School of Information Technology and Engineering at the ADA University

School of Engineering and Applied Science at the George Washington University

DEFINING LINEARIZABILITY IN DISTRIBUTED SYSTEMS BY LINEARIZABILITY CHECKER

A Thesis
Presented to the Graduate Program of Computer Science and Data Analytics
of the School of Information Technology and Engineering
ADA University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science and Data Analytics
ADA University

By
Hasan Naghiyev

April, 2023

THESIS ACCEPTANCE

This Thesis by: Hasan Naghiyev
Entitled: *Defining Linearizability in Distributed Systems by Linearizability Checker*

has been approved as meeting the requirement for the Degree of Master of Science in Computer Science and Data Analytics of the School of Information Technology and Engineering, ADA University.

Approved:

Jamaladdin Hasanov, PhD	
(Adviser)	(Date)
Abzatdin Adamov, PhD	
(Program Director)	(Date)
Abzatdin Adamov, PhD	
(Dean)	(Date)

ACADEMIC INTEGRITY STATEMENT

“I affirm that this is my own work, I attributed where I used the work of others, I did not facilitate academic dishonesty for myself or others, and I used only authorized resources for my Thesis, per the ADA University Academic Integrity requirements. If I failed to comply with this statement, I understand consequences will follow my actions. Consequences may range from failing the course to expulsion from the program/university and may include a transcript notation.”

Naghiyev Hasan		24.04.2023
(Full Name)	(Signature)	(Date: DD.MM.YY)

ABSTRACT

Distributed systems are a vital part of contemporary computing infrastructure, and upholding data consistency across these systems is an essential aspect of their design. Consistency models, including strong, sequential, and eventual consistency, are pivotal in ensuring coherence and precision of data within distributed systems. Nevertheless, the choice of an appropriate consistency model is contingent upon the specific needs of a project, as it frequently entails balancing trade-offs among consistency, availability, and partition tolerance, as emphasized by the CAP theorem. In this study, we concentrate on strong consistency, or linearizability, since it offers an instinctive approach to reasoning about data in distributed systems.

The main goal of this study is to create a linearizability checker that can analyze files from client-distributed databases and pinpoint transactions that violate strong consistency. The checker also yields output proposing the accurate values for these transactions. Our methodology encompasses gathering query history files from multiple database instances, merging them into a unified history file, and subsequently examining this file for consistency violations. A report is then generated to inform the user about any detected inconsistencies.

Our linearizability checker is composed of two subsystems: the User Interface and the Server. The User Interface is accountable for securely obtaining input files from the user, transferring them to the server, and presenting the generated report. In contrast, the Server is in charge of merging the input files, validating strong consistency, producing a report, and returning it to the User Interface. To tackle the time skew issue in distributed systems, we employ the Network Time Protocol (NTP) to synchronize clocks throughout the system.

Currently, our checker is confined to working with Redis database files; however, it can be expanded to accommodate additional databases in the future. By identifying consistency violations in distributed systems, our linearizability checker serves as an indispensable instrument for ensuring data accuracy and integrity in enterprise projects.

Keywords: Linearizability, Consistency, Distributed systems, CAP theorem, Time skew

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abbreviation	Explanation
BFT	Byzantine Fault Tolerant
CLP	Constraint-Based Synthesis
GFS	Google File System
GPS	Global Positioning System
JDK	Java Development Kit
NTP	Network Time Protocol
PAT	Process Analysis Toolkit
PBS	Probabilistically Bounded Staleness
PNUTS	Platform for Nimble Universal Table Storage
PRI	Public Key Infrastructure
PTP	Precision Time Protocol
SDK	Software Development Kit
SQL	Structured Query Language
TAO	The Association and Objects
UNIX	UNiplexed Information Computing System

