Evaluation of Eventual Consistency and Linearizability in MongoDB

by

Vora Harshal Rajeshbhai
202111017

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

May, 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.

[illegible]

Vora Harshal Rajeshbhai

Certificate

This is to certify that the thesis work entitled Evaluation of Eventual Consistency in MongoDB has been carried out by VORA HARSHAL RAJESHBHAI 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.

[illegible]

Prof. P. M. Jat
Thesis Supervisor

Acknowledgments

I want to thank the people who have helped me during my research. Their support and guidance have been very important to me. I am incredibly grateful to my supervisor, Prof. P M Jat. He has always been there for me, providing support and valuable advice. Their expertise and guidance have shaped my research in a significant way.

I am also grateful to the faculty members of DA-IICT for their exceptional teaching, which laid the foundation for my knowledge and skills. Also, I want to express my deep gratitude to my family and friends. Their unwavering love, understanding, and encouragement have been my constant motivation and inspiration.

I would also like to thank my fellow researcher Mr. Kavan Hajare. He has been a great companion on this journey, engaging in exciting discussions and working together. I am grateful to the participants of my thesis. Their willingness to share their time and insights has been crucial to the progress of my research.

In conclusion, I am thankful to everyone mentioned above, as well as anyone else who has supported me along the way. Your contributions have been invaluable, and I am sincerely grateful for your presence in my academic and personal life.

Contents

Abstract

Sharding and replication are crucial techniques for scaling distributed systems, enabling data distribution across multiple nodes. However, as the number of replicas increases, maintaining consistency across them becomes increasingly challenging. Developers require understanding the consistency guarantees offered by different distributed systems to make informed decisions about the trade-offs between consistency and low latency. The study of consistency is essential to maintain data integrity and optimize performance and scaling.

In this work, we focus on evaluating the eventual consistency and linearizability provided by MongoDB, a popular distributed database system. The experiment considers various combinations of read and write concern levels and how they affect the consistency of the system. We also take into account different sizes of the document as a parameter in measuring the consistency. By analyzing these factors, we aim to quantify the impact of different parameters on the system's consistency.

We evaluate the performance of MongoDB by measuring the read and write latency of the operations by varying the read and write concern levels as well as by varying the document size. The evaluation of linearizability is based on measuring the occurrence of stale reads, that happen when a read operation accesses outdated or inconsistent data. By analyzing these variables, the aim is to provide a more comprehensive understanding of MongoDB's eventual consistency and linearizability and how it behaves in different scenarios.

Our findings reveal that using "linearizable" readConcern has a significant impact on the read latency and hence should only be used in scenarios where strong consistency guarantee is absolutely essential. Furthermore, document size has a significant impact on write latency and consistency but not on the read latency. Using "majority" readConcern and writeConcern provides a good balance between consistency and latency in MongoDB.

List of Figures

Chapter 1
Introduction

1.1 What is Consistency and Consistency in MongoDB

In NoSQL systems, maintaining consistency is crucial to ensure data correctness across distributed replicas. There are various consistency levels available in distributed databases. A consistency level is essentially a contract between processes and data store. It says that if processes agree to obey certain rules, the store promises to work correctly[7]. Distributed databases focus primarily on the following three consistency levels:-

1. Strong Consistency:- One of the consistency levels is Strong Consistency, which guarantees immediate and synchronized data updates. With strong consistency, when a write operation occurs, all subsequent read operations will only return the latest updated data. This level of consistency ensures that all replicas have the most up-to-date information, preserving data integrity and accuracy. Strong consistency is suitable for applications that prioritize consistency over availability.

2. Causal Consistency:- Causal Consistency guarantees that if one operation causally depends on another operation, the dependent operation will appear to have been executed after the causally preceding operation. In other words, suppose a client performs two operations, A and B, where B depends causally on A. Now, any subsequent read operation will see the effects of A before B.

3. Eventual Consistency:- Eventual consistency allows for temporary inconsistencies that eventually converge to a consistent state. In this consistency level, after a write operation is complete, the updates may propagate gradually to all replicas. This results in a period where different replicas might have different versions of the data. However, these inconsistencies are resolved over time, and all replicas eventually reach the same state. Eventual

consistency provides higher availability and improved performance than strong consistency, making it suitable for applications that can tolerate temporary inconsistencies and prioritize availability and scalability.

MongoDB is a popular open-source document-oriented NoSQL database system. MongoDB provides high scalability, performance, and flexibility [1]. MongoDB supports the above three consistency levels, and it allows for fine-tuning of consistency levels during read and write operations ranging from strong consistency to eventual consistency with the help of readConcern and writeConcern settings. readConcern and writeConcern in MongoDB can be explained as follows:-

- readConcern:- It allows you to control the consistency and isolation properties of the data read from the replica set [1].
- writeConcern:- It allows you to set the level of acknowledgement that is desired for a write operation [1].

1.2 Motivation

Analysis of Consistency offered by various distributed systems is an important task as it helps the developers to understand whether it is worth sacrificing consistency for low latency.

Performance and scalability are critical considerations in distributed systems. Consistency levels directly impact these factors. Strong consistency level often introduces a significant overhead and may limit scalability, while eventual consistency level offers greater scalability at the cost of temporary inconsistencies. By exploring the different consistency models offered by distributed systems, developers can determine the appropriate level of consistency needed to maintain data integrity and correctness.

Analyzing consistency in distributed databases is vital for maintaining data integrity, optimizing system performance, and understanding the behaviour of the system under varying conditions. By thoroughly examining these factors, developers can make well-informed decisions regarding the choice of a distributed system and consistency model that best aligns with the needs of their application.

1.3 Problem Statement

We focus on experiments to test the consistency of MongoDB under various consistency settings. The objective of the experiment is to measure the read latency, write latency and the number of stale reads during concurrent read and write operations. We attempt finding the answer to the following questions:-

- How do various factors such as readConcern, writeConcern and document size affect the consistency of the database during concurrent read and write operations?
- Does MongoDB provide the linearizability that it guarantees? And if it does, then at what cost?

1.4 Our Contribution

We conduct experiments to analyze the consistency of the MongoDB database under various conditions, that are operation-level consistency,session-level consistency and consistency in the presence of multiple readers and writers.

1. Understanding the impact of readConcern and writeConcern: Through our experiment, we observe how different factors such as readConcern and writeConcern affect the consistency and performance of the database.
2. Understanding the impact of Document size: We also observe the effect that various document sizes have on the consistency and performance of the database.
3. Evaluation of Linearizability in MongoDB: One of the important contributions of our research is the evaluation of Linearizability. We examined to which extent MongoDB achieves Linearizability and what are the associated costs or trade-offs.

1.5 Outline

Chapter 2 discusses Eventual Consistency, Linearizability in MongoDB, and Consistency support in MongoDB. Chapter 3 discusses Related Works, and Chapter 4 discusses the Experiment setup, details and Results.

Chapter 2
Background

2.1 Eventual Consistency

In distributed databases like MongoDB, eventual consistency is a core principle that prioritizes availability and scalability in large-scale systems. Traditional relational databases emphasize strong consistency while eventual consistency acknowledges the challenges of distributed systems and relaxes the immediate consistency requirements.

CAP theorem [2] states that out of the three i.e. Consistency, Availability and Partition, only two can be guaranteed at a particular moment in time. In a distributed database, we already have partitions, and hence we have to make sacrifice on either consistency or availability [3]. In an eventually consistent system, replicas or nodes may temporarily hold inconsistent versions of data. This can occur due to delays in propagating updates across the network, which can be caused by factors like network partitions, latency, and concurrent updates happening simultaneously on different nodes. As a result, clients accessing different replicas may observe different values or stale values of the data.

However, the key aspect of eventual consistency is that given enough time and the absence of further updates, all replicas will eventually converge to a consistent state [8]. This convergence is achieved through asynchronous replication mechanisms that propagate updates across the system. Over time, the replicas synchronize and resolve any conflicting changes, ensuring they all reach a consistent state.

Eventual consistency allows distributed databases like MongoDB to provide high availability and scalability by tolerating temporary inconsistencies that eventually get resolved. This flexibility enables systems to handle network partitions, scale horizontally, and continue functioning even in the face of failures.

2.2 Linearizability in MongoDB

Linearizability is one of the highest achievable consistency standards in distributed systems, which states that each individual operation should appear to take place instantaneously or atomically. It should appear to take effect at some moment between its start time and complete time. To explain Linearizability in simple terms while executing multiple concurrent read and write operations, once a write is executed, all the later reads should provide us with the value of that write operation or the value of the most recently executed write operation. Similarly, once a read operation has returned a particular value, all the subsequent read operations should yield the same value or the value of a later write.

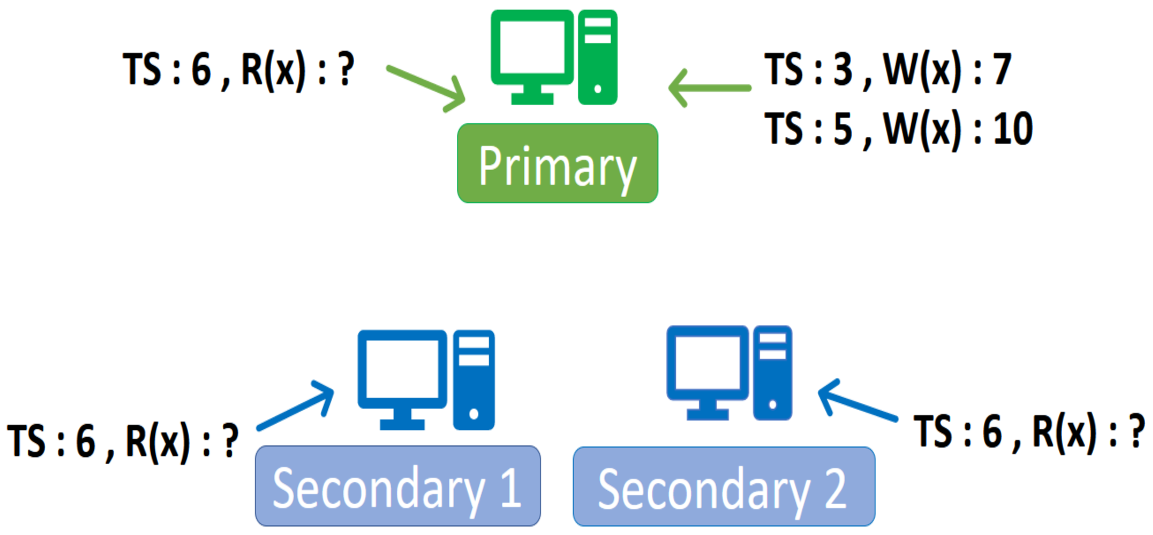

Figure 2.1: Linearizability for 3-node Replica Set

Here in Fig.2.1, we have a 3-node replica set which consists of one primary node and two secondary nodes. It is important to note that all the write operations in MongoDB go to the primary node, while the read operations can go to either primary or secondary nodes. We can see in Fig.2.1. that at TimeStamp TS: 3, a write operation occurs, which writes the value of x as W(x): 7 in the primary. Now, at TimeStamp TS: 5 another write operation occurs which writes the value of x as W(x): 10. Now, just after this write finishes a read operation is initiated in the replica set which can either go to the primary or any of the two secondaries. Linearizability states that irrespective of which member of the replica set returns the value of x, it should be the latest written value of x which is 10.

In the context of MongoDB, linearizability ensures that reads and writes to a specific document are seen in a strict sequential order, as if they happened one after the other in a linear fashion. When linearizability is achieved, a read operation always returns the most recent version of the document that has been acknowledged by a majority of the replica set members. This ensures that the read operation reflects the effects of all previous writes acknowledged by a majority, thereby providing strong consistency. Linearizability is the highest level of consistency offered by MongoDB and is particularly useful for scenarios where strict ordering and synchronization of reads and writes are critical, such as in financial systems or applications with complex data dependencies. So, in MongoDB as the write operations only go to the primary, the understanding of Linearizability can be reduced to the read operations not reading stale values of the data.

