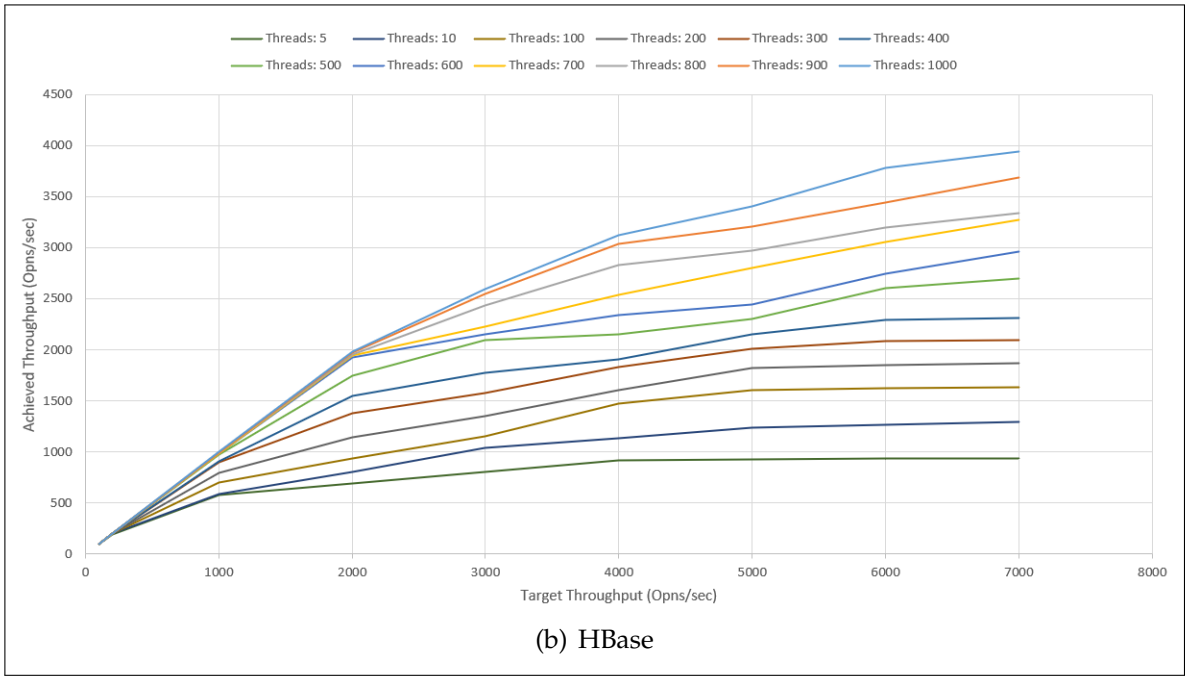


(b) HBase

Figure 5.2: Update Latency vs Achieved Throughput for Workload A



(a) Cassandra



(b) HBase

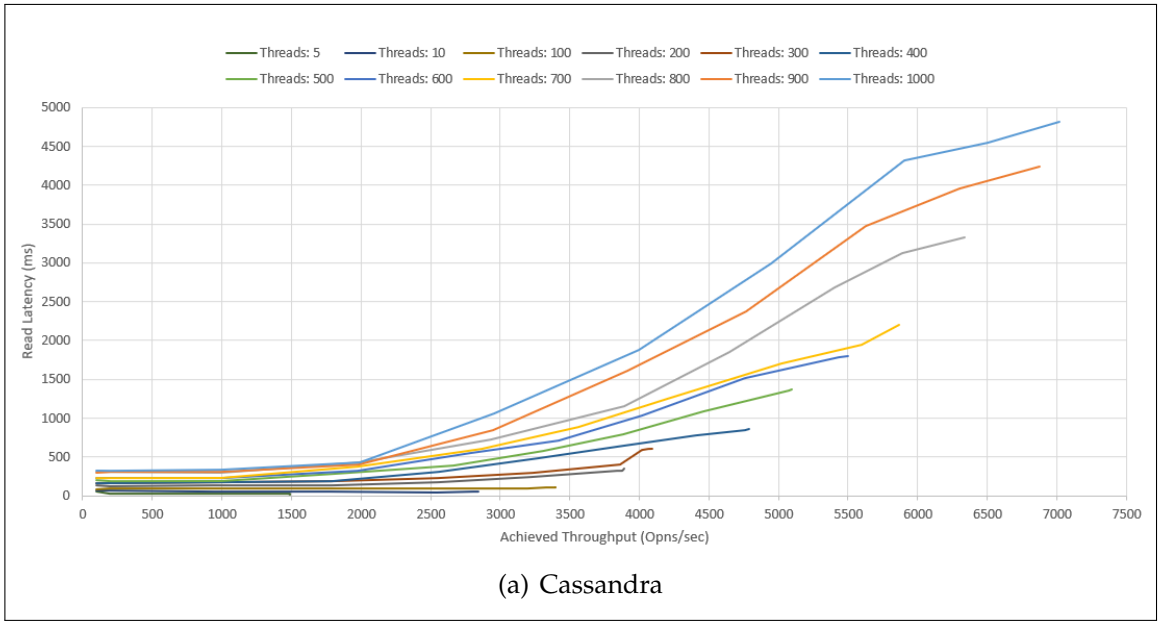
Figure 5.3: Achieved Throughput vs Target Throughput for Workload A