2.3 Consistency Settings in MongoDB

The ability to fine-tune consistency is an important feature in distributed databases like MongoDB, allowing developers to customize the level of consistency based on specific application requirements. In MongoDB, tunable consistency offers a flexible approach to balance consistency, availability, and performance trade-offs in distributed systems. This section explores the concept of tunable consistency in MongoDB and its significance in meeting varying application needs. The consistency levels in MongoDB replica sets are exposed to clients via readConcern and writeConcern levels, which are parameters of any read or write operation respectively [6].

Strong Consistency: Strong consistency in MongoDB ensures that every read operation from any replica or node in the distributed system returns the most recent and up-to-date value. MongoDB achieves strong consistency through synchronous replication, where write operations are not considered committed until they are acknowledged by a majority of replicas.

Eventual Consistency: Eventual consistency in MongoDB, as discussed earlier, relaxes immediate consistency guarantees in favour of availability and scalability. Updates made to the database are asynchronously propagated across replicas, and although temporary inconsistencies may occur, they eventually converge to a consistent state.

In MongoDB, we can vary the level of consistency from Strong Consistency to various levels of Eventual Consistency according to our needs with the help of read and write concerns.

2.3.1 writeConcern

MongoDB offers read and write concerns as tunable parameters to customize the consistency level. We can vary the consistency of operations from Strong consistency to various levels of Eventual consistency with the help of read and write concerns. writeConcern allows the developer to specify the consistency guarantees for the write operation.

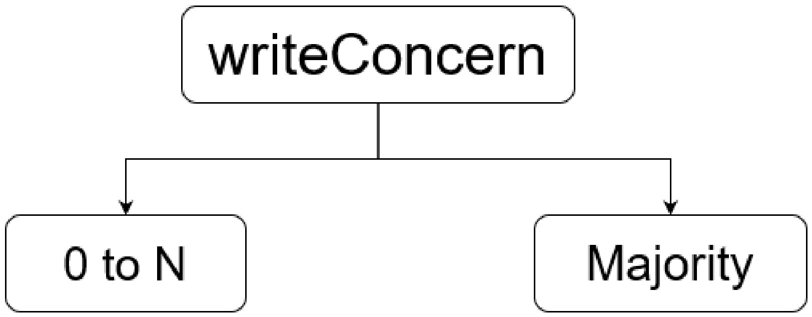

Figure 2.2: Parameter Options for writeConcern

As shown in Fig.2.2, we can specify the value of writeConcern from 0 to N where N is the number of nodes in our replica set or as "majority". Here are the different parameters of `writeConcern`: `w`. The `w` option specifies the number of nodes that must acknowledge the write operation before it is considered successful. It controls the level of acknowledgement and replication. The possible values for `w` are:

- w: 0 - No acknowledgement is requested from the server.
- w: 1 - The write operation waits for acknowledgement from the primary node (default value).
- w: <number> - The write operation waits for acknowledgement from the specified number of nodes.
- w: "majority" - The write operation waits for acknowledgement from the majority of nodes in the replica set or cluster.

A write operation executed with w: "majority" is enough to ensure the durability of the data in the replica set even in the event of failures.

2.3.2 readConcern

Similarly, readConcern allows developers to specify the consistency guarantees for read operations. As shown in Fig.2.3, we can set to values of readConcern as local, available, majority, snapshot or linearizable which can be explained as follows:-

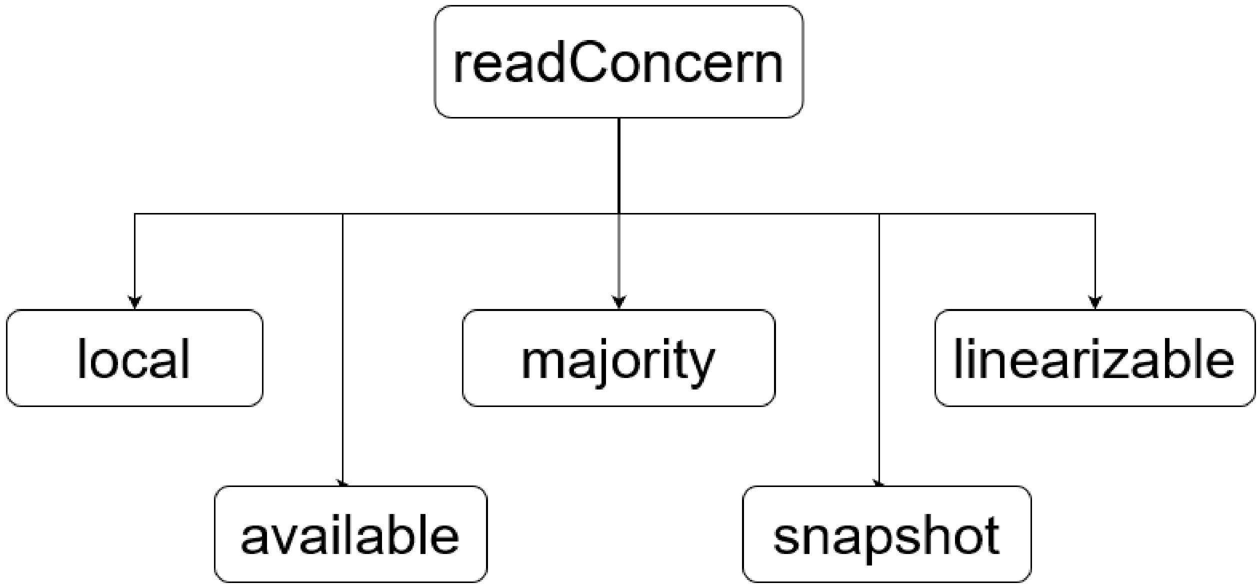

Figure 2.3: Parameter Options for readConcern

- r: "local" - It is the default read concern level.
- r: "available" - This level ensures that the read operation returns the most recent data available on a replica set or a sharded cluster. However, there is no guarantee of linearizability or consistency across multiple nodes.
- r: "majority" - This level provides strong consistency by ensuring that the read operation returns data that has been acknowledged by a majority of replica set members or shards. It guarantees that the data read is durable and reflects a point of consistency across the system.
- r: "snapshot" - The snapshot option is used to perform a read operation with snapshot isolation. It provides a consistent view of the data at a specific point in time.
- r: "linearizable" - This level provides linearizability, which guarantees that a read operation reflects the latest write operation on a global scale. It ensures

that reads never see stale or out-of-order data but may introduce additional latency.

MongoDB version 4.0 was introduced with readConcern: "linearizable", which helps achieve linearizability in MongoDB. By using the linearizable read concern, MongoDB ensures that the read operation appears as if it occurs atomically and instantaneously after all prior write operations have been completed.

It guarantees that the read operation returns the latest committed value, regardless of concurrent writes or network delays. By adjusting the read and write concerns, developers can fine-tune the consistency and performance balance according to the specific needs of their application.

2.3.3 Consideration and Tradeoffs

When leveraging tunable consistency in MongoDB, it is crucial to consider the trade-offs between consistency, availability, and performance. Using stronger read and write concern values provides data integrity but can introduce higher latencies and potential limitations in availability during network partitions or replica failures. On the other hand, weaker read and write concern values offers improved availability and scalability but may result in temporary data inconsistencies.

CHAPTER 3
Literature Survey

3.1 Checking Causal Consistency of MongoDB

The objective of the paper [5] is to propose a methodology for evaluating causal consistency, specifically in the context of MongoDB. The authors aim to address the research gap in evaluating causal consistency in MongoDB and provide insights into MongoDB's behaviour concerning this crucial consistency property. By developing a robust methodology, the authors seek to enable practitioners and researchers to assess and understand the causal consistency guarantees offered by MongoDB, contributing to the field of distributed systems and database consistency.

Their Experimental setup consisted of two shards, each of which is a replica set of 5 nodes. For each experiment, there were 100 registers and 10 clients and the generator generated read and write operations which were then appended into a queue. The ratio between the read and write operations is 3:1. They varied the total number of operations and the readConcern and writeConcern levels for operations. They performed the experiment for both the scenarios with and without nemesis, where nemesis is a tool used to simulate various network-related failures and anomalies. They checked if MongoDB satisfies all three variants of causal consistency(CC, CM and CCv).

Their results showed that in the presence of a nemesis, causal consistency can only be guaranteed for reads with readConcern : "majority" and writes with writeConcern : "majority".If the nemesis is not present, MongoDB can satisfy all the three variants of causal consistency which are CC,CM and CCv even with readConcern : "local" and writeConcern : "w1".

3.2 Tunable Consistency in MongoDB

The paper [6] aims to discuss and explore the concept of tunable consistency models in MongoDB's replication system. It highlights the utility of these models for application developers, explores the underlying mechanisms that enable tunable consistency, presents case studies of real-world applications, characterizes the performance trade-offs, compares MongoDB's consistency offerings with other databases, discusses implementation details, and includes performance evaluations in the presence of failures. Overall, the paper provides a comprehensive understanding of tunable consistency in MongoDB and its implications for distributed database systems.

The experimental setup involved three experiments comparing write latency for different writeConcern values in MongoDB's replication system. The experiments were conducted on 3-node replica sets with varying geographical distributions of replica set members. The results showed the impact of writeConcern on latency in local, Cross-Availability Zone, and Cross-Region scenarios. The experiments were performed using MongoDB versions 4.0.2 and 4.0.3, with SSL enabled or disabled, and deployment is done using MongoDB Atlas. The results provided insights into the trade-off between durability guarantees and latency in different deployment configurations.

The experiment showed the latency comparison for different geographical locations and it was found that the latency is lowest for local setup where the replica set and the client were in the same AWS while it increased by 53% on average for writeConcern : "local". And it increased by 44% for writeConcern : "majority" when the replica set members were in different Availability Zones. Furthermore, the latency increased by 5746.4% for writeConcern : "w1" and it increased by 6220.75% for writeConcern : "majority" as compared to local setup when the replica set members were in different region.

3.3 Data Consistency Properties of Document Store as a Service (DSaaS)

The paper [4] aimed to conduct an empirical study to quantify the inconsistency observed in data held in MongoDB Atlas, a hosted offering of MongoDB as a Service. The study focuses on understanding the level of data inconsistency in MongoDB Atlas, considering its characteristics as a document store and the trade-off between consistency and low latency/high availability during partitions. The paper seeks to benchmark the consistency level of MongoDB Atlas, specifically by measuring the probability that a read operation encounters stale values.

The MongoDB Atlas cluster is hosted on AWS in the Sydney region and they used dedicated M10 clusters whose configurations were 2GB RAM, 10GB Storage and 0.2vCPU. The application is run on EC2 instance of AWS.The chosen EC2 instance type is c4.2xlarge with Ubuntu Server 16.04 LTS (HVM) as the operating system.The c4.2xlarge instance specifications include 32GB RAM, 8GB storage, and 8 vCPUs.

The evaluation results showed that there were zero inconsistencies when reading from the primary as it is expected but there were some inconsistencies when the read happened from secondary copy and nearest copy. When the MongoDB Atlas cluster and the evaluating application were in the same region, they observed that there were inconsistencies when reading from nearest copy as compared to primary copy and there is no significant difference in latency. So, they concluded that the sacrifice of consistency did not make any sense if the application and the cluster are in the same region.

3.4 Summary of Literature Survey

Name of paper	Methodology	Results
Checking Causal Consistency of MongoDB	The paper[5] proposes a methodology to evaluate causal consistency in MongoDB. By conducting experiments with two shards, each consisting of a replica set of 5 nodes, they assess MongoDB's behaviour concerning causal consistency guarantees. The experiments include 100 registers, 10 clients, and a mix of read and write operations.	The results indicate that MongoDB can achieve all three variants of causal consistency (CC, CM, and CCv) without any issues when nemesis (a tool simulating network failures) is absent. However, in the presence of nemesis, MongoDB can only guarantee causal consistency for reads with readConcern: "majority" and writes with writeConcern: "majority."
Tunable Consistency in MongoDB	The paper[6] discusses the concept of tunable consistency models in MongoDB's replication system. It includes comparisons with other databases, implementation details, and performance evaluations in the presence of failures. The experiments involved three scenarios comparing write latency for different writeConcern values in 3-node replica sets with varying geographical distributions.	The results revealed the impact on latency in local, Cross-Availability Zone, and Cross-Region setups, highlighting the trade-off between durability guarantees and latency in different configurations. The latency was lowest for local setup while it increased for Cross-Availability zone setup and it was highest for Cross-Region setup.
Data Consistency Properties of Document Store as a Service (DSaaS)	The paper[4] presents a study aiming to quantify data inconsistency in MongoDB Atlas. It focuses on understanding the trade-off between consistency and low latency/high availability during partitions. The study benchmarks the consistency level by measuring the probability of encountering stale values during read operations.	The results revealed that when reading from the primary replica, there were no inconsistencies. However, inconsistencies were observed when reading from secondary and nearest replicas. Notably, when the application and MongoDB Atlas cluster were in the same region, inconsistencies occurred.

CHAPTER 4
Experimentation and Results

In this section, we have mentioned the hardware specifications as well as the software specifications of the systems used for experimentation. The experiment is performed on a 3-node replica set in MongoDB.Also, we have discussed the results that were obtained from the experiment.

4.1 Experimentation Setup

Our setup consists of a 3-node replica set where each system is installed with Mongodb version 6.0.4 which is the latest at the time this experiment is performed. The individual system has Intel(R) Core i5-6700 processor with 4 cores. All the systems are installed with Ubuntu 16.04 as the operating system. Each system has 4GB of RAM with 500GB of storage space.The code for this experiment is written in java . The version of the jar file used for connecting to MongoDB is 3.12.13. And the code is executed in the Eclipse IDE with the version of Eclipse being 2023-03.

4.2 Experimentation Details

4.2.1 Architectural overview

As shown in Fig.4.1. There are three roles in our experiment - a writer, a reader, and a MongoDB local server setup. The writer continuously writes the timestamp into the database which is read by the reader simultaneously. The reader and writer are kept in different processes. Stale Reads for our experiment are defined as follows: The writer writes a document into the database which consists of some value of X which contains the current timestamp in it. After the document is written in the database the writer acknowledges the write. Now, when the reader reads some value of some version of X from the database. If the version of X that is read by the database is not the latest version of X in the database at that point in time, then it is considered as a stale read.

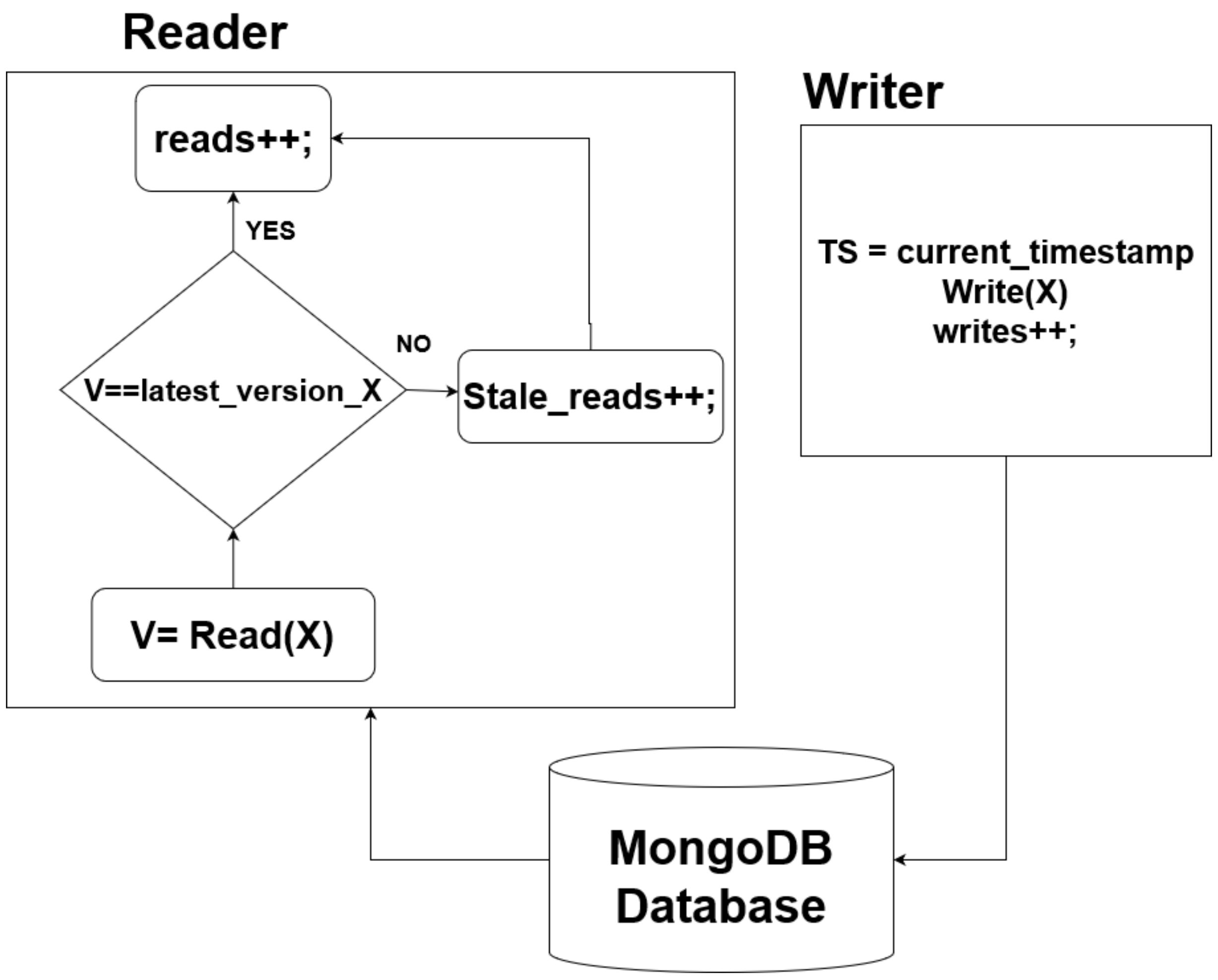

Figure 4.1: Architecture Diagram of Setup

4.2.2 Performance Metrics

In this experiment we analyze the consistency provided by MongoDB by varying the parameters discussed in section 4.2.3 . We have not changed the internal configurations of MongoDB. It is how any user would experience it in real life. We measure the following in our experiment:

1. Write Latency
2. Read Latency
3. Stale Reads

4.2.3 Varied Parameters

The parameters that are varied in the experiment are readConcern, writeConcern and the document size. Also we have taken three consistency settings into account which are operation-level consistency, session-level consistency, and consistency in presence of multiple readers and writers respectively. The different writeConcern values that we have used are:

1. Majority
2. Local

and the readConcern values that we have used are:

1. Linearizable
2. Majority
3. Local

We took every possible combination of these values and we have taken the average of 10 iterations of every configuration. We have also considered document size as one of the parameters and the document sizes that we have considered are 1 KB, 50 KB, 100 KB, and 1 MB. There are 6 possible combinations of readConcern and writeConcern values and we have run the experiment for 10 iterations of every combination of readConcern, writeConcern as well as the document size (readConcern, writeConcern, Document size). So, in total for each setup the experiment is run 240 times. And as there are 3 setups, so in total the experiment is run for 720 times. There are total 3 setups of our experiment which are as follows:

1. Operation-Level consistency: Our first experiment setup consisted of a Single reader and single writer for our MongoDB database. The writer continuously writes into the database with a time between two writes of one sec. Similarly, the reader continuously reads from the database where the time between two reads is one msec. In this setup, the writer writes a hundred documents in the database, with the time between the insertion of two documents being one second. And the reader continuously reads from the database until the writer is writing into the database. Both the writer and reader are implemented through multithreading in Java to simulate concurrency. We measure the write latency, the read latency, and the stale reads. We have used operation level consistency for this setup wherein we specify the writeConcern and readConcern values for each individual operation.

2. Session-Level consistency: Our second experiment setup consists of a Single writer and a single reader, with the writer writing into the database with the time difference between two writes being 1 second and the reader reading continuously from the database with the difference between two reads is one millisecond. In session-level consistency, we specify the writeConcern and readConcern values in the connection string itself rather than the write and read operation. For example, the normal connection string looks as follows:

   ```
   mongodb://localhost:27017/?replicaSet=rs1.
   ```

   Now, if we specify the writeConcern or readConcern in the connection string, it looks as follows:

   ```
   mongodb://localhost:27017/?replicaSet=rs1&w=<wc_level>&r=<rc_level>
   ```

   In this setup, same as the above setting, the writer inserts a hundred documents in the database, and the reader keeps reading from the database until the writer finishes insertion.

3. Multiple Reader-Writer: Our third experiment setup consists of multiple writers and readers performing simultaneous write and read operations into the database. Here, we have taken two writers who write into the database, with the difference between the two writes being one second and ten readers continuously reading from the database with the time difference being one millisecond. The two writers write a hundred documents into the database and all ten readers reads from the database until both writers finish execution. The purpose of having multiple writers and readers is to increase the concurrency of operations and the workload on the MongoDB server and the replica set.

4.3 Results

In this experiment, we tested the consistency of a distributed database system under different conditions. Specifically, we examined operation-level consistency, session-level consistency, and multiple reader-writer consistency. The goal of the experiment is to determine how different factors, such as write concern, read concern, and document size, affect the consistency of the database system. In all the figures we have taken the readConcern pair on the x-axis which shows the writeConcern of the write operation as well as the readConcern of the read operations that are happening in parallel. And on the y-axis we have specified the values which we are measuring which can be either write latency or read latency or percentage stale reads.

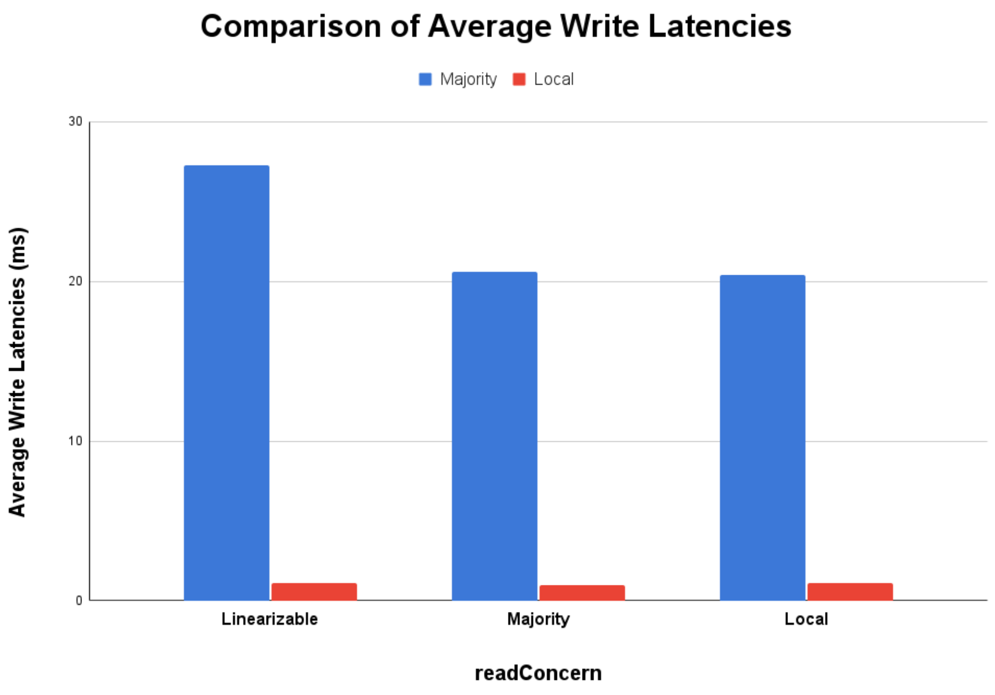

Figure 4.2: Write Latency with different writeConcern and readConcern settings

4.3.1 Anomaly in Write latency

In Fig.4.2 we can see that the Write Latency increases as we use stronger writeConcern values. Here, the write latency for writeConcern: "majority" is much higher than writeConcern: "local". Also, we have mentioned readConcern along with the writeConcern in the figure above because we can observe that the write latency for readConcern: "linearizable" is higher than its other two configurations. And similar pattern is observed even on increasing the document size. So, the question that arises is whether the readConcern of the read operations happening in parallel affects the write latency of the write operation even if both the writer and reader are in separate processes. And the answer we have found is that yes, it is possible that the readConcern: "linearizable" of a read operation can affect the write latency of another write happening simultaneously in MongoDB.