5.3 Results for Update-Heavy Workload

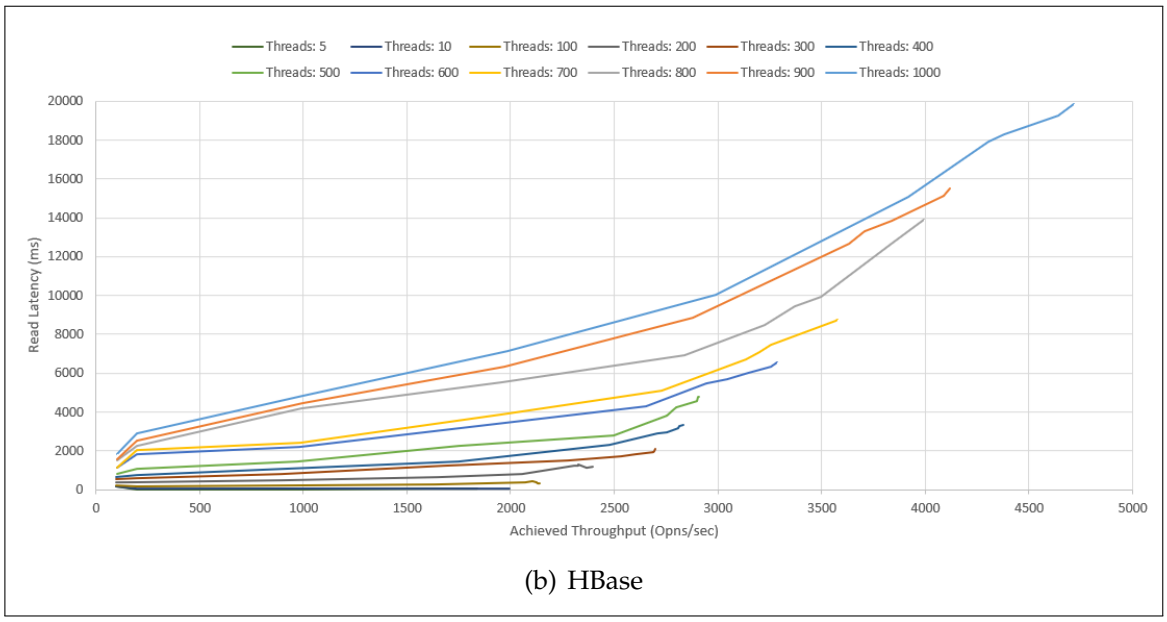
From the experiments performed for update-heavy workload, the following results can be concluded:

- For both the column families, the read latency and update latency increase with an increase in the number of threads.
- The increase in read latency w.r.t. the achieved throughput in Cassandra is relatively sharp.
- The increase in read latency w.r.t. the achieved throughput in HBase is gradual.
- From Fig. 5.1, we can see that Cassandra has a low average read latency compared to HBase. For a given value of thread count and achieved throughput, the read latency of Cassandra is approximately 0.4 times the read latency of HBase.
- The increase in update latency w.r.t. the achieved throughput in Cassandra is relatively sharp.
- The increase in update latency w.r.t. the achieved throughput in HBase is almost constant.
- From Fig. 5.2, we can see that Cassandra has a high average update latency compared to HBase. For a given value of thread count and achieved throughput, the update latency of Cassandra is approximately 1.85 times the update latency of HBase.
- The achieved throughput increases with an increase in the number of threads.
- Each thread is associated with a threshold after which the achieved throughput remains constant, no matter how much the target throughput is increased.
- From Fig. 5.3, we can see that the throughput of Cassandra is nearly 2.8 times the throughput of HBase.