When a read operation with "linearizable" readConcern is executed, MongoDB ensures that the read operation returns the most recent committed data at the time the operation is initiated. This requires MongoDB to lock the database at a point in time, which means that no writes can be committed to the database after the lock is acquired until the read operation is completed. As a result, any concurrent write operations that happen during the time that the lock is held will be blocked until the read operation completes, which can lead to increased write latency. Additionally, MongoDB uses a technique called "commit quorum" to ensure the durability of write operations. When a write operation is executed with a writeConcern level of "majority", MongoDB requires that a majority of the replica set members acknowledge the write before it is considered committed. This means that if a read operation with "linearizable" readConcern is executing at the same time as a write operation with "majority" writeConcern, the write operation may have to wait for the read operation to complete before it can get the required number of acknowledgements from the replica set members, leading to increased write latency.

4.3.2 Operation-Level Consistency Setting

The figures from Fig.4.3 to Fig.4.9 are of the first setup of the experiment which is the operation-level consistency setup where the consistency settings are applied at the operation level. In Fig.4.3 and Fig.4.4 it can be observed that write latency tend to increase as the consistency level becomes stronger. This is expected as stronger consistency levels require more communication between nodes and can lead to increased latency. In the figures, we can also see that the write latency increases as the document size increases and the increase in write latency appears to be linear with respect to the increase in document size.For writeConcern:"local", we can see that there is a major spike in write latency when the document size is 1 MB and the reason for that could be that the larger the document size, the more time it would take to insert and hence causing the write latency.

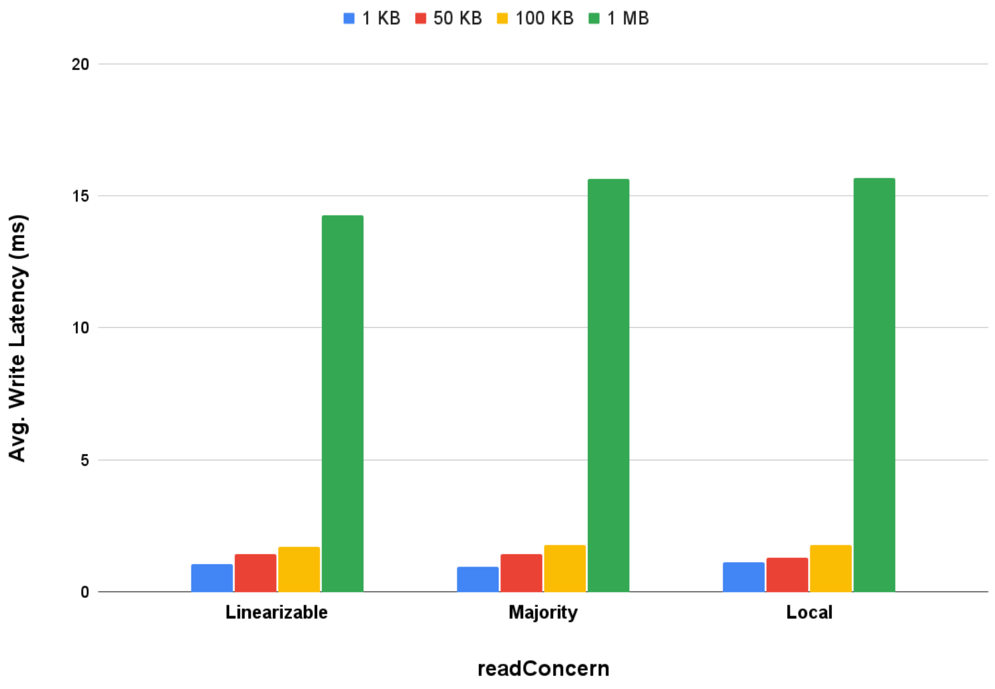

Figure 4.3: Comparison of Avg. Write Latencies for writeConcern: local

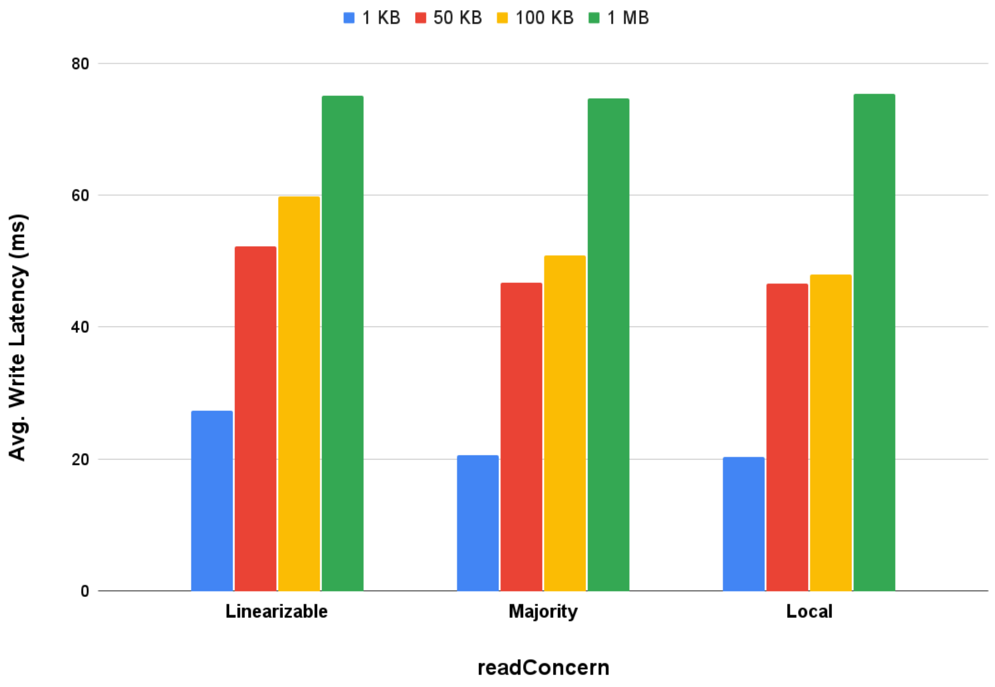

Figure 4.4: Comparison of Avg. Write Latencies for writeConcern: majority

In Fig.4.5, Fig.4.6 and Fig.4.7, we can see that read latency increases as the consistency level increases, similar to what is observed for write latency. Additionally, it can be seen that the readConcern level has a larger impact on read latency than the writeConcern level has on write latency. This is because read operations in MongoDB can be affected by the consistency level more significantly than write operations, as reads may need to wait for the latest data to propagate across the replica set in order to ensure linearizable reads. The document size has a minor impact on the read latency, which can be seen in the figures, as when the document size increases, the read latency does not increase significantly. now, in Fig.4.8 and Fig.4.9 we can observe that the stale reads decreases as the writeConcern and readConcern values become more strict. For example, when we use writeConcern:- Majority and readConcern:- Linearizable, there are almost no stale reads for any document size. This is because these settings ensure that the data is written and read from a majority of replica set members and in a linearizable order, respectively.

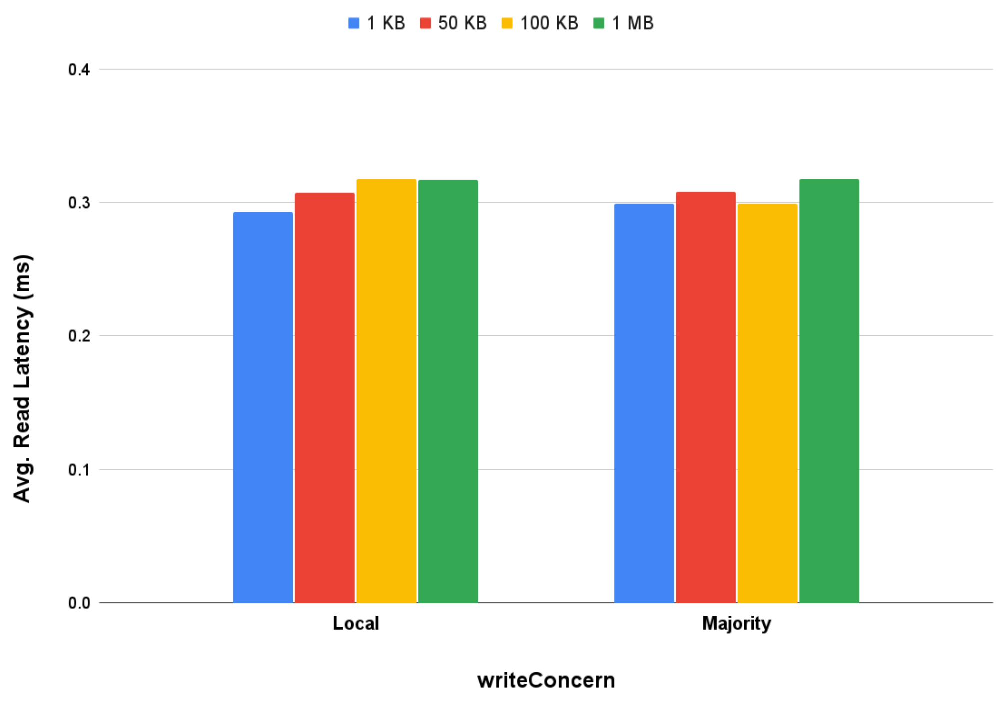

Figure 4.5: Comparison of Avg. Read Latencies for readConcern: local

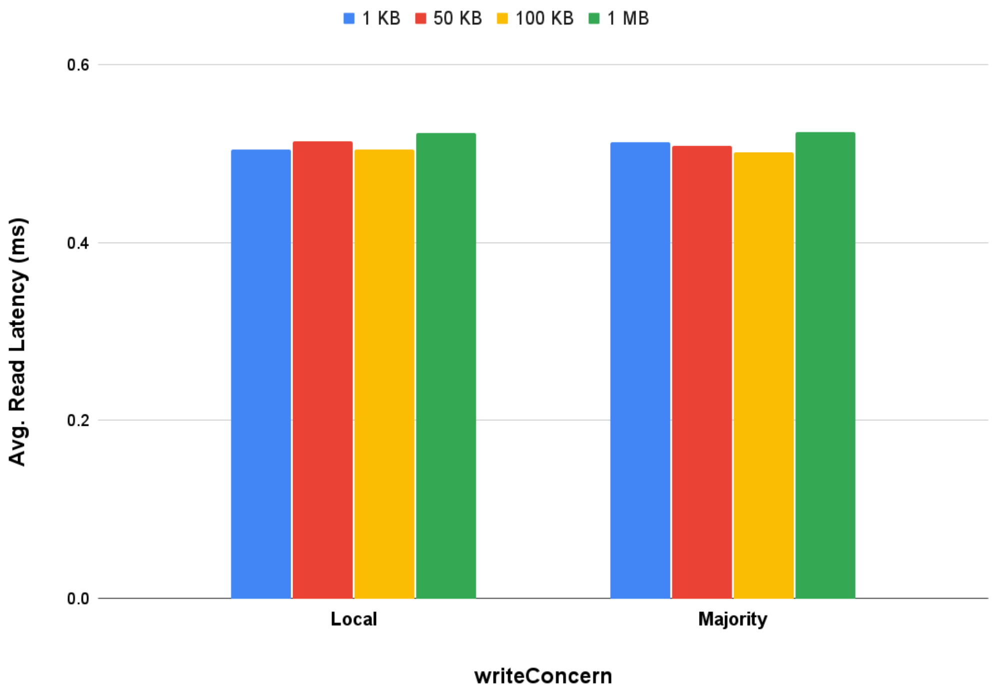

Figure 4.6: Comparison of Avg. Read Latencies for readConcern: majority

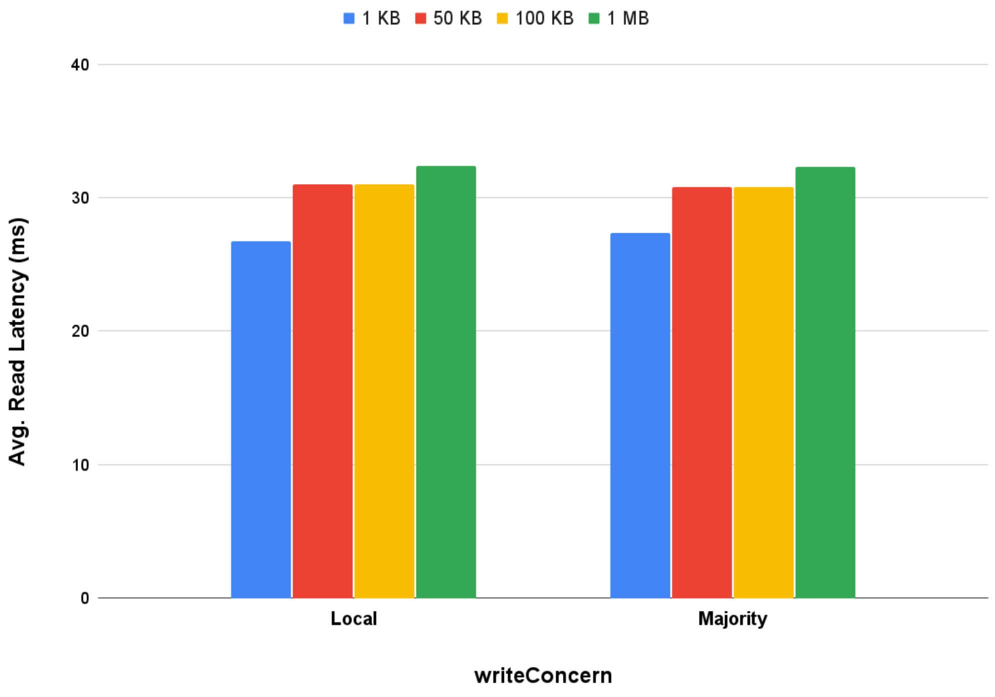

Figure 4.7: Comparison of Avg. Read Latencies for readConcern: linearizable

In Fig.4.8 and Fig.4.9, we can see that stale reads increase significantly with the increase in document size but only for (writeConcern:"local" ,readConcern:"majority") and (writeConcern:"majority",readConcern:"local"). The reason for that could be when a write operation is performed with writeConcern:"local", it means that the write operation must be acknowledged by at least one replica set member before the operation is considered successful. However, the read operation is performed with readConcern:"majority", which means that the read operation only considers data that has been acknowledged by the majority of replica set members.

This difference in writeConcern and readConcern levels can lead to a situation where a read operation may return stale data if the majority of replica set members have not yet acknowledged the write operation, even if the write operation has been acknowledged by the primary node. This effect can become more pronounced with larger document sizes, as there is more data to replicate and acknowledge.

On the other hand, when the write operation is performed with writeConcern:-majority and the read operation is performed with readConcern:- local, it means that the write operation must be acknowledged by the majority of replica set members, and the read operation only considers data from the local replica set member. In this case, the potential for stale reads is reduced since the read operation only considers data that is immediately available on the local replica set member, regardless of whether it has been acknowledged by the majority of replica set members or not.

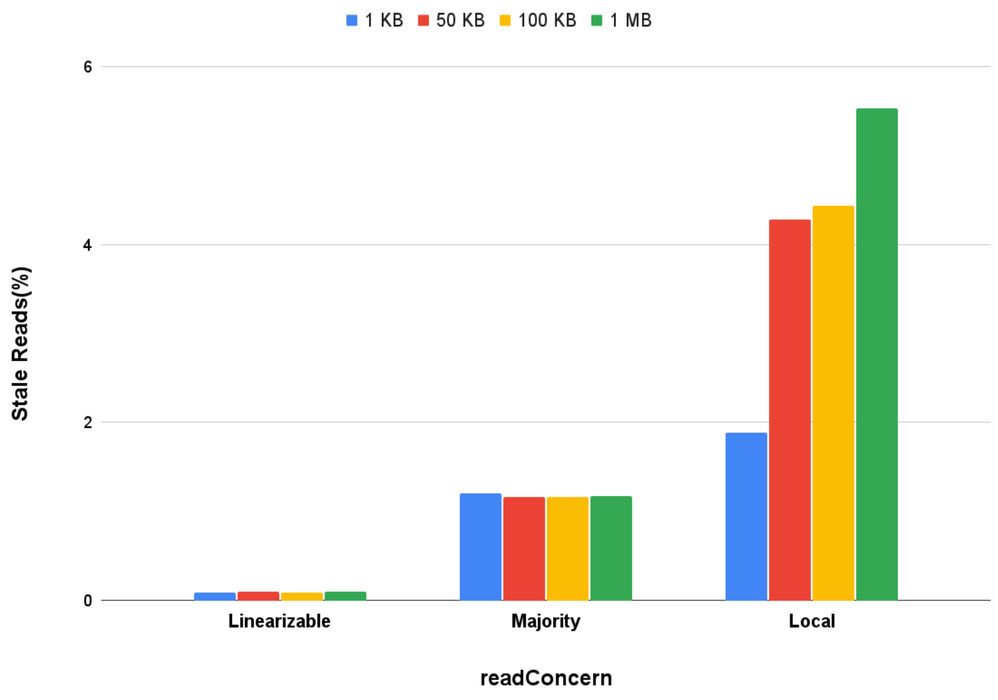

Figure 4.8: Comparison of Percentage Stale Reads for writeConcern: local

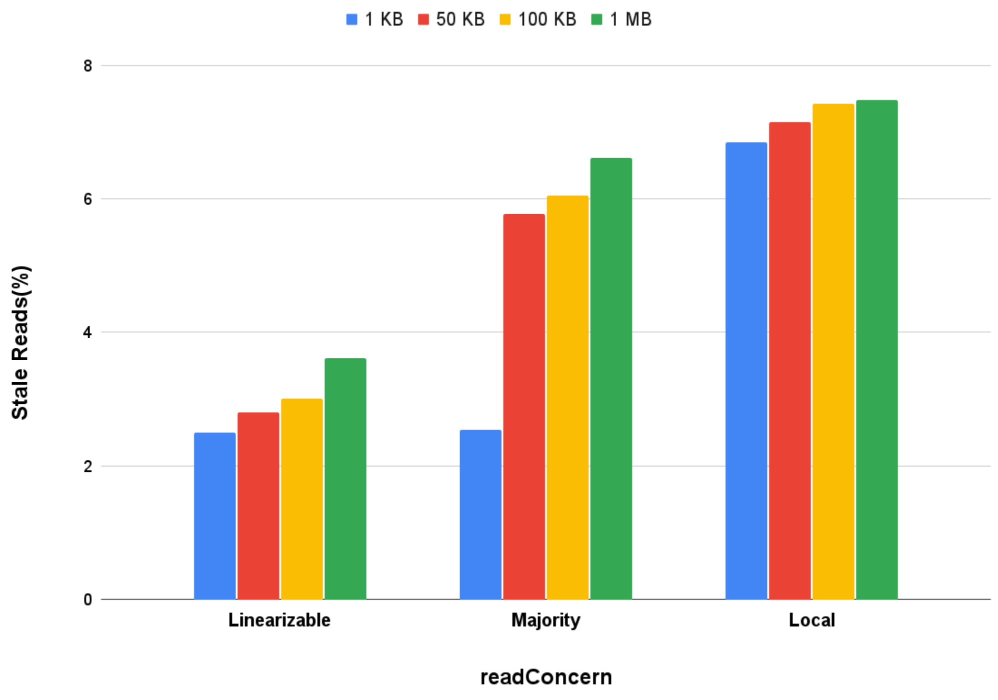

Figure 4.9: Comparison of Percentage Stale Reads for writeConcern: majority

4.3.3 Session-level Consistency Setting

This section discusses the results of the session-level consistency setup. As discussed before, in this setup we specify the consistency level when we start a session rather than during each and every individual operations. Although many of the results obtained for session-level consistency were similar to that of the operation-level consistency setting, we were able to confirm the trends that were seen in the operation-level consistency setup. There is no significant difference in the write latency and read latency for any configurations while comparing the results of session-level consistency and operation-level consistency.

In Fig.4.10 and Fig.4.11 we can see that the increase in document size leads to an increase in write latency which is similar to the one seen in operation-level consistency. Also, the write latency increases as we use stricter writeConcern values which are expected.