5.4 Analysis for Read-Heavy workload (YCSB Workload B)

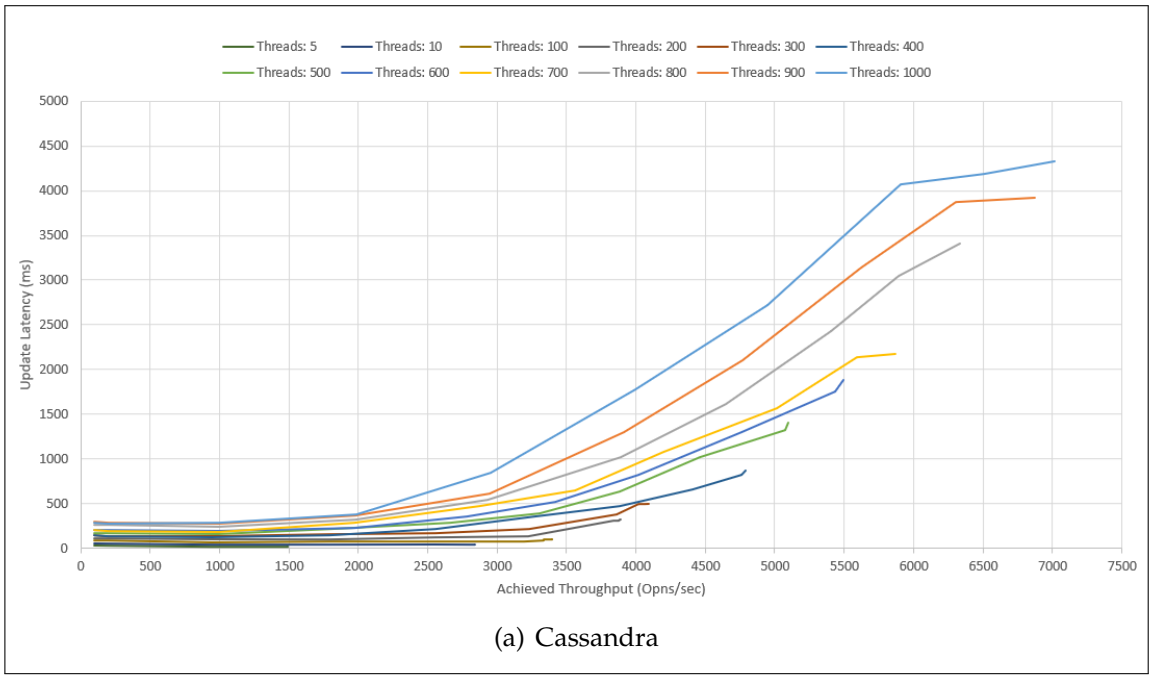


(a) Cassandra

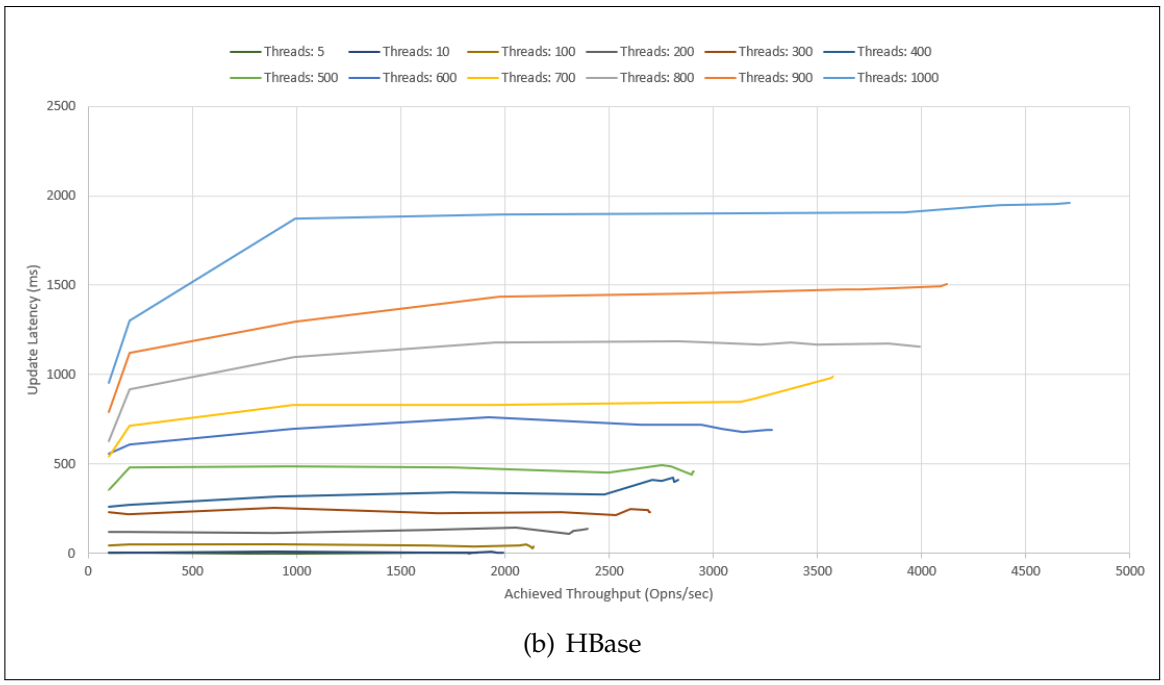


(b) HBase

Figure 5.4: Read Latency vs Achieved Throughput for Workload B

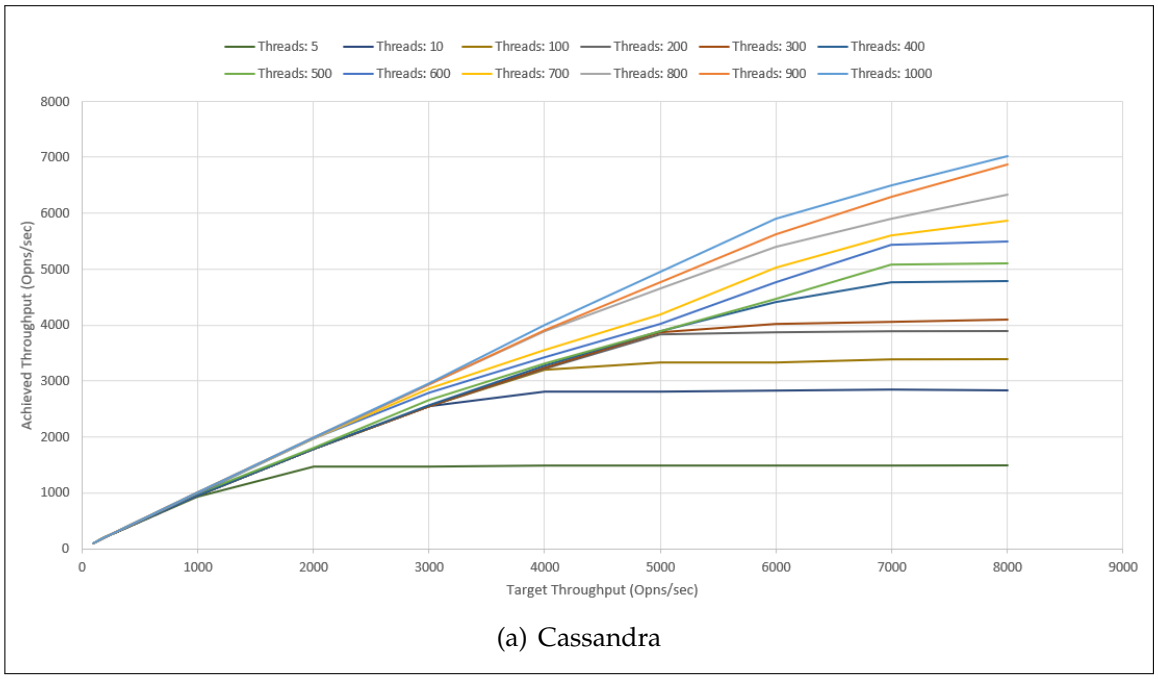


(a) Cassandra

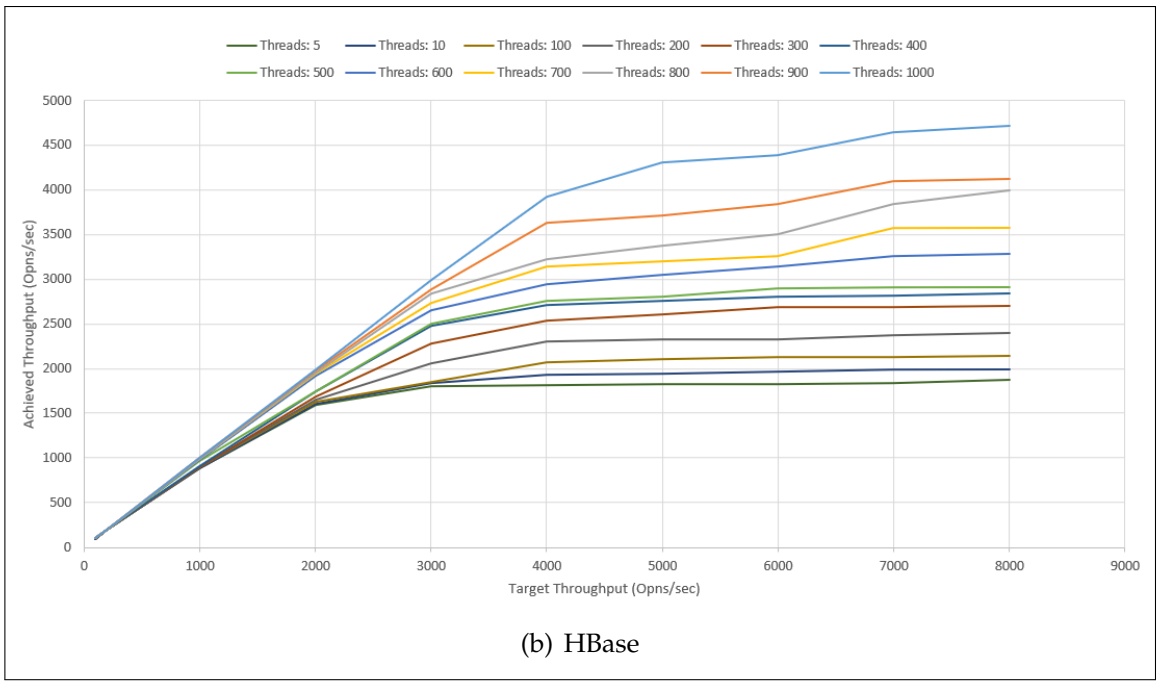


(b) HBase

Figure 5.5: Update Latency vs Achieved Throughput for Workload B



(a) Cassandra



(b) HBase

Figure 5.6: Achieved Throughput vs Target Throughput for Workload B

5.5 Results for Read-Heavy Workload

From the experiments performed for read-heavy workload, the following results can be concluded:

- For both the column families, the read latency and update latency increase with an increase in the number of threads.
- The increase in read latency w.r.t. the achieved throughput in Cassandra is relatively sharp.
- The increase in read latency w.r.t. the achieved throughput in HBase is gradual.
- From Fig. 5.4, we can see that Cassandra has a low average read latency compared to HBase. For a given value of thread count and achieved throughput, the read latency of Cassandra is approximately 0.25 times the read latency of HBase.
- The increase in update latency w.r.t. the achieved throughput in Cassandra is relatively sharp.
- The increase in update latency w.r.t. the achieved throughput in HBase is almost constant.
- From Fig. 5.5, we can see that Cassandra has a high average update latency compared to HBase. For a given value of thread count and achieved throughput, the update latency of Cassandra is approximately 2.5 times the update latency of HBase.
- The achieved throughput increases with an increase in the number of threads.
- Each thread is associated with a threshold after which the achieved throughput remains constant, no matter how much the target throughput is increased.
- From Fig. 5.6, we can see that the throughput of Cassandra is nearly 1.6 times the throughput of HBase.

5.6 Result Discussion

5.6.1 Read Latency: Cassandra<HBase

Our results align with the findings reported in past work [10].