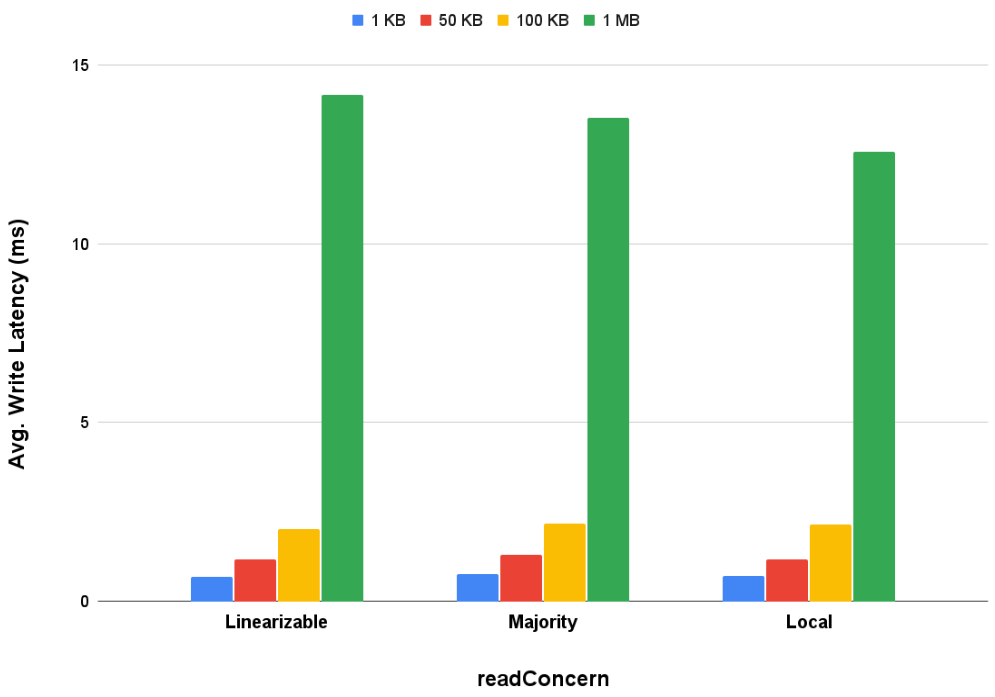

Figure 4.10: Comparison of Avg. Write Latency for writeConcern: local

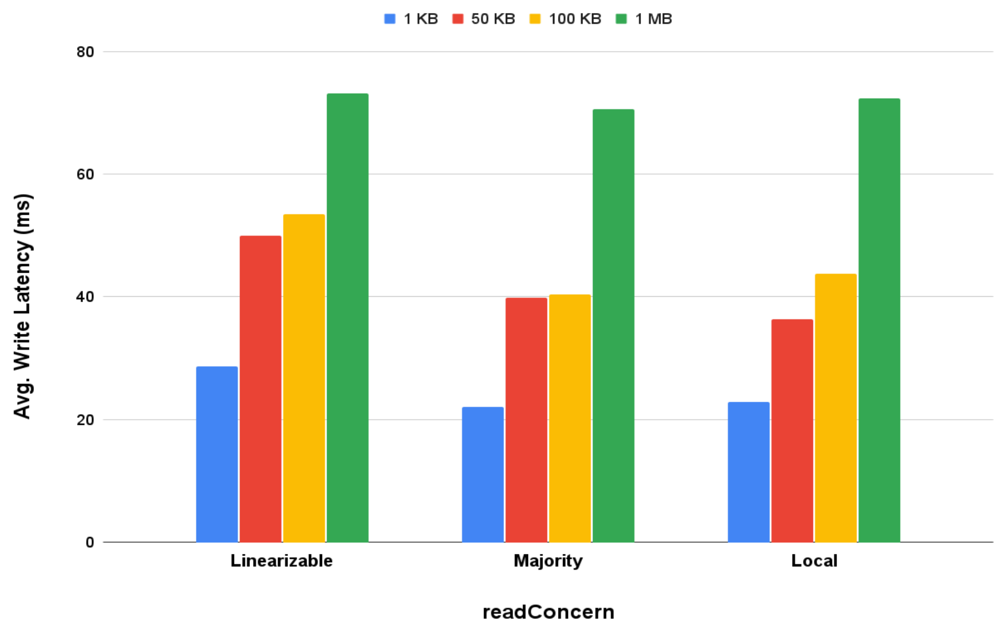

Figure 4.11: Comparison of Avg. Write Latency for writeConcern: majority

In Fig.4.12 and Fig.4.13 and Fig.4.14 we can see that the read latency is not much affected by the increase in document size. But there is an increase in read latency as we use stricter readConcern values.

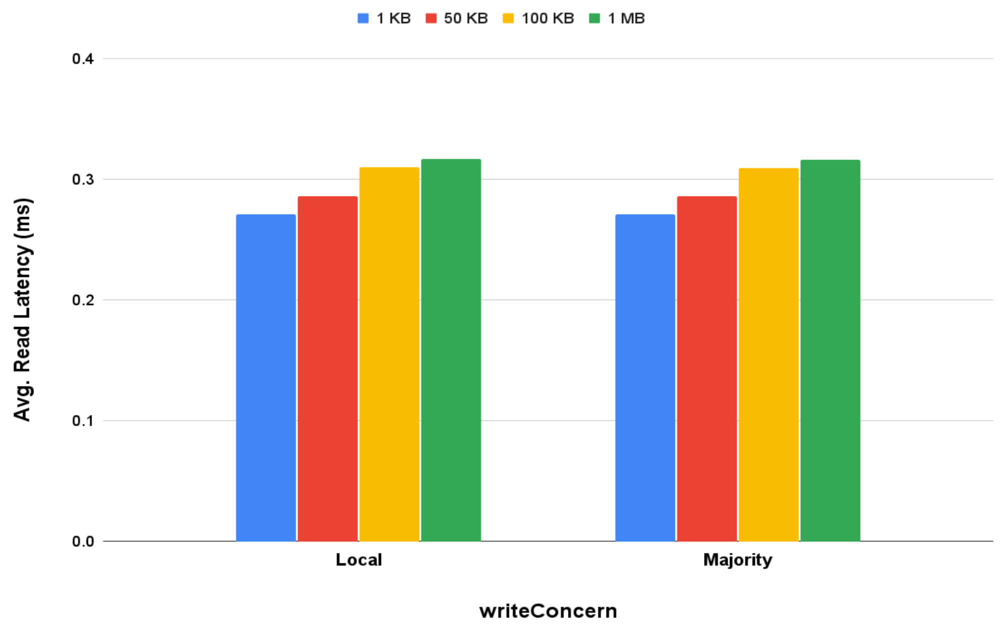

Figure 4.12: Comparison of Avg. Read Latencies for readConcern: local

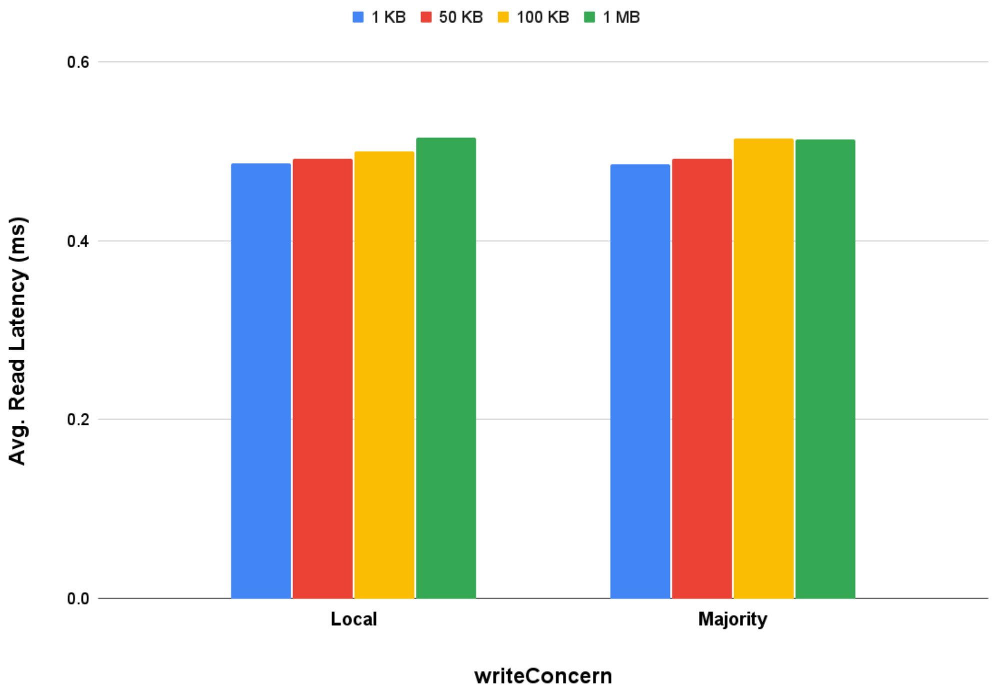

Figure 4.13: Comparison of Avg. Read Latencies for readConcern: majority

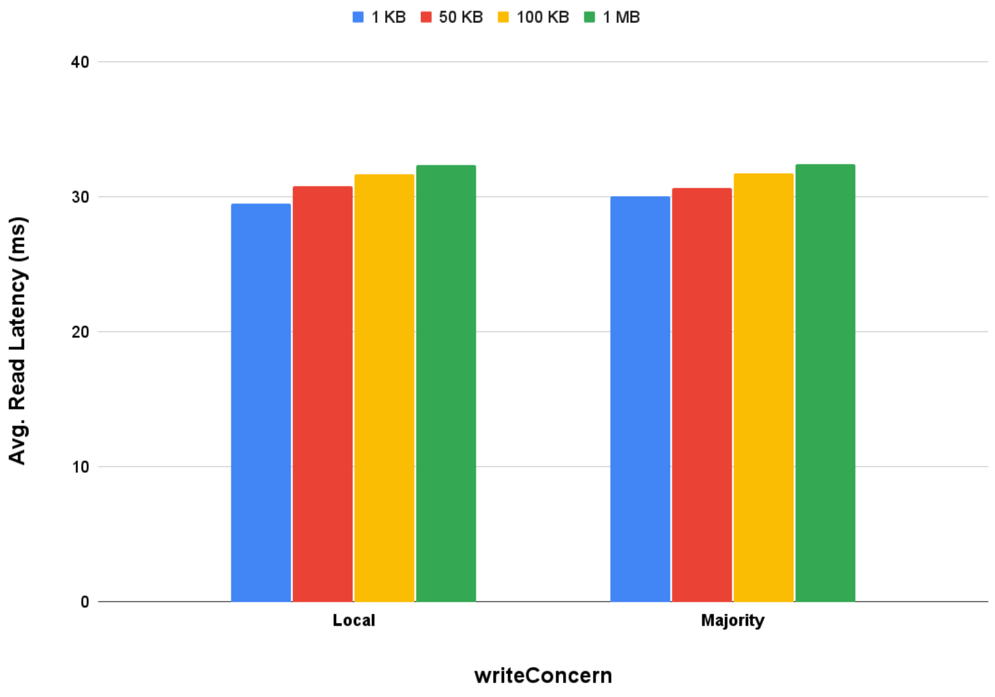

Figure 4.14: Comparison of Avg. Read Latencies for readConcern: linearizable

For Fig.4.16 we get the least amount of stale reads for writeConcern: "majority" and readConcern: "linearizable". And also in Fig.4.15 and Fig.4.16 there is an increase in the percentage of stale reads when increasing document size for (writeConcern: "majority",readConcern: "local") and (writeConcern: "local",readConcern: "local") for the same reasons which were discussed in the operation-level consistency setup. There is a slight increase in the percentage stale reads for writeconcern: "local" and readConcern: "local" when increasing the document size which can be seen in Fig.4.15.

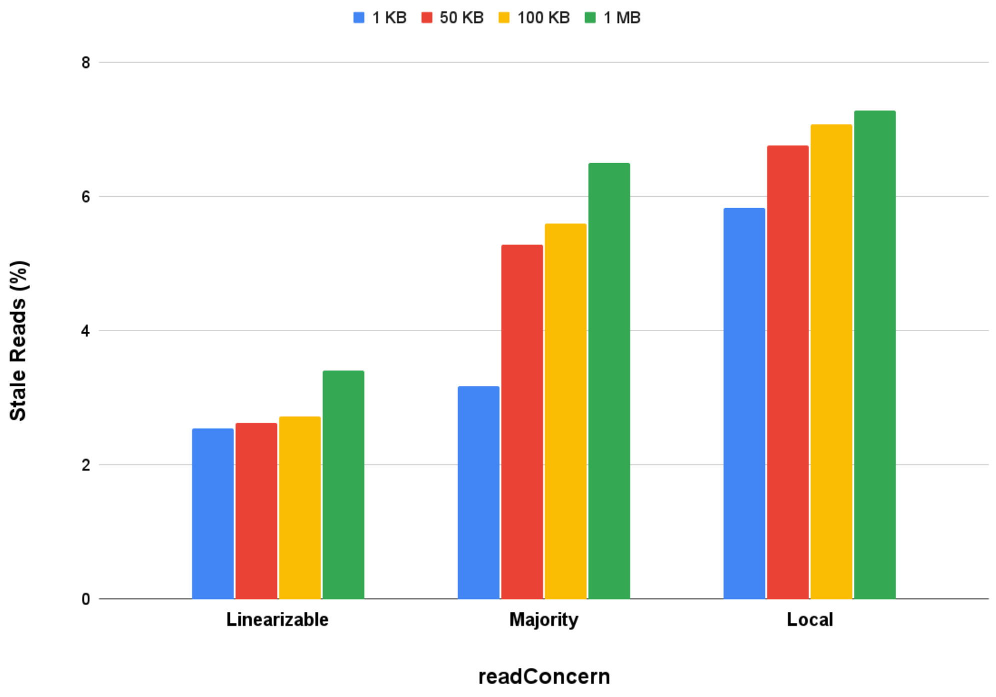

Figure 4.15: Comparison of Percentage Stale Reads for writeConcern: local

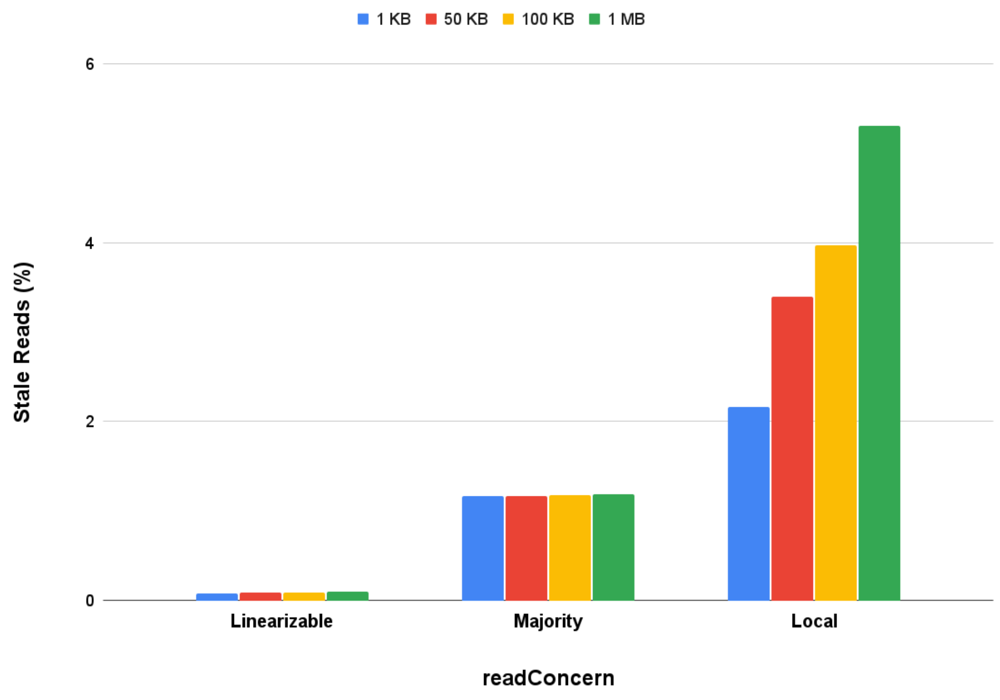

Figure 4.16: Comparison of Percentage Stale Reads for writeConcern: majority

4.3.4 Multiple Reader and Writer Setting

We had taken multiple readers and writers in this setup to increase the concurrency of operations and observe the performance of MongoDB.All the readers and writers were executed in different processes and were not interacting with each other. Fig.4.17 to Fig.4.23 show the results of the multiple reader-writer setup and the trends that we can see while varying the document size.

In Fig.4.17 and Fig.4.18 we can observe that the write latency increases as the document size increases and also the write latency increases as we use stricter writeConcern values. The write latency observed in each case is more then when we were using single writer and single reader. By increasing the number of readers and writers or in other words, by increasing the concurrency of operations in the system, we can see that the write latency has increased for every document size.

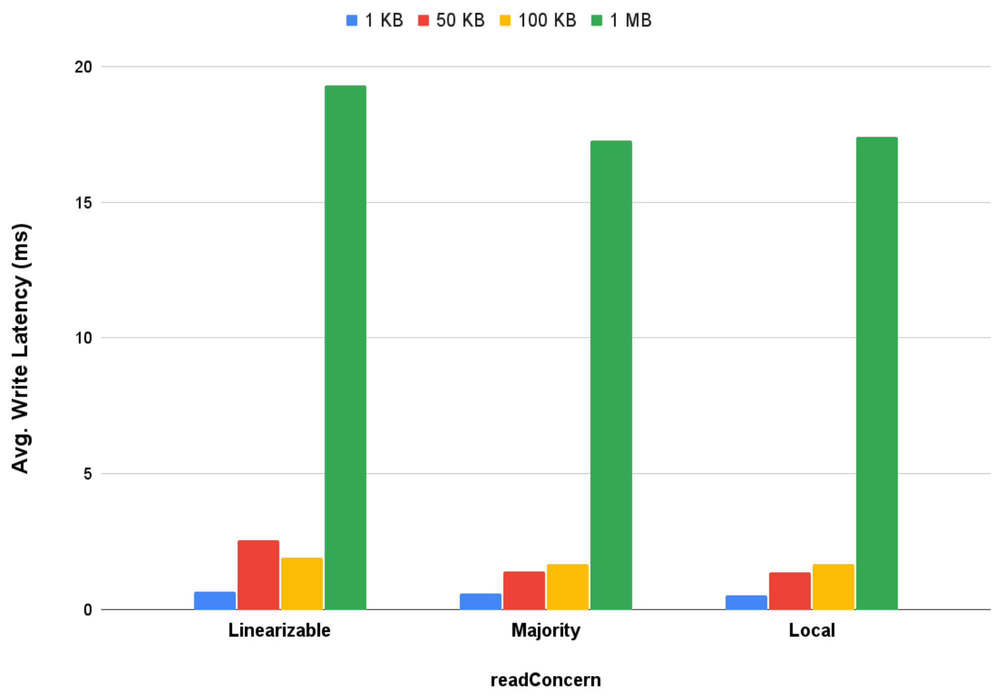

Figure 4.17: Comparison of Avg. Write Latency for writeConcern: local

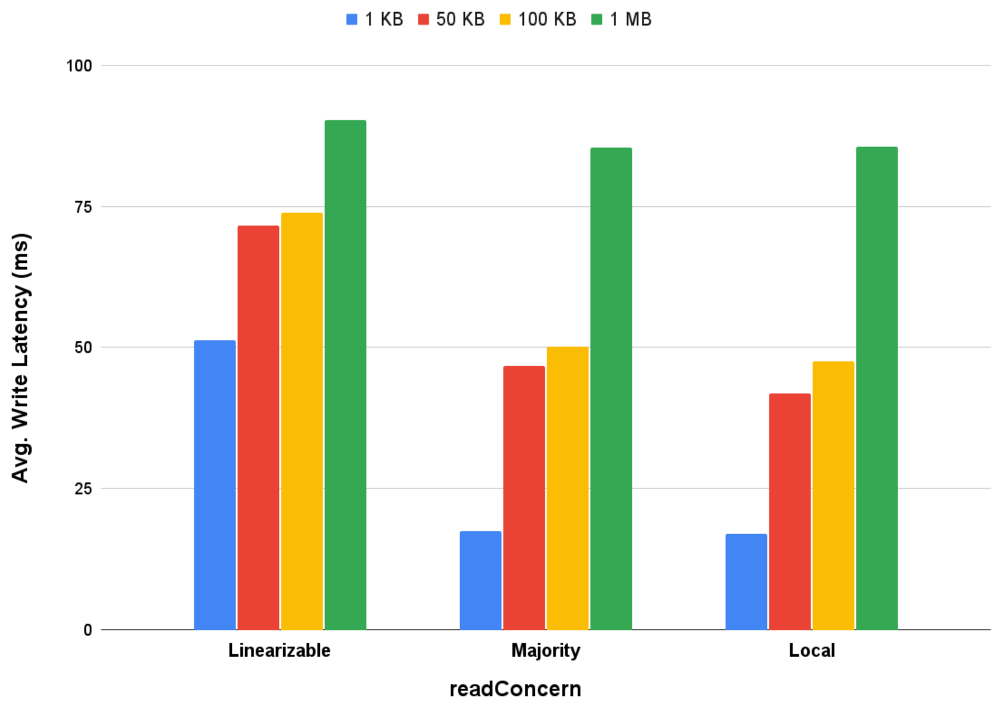

Figure 4.18: Comparison of Avg. Write Latency for writeConcern: majority

And now discussing about the Read latency, In Fig.4.19 and Fig.4.20 and Fig.4.21, we can see that the average read latency has increased for only (writeConcern: "local",readConcern: "local") and (writeConcern: "majority",readConcern: "linearizable") and for every document size. Also, there is a minor increase in read latency while increasing the document size.

In Fig.4.21, we can see that there is an increase in read latency when compared to operation-level consistency and session-level consistency but only for readConcern: "linearizable". There is no visible change in read latency for other readConcern values. Also, there is huge increase in read latency when we use linearizable readConcern then when we use the other two readConcern values.