- When Memstore data is flushed to disk in HBase, it is written to different files referred to as HFiles. The data in these files is represented as a sorted list of key-value pairs that refer to the data present in the Memstore. Nevertheless, several copies of the same data may be saved across various HFiles since HBase employs a log-structured storage system. Increased read latencies may result from the necessity for HBase to look through several HFiles to locate all relevant record fragments during read operations.
- HBase may also need to reconstruct record fragments from various disk pages. This is due to the fact that key-value pairs are sequentially stored to the disk in blocks known as data blocks when an HFile is written. Several key-value pairs may be present in these data blocks, and it is possible that several data blocks are required to hold all the elements of a single record. A read operation that has to retrieve a record might have to access numerous data blocks to get all the required fragments. Thereafter, it must assemble all those fragments and reconstruct the record.
- Cassandra, in contrast, employs a different storage mechanism that is enhanced for high read performance. A series of SSTables, each containing a sorted list of key-value pairs, are used by Cassandra to store data. Data is initially written to an in-memory structure called Memtable, which is periodically flushed to the disk as an SSTable. Since SSTables provide a sorted list of key-value pairs, Cassandra can execute read queries more quickly than HBase.

5.6.2 Update Latency: Cassandra>HBase

Our results align with the findings reported in past works [10] [23].

- HBase can be more effective for writes since it writes data to the disk in blocks. HBase's block-oriented storage strategy enables it to combine several writes into a single disk write operation, minimizing the number of disk I/O operations required during writes. Compared to Cassandra, which stores data as a series of SSTables, this can lead to lower write latencies.

- The write path of HBase is optimized for low-latency updates. When data is written to HBase, it is initially written to an in-memory structure Memstore, and then it is flushed to the disk. The Memstore is designed to handle a lot of updates and can be tuned as per the workload requirements. Contrarily, Cassandra stores updates in an append-only commit log before writing them to the Memtable, which might result in higher write latency.

5.6.3 Throughput: Cassandra>HBase

Our results align with the findings reported in past works [12] [22].

- We have inferred that the read latency of Cassandra is less than that of HBase. A read-intensive workload would comprise majorly of read operations. Since the read latency of Cassandra is less, it would be able to perform a greater number of database operations per unit time. Hence, the achieved throughput of Cassandra is greater than that of HBase.

- For the case of update-intensive workload, the number of read/update operations is equal. However, for both the database systems, the read latency is significantly greater in magnitude than the update latency. As a result, the throughput would be strongly impacted by the read latencies. Since Cassandra offers a lower read latency than HBase, the achieved throughput of Cassandra is greater.

CHAPTER 6
Conclusion and Future Work

The primary purpose of this thesis is to assess the performance of Cassandra under various configurations of system parameters/properties and to identify an ideal configuration that achieves desirable performance goals. Cassandra instances are benchmarked against YCSB to determine system performance. Throughput, latency and CPU utilization are computed using pre-configured YCSB workloads. The performance metrics are calculated for various thread counts, record counts, consistency levels, operation counts and dataset sizes.

Our findings align with previous research in the field, corroborating the established understanding of the relationship between various system parameters and the evaluation metrics. The observed patterns of the effect of thread count on latency; record count, consistency level, operation count on throughput and latency; dataset size on throughput are consistent with prior studies, providing further validation to the existing body of knowledge. The novel findings demonstrate that increasing the thread count initially improves the throughput and CPU utilization, but eventually leads to a decline after reaching a peak. Strengthening the consistency level increases the CPU utilization. Lastly, as the dataset size increases, the latency also increases. These findings emphasize the importance of considering these parameters and their implications when optimizing system performance.

The experiment may be expanded by incorporating other system parameters/ properties (such as request distribution, cluster size, and replication factor) into the experimental configurations. Later, an adaptation model may be used to find the optimal system design given certain constraints.

The comparative performance analysis of Cassandra and HBase is done for update-heavy and read-heavy workloads. For update-heavy workloads, we discovered

that the read latency, update latency, and throughput values for Cassandra are 0.4, 1.85, and 2.8 times the corresponding values for HBase. While for read-heavy workloads, the read latency, update latency, and throughput values for Cassandra are 0.25, 2.5, and 1.6 times the corresponding values for HBase. Hence, we infer that HBase is better suited for high update-intensive applications. In use cases where the number of reads is very high, Cassandra would be a better choice. While selecting a database for an application, it is important to keep in mind the data model, architectural implementation, query access path and performance trait of the candidate systems.

Benchmarking database performances in relation to the number of nodes can be done as a part of the future work. This will clarify on how the cluster size and system characteristics relate to one another. A specific database can also be compared to its earlier versions to evaluate if the present version performs better.

References

[1] Fully Distributed Mode HBase Cluster Setup. https://www.guru99.com/hbase-installation-guide.html. Accessed on May 29, 2023.

[2] Apache Cassandra Documentation. https://cassandra.apache.org/doc/latest/, 2008-2023. Accessed on May 29, 2023.

[3] Apache HBase Documentation. https://hbase.apache.org/, 2008-2023. Accessed on May 29, 2023.

[4] Multi-Node Cassandra Cluster Setup. https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/initialize/initSingleDS.html, 2008-2023. Accessed on May 29, 2023.

[5] S. Achari. *Hadoop Essentials - Tackling the Challenges of Big Data with Hadoop*. Packt Publishing, 2015.

[6] J. Carpenter and E. Hewitt. *Cassandra: The Definitive Guide*. O'Reilly Media, Inc., 2nd edition, 2016.

[7] B. F. Cooper, A. Silberstein, E. Tam, and R. Ramakrishnan. YCSB: Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB, 2009-2021. Accessed on May 29, 2023.

[8] B. F. Cooper, A. Silberstein, E. Tam, and R. Ramakrishnan. YCSB: Yahoo! Cloud Serving Benchmark- Core Properties. https://github.com/brianfrankcooper/YCSB/wiki/Core-Properties, 2009-2021. Accessed on May 29, 2023.

[9] B. F. Cooper, A. Silberstein, E. Tam, and R. Ramakrishnan. YCSB: Yahoo! Cloud Serving Benchmark- Core Workloads. https://github.com/brianfrankcooper/YCSB/blob/0.17.0/core/src/main/java/site/ycsb/workloads/CoreWorkload.java, 2009-2021. Accessed on May 29, 2023.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[11] X. Cui and W. Chen. Performance comparison test of hbase and cassandra based on ycsb. In *2021 IEEE/ACIS 19th International Conference on Computer and Information Science (ICIS)*, pages 70–77, 2021.

[12] V. Duarte, J. Bernardino, and P. Furtado. Experimental evaluation of nosql databases. *International Journal of Database Management Systems*, 6, 10 2014.

[13] L. George. *HBase: the Definitive Guide*. Aug 2011.

[14] A. Gorbenko, A. Romanovsky, and O. Tarasyuk. Interplaying cassandra nosql consistency and performance: A benchmarking approach. In S. Bernardi, V. Vittorini, F. Flammini, R. Nardone, S. Marrone, R. Adler, D. Schneider, P. Schleiß, N. Nostro, R. Løvenstein Olsen, A. Di Salle, and P. Masci, editors, *Dependable Computing - EDCC 2020 Workshops*, pages 168–184, Cham, 2020. Springer International Publishing.

[15] R. Hecht and S. Jablonski. Nosql evaluation: A use case oriented survey. In *2011 International Conference on Cloud and Service Computing*, pages 336–341, 2011.

[16] 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)*, pages 586–590, 2016.

[17] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, apr 2010.

[18] M. Noll. Multi-Node Hadoop Cluster Setup. https://www.michael-noll.com/tutorials/running-hadoop-on-ubuntu-linux-multi-node-cluster/. Accessed on May 29, 2023.

[19] P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st edition, 2012.

[20] M. Silva-Muñoz, A. Franzin, and H. Bersini. Automatic configuration of the cassandra database using irace. *PeerJ Computer Science*, 7, 2021.

[21] D. Sullivan. *NoSQL for Mere Mortals*. Addison-Wesley Professional, 1st edition, 2015.

[22] S. N. Swaminathan and R. Elmasri. Quantitative analysis of scalable nosql databases. In *2016 IEEE International Congress on Big Data (BigData Congress)*, pages 323–326, 2016.

[23] B. G. Tudorica and C. Bucur. A comparison between several nosql databases with comments and notes. In *2011 RoEduNet International Conference 10th Edition: Networking in Education and Research*, pages 1–5, 2011.

[24] H. Wang, J. Li, H. Zhang, and Y. Zhou. Benchmarking replication and consistency strategies in cloud serving databases: Hbase and cassandra. In J. Zhan, R. Han, and C. Weng, editors, *Big Data Benchmarks, Performance Optimization, and Emerging Hardware*, pages 71–82, Cham, 2014. Springer International Publishing.