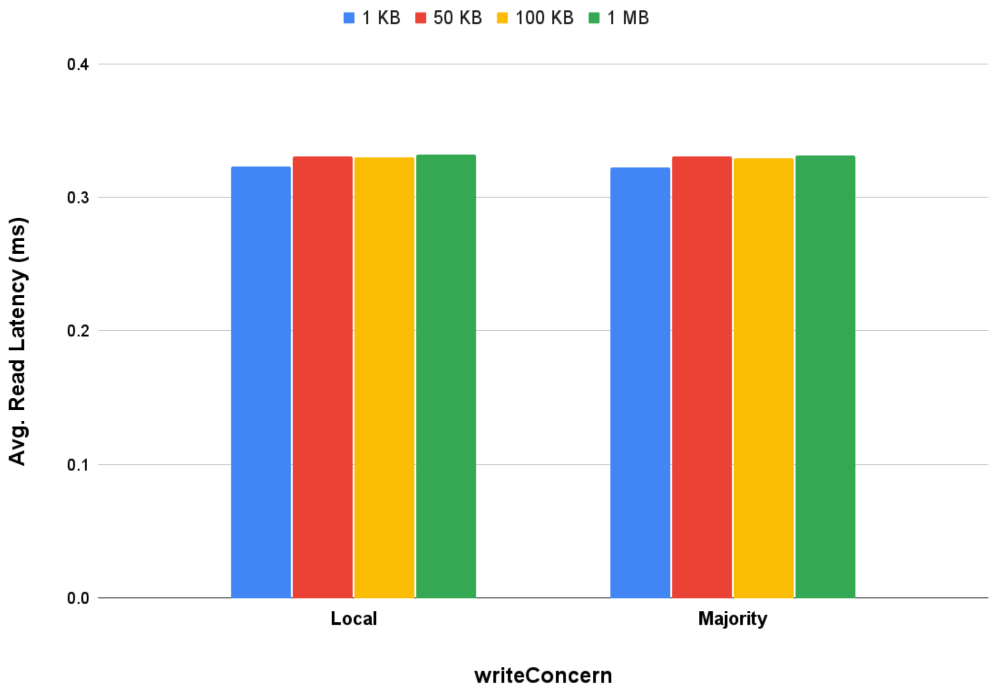

Figure 4.19: Comparison of Avg. Read Latencies for readConcern: local

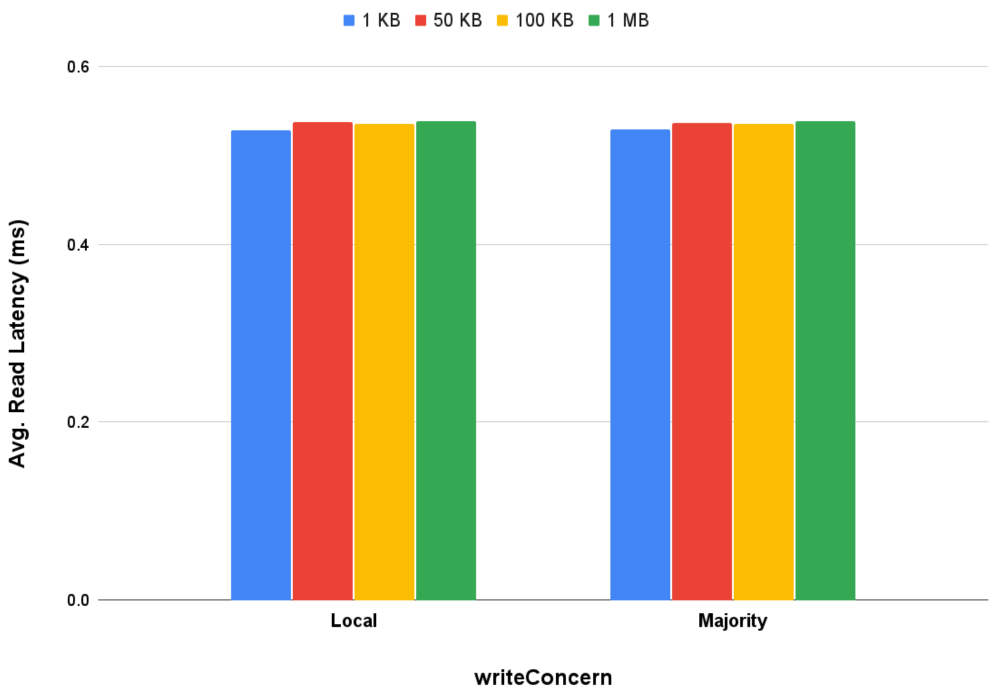

Figure 4.20: Comparison of Avg. Read Latencies for readConcern: majority

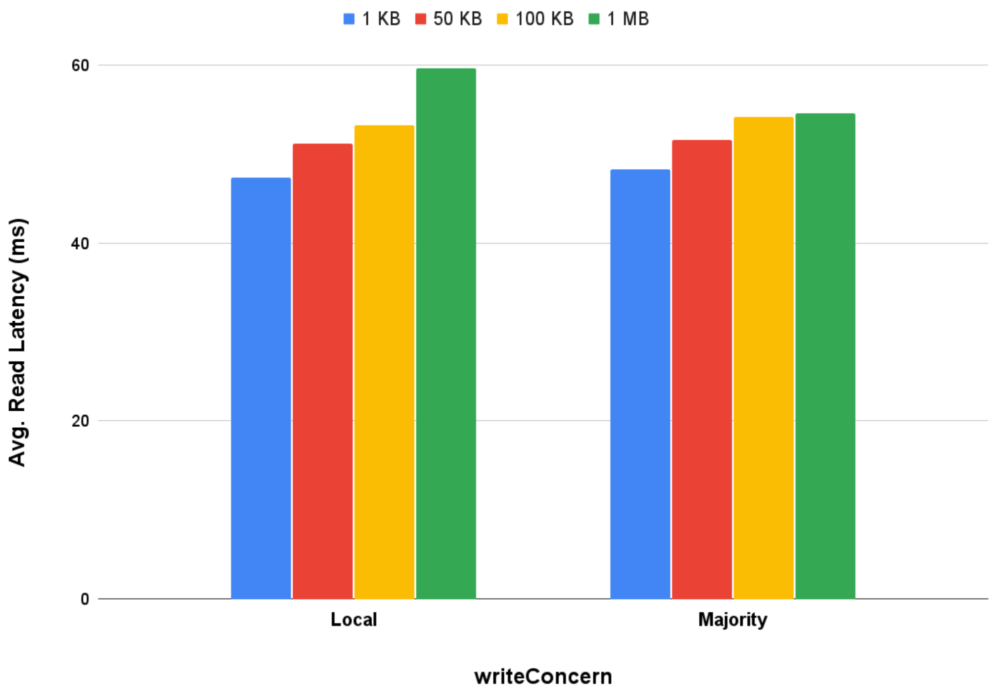

Figure 4.21: Comparison of Avg. Read Latencies for readConcern: linearizable

In Fig.4.22 and Fig.4.23, we can observe that when we look for the percentage of stale reads, they are increasing as compared to operation-level consistency or session-level consistency because of the increase in the workload of the system. Also, a similar trend is observed for this setup also, where the percentage of stale reads were increasing with document size for the pairs (writeConcern: "local",readConcern : "majority") and (writeConcern: "majority",readConcern: "local"). Also, some linear increase in stale reads with increasing document size is observed for writeConcern: "local" and readConcern: "linearizable".

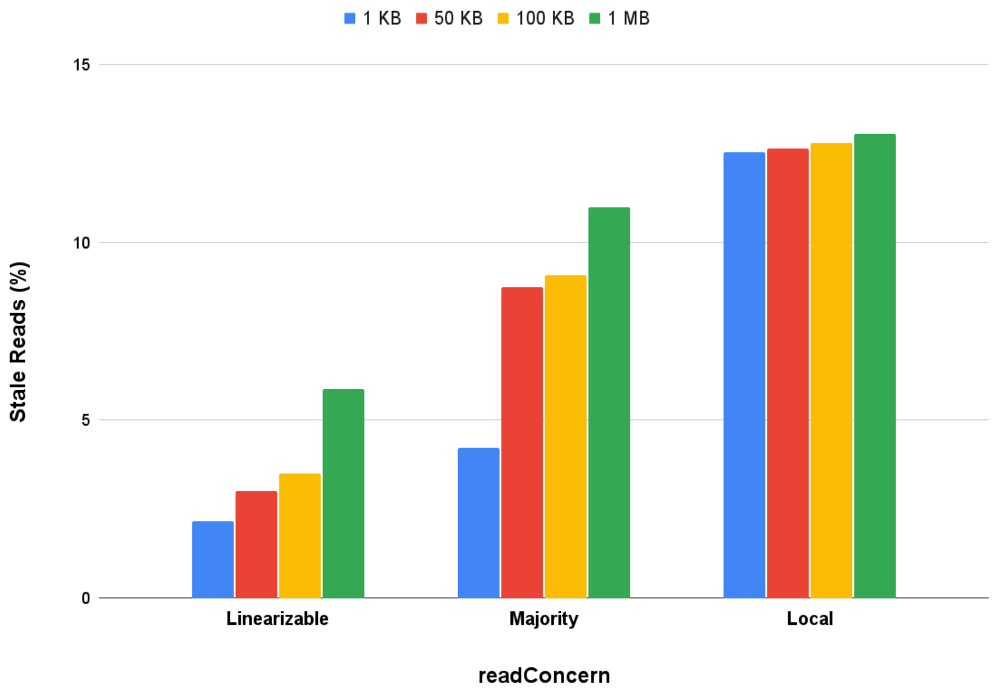

Figure 4.22: Comparison of Percentage Stale Reads for writeConcern: local

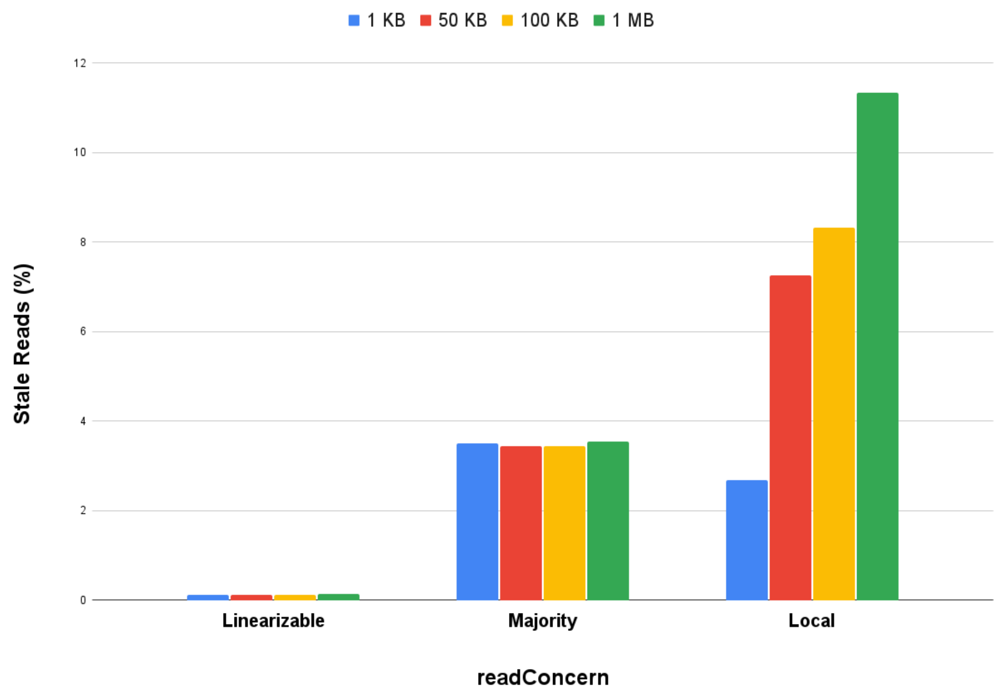

Figure 4.23: Comparison of Percentage Stale Reads for writeConcern: majority

CHAPTER 5

Conclusion and Future work

5.1 Conclusion

In this section, we provide the summary of the results obtained by varying the readConcern and writeConcern settings and the document size in our experiment. We compare readConcern:- linearizable with readConcern:- majority and the performance trade-offs that come with using the stronger readConcern values.

- For baseline concurrency, using "linearizable" readConcern resulted in 1.54 ms increase in the write latency and 32 ms average increase in the read latency compared to when we use the "majority" readConcern.
- The average difference in the amount of stale reads is 1.67%.

We also observe that as the document size and the number of simultaneous writers and readers increase, the system's consistency decreases. This means there's a trade-off between consistency and scalability.

- Increasing the document size has a significant impact on the write latency, with the average increase in write latency being 135.89%. However, the document size does not have a significant impact on read latency, with the average increase being only 2.98%.
- For increased concurrency where we use multiple readers and writers, using linearizable readConcern resulted in 11 ms average increase in the write latency and 51.92 ms average increase in the read latency.
- The difference in the amount of stale reads is 2.73%.

Both writeConcern and readConcern majority provide a proper balance between the consistency guarantee and the write and read latency with some reasonable amount of stale reads.

5.2 Future work

Looking ahead, there are several possibilities for future research. Firstly, one could explore different workloads and benchmarking methods to gain a deeper understanding of MongoDB's consistency behaviour. One could also investigate how network latency and geographic distribution affect consistency, especially in distributed database environments. Furthermore, it would be valuable to explore alternative consistency models and techniques for MongoDB, which could involve experimenting with different combinations of writeConcern and readConcern settings or exploring hybrid models that blend strong and eventual consistency guarantees.

Lastly, studying how consistency impacts specific application domains and workloads would be beneficial. By understanding the requirements and trade-offs in scenarios like e-commerce transactions, real-time analytics, or collaborative editing, users can make more informed decisions when configuring MongoDB for their specific use cases.

References

[1] Mongodb documentation. https://www.mongodb.com/docs/, 2023. Accessed: June 2023.

[2] D. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *IEEE Computer*, 45:37–42, 02 2012.

[3] E. Brewer. Towards robust distributed systems. page 7, 07 2000.

[4] C. Huang, M. J. Cahill, A. D. Fekete, and U. Röhm. Data consistency properties of document store as a service (dsaas): Using mongodb atlas as an example. In *TPC Technology Conference*, 2018.

[5] H.-R. Ouyang, H.-F. Wei, H.-X. Li, and et al. Checking causal consistency of mongodb. *Journal of Computer Science and Technology*, 37:128–146, 2022.

[6] W. Schultz, T. Avitabile, and A. Cabral. Tunable consistency in mongodb. *Proc. VLDB Endow.*, 12(12):2071–2081, aug 2019.

[7] M. van Steen and A. Tanenbaum. *Distributed Systems*. distributed-systems.net, 3rd edition, 2017.

[8] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, jan 2009.