1 INTRODUCTION

1.1 Definition of the Problem

In the context of distributed systems, one cannot overlook the importance of consistency models as an integral element. Different enterprise projects have different consistency models such as strong, sequential, eventual, etc. There is not a consistency model that we can say best suits all projects, rather it depends on the needs of a project. For instance, if you choose to adopt strict consistency, you may sacrifice availability. This paper will refer to linearizability as strong consistency. It analyzes different consistency models and scrutinizes the consistency models that can be utilized. There are some consistency models such as strong consistency, per-object-sequential consistency, read-after-write consistency, eventual consistency, and other custom consistency models that companies create for themselves according to their use cases. Firstly, we would like to start with a brief introduction to eventual consistency which, is mainly used in the enterprise distributed system world. Eventual consistency requires that if replicas or nodes get the same list of writes, they eventually must have the same value. This model allows all read queries to be returned by replicas immediately. However, writes take some time to be propagated between the set of replicas, therefore, while this propagation happens, you may get different values from different replicas for the same get request. More formally, writes may not be reflected in all replicas within time t after the write is sent to the system. In a system that eventually achieves consistency, many replicas may accept writes concurrently. However, strong consistent systems do not accept this t time which is allowed by eventual consistent systems. Between the time the client calls the operation and the time it receives the answer, strong consistency makes sure that each operation appears to take effect instantly at some point. Formally, linearizability requires that there be a total overall system operation that are consistent with the sequence of operations carried out in real-time. For instance, if operation A is finished before operation B, A will be prioritized over B. By guaranteeing that writes are implemented in a real-time, sequential order and that reads always view the outcomes of the most recent writing, linearizability prevents anomalies. One of the advantages of a linearizable system is that it takes the responsibility of handling consistency in the software part by writing complex logic. Thirdly, another consistency model which is read after write consistency model states that within each cluster (set of servers) ensuing read requests to that cache always reflect the most recent writing or any subsequent writes once a write request has been committed. Another significant consistency model is per-object-sequential which means that there is always one version of an object for each new client and each client operates on his own object it guarantees that the client will always see the last version of the object that they own, but it may obviously cause different clients to have a different version of the object. So far, l tried to give brief information about some consistency models, henceforth, I will analyze strong consistency and how to make sure that distributed system which proposes strong consistency is strong consistency or not. Linearizability in other words strong consistency is required in some parts of enterprise projects and makes software logic easier and applications more trustworthy. In a write skew, two transactions read a set of objects

and modify some of those objects. However, the changes that each transaction makes affect what other transactions should read. In this case, the chances of data inconsistency are high. For instance, two transactions read the same row from the database, perform computations based on what they read, and then both commits based on the data read, not on the data recently modified by either transaction which may cause inconsistency in the database. For the aforementioned reasons, strong consistency is important in some parts of the system where consistency is more important than availability. An eventually consistent system is more available than strong consistent; however, availability does not always have an important role which especially comes to bank transactional data. Heretofore, I have discussed the consistency models and why strong consistency is significant in enterprise systems. A question arises how can we make sure that our distributed databases ensure strong consistency before going to a production environment? We can never be sure that our system is strongly consistent until performing some checks. The tests performed by developers are, in general, limited to unit, integration, and functional tests. However, such tests mainly do not check all possible cases to simulate whether the system is strongly consistent or not. Thus, there is usually an additional need for functional and end-to-end testing.

Another important problem relevant for this study is time skew. In a distributed system, multiple nodes work together to provide a service. Each node has its own clock, and these clocks may not be perfectly synchronized with each other. This can lead to time skew, which is a discrepancy between the perceived time on different nodes.

1.2 Objective of the Study

The main objective of this paper is to build a linearizability checker that acquires a file from client-distributed databases and by analyzing this file, generates output such as which transactions violated strong consistency, and what should be the correct value for those transactions. This system is developed for those who build their database in a distributed fashion and want to achieve strong consistency in the production environment. The system will take the input file and produce an output file based on that. The size of the input file does not matter to the system, since it shall be designed to handle big files. Moreover, the file content will be securely uploaded to the system since it contains many productional data. This tool will detect consistency violations in a client database and will report to the client about that. The client will easily know where the problem is and what should be the correct value for that violation. This checker may be advanced by not getting the ready files from the client but by getting database credentials and connecting to the database itself and extracting the file. The system is expected to have one internal model for file structure and have different mappers for each key-value store (Redis, mongo, etc.). Later more databases can be integrated into the system. The main challenge is that the program should understand each key value language statement regardless of simple or complex query. This state will be stored in key-value data structures such as hash tables. This key-value store can be decoupled from the application by introducing Redis or any technology that supports key-value data structure.The linearizability checker's main goal is to offer a complete method for identifying and fixing consistency issues in distributed databases. This tool will enable developers to retain strong consistency in their production

settings while supporting different database technologies by securely processing input files, producing thorough output reports, and providing the opportunity for direct database integration.

1.3 Significance of the Problem

Achieving strong consistency is a paramount issue in the domain of computer science.

The linearizability problem is important because it helps maintain consistency, accuracy, and dependability in concurrent systems. The need of upholding consistency across these systems grows as current software systems increasingly rely on parallelism and distributed computing to achieve high performance and scalability. This need is met by the powerful consistency model of linearizability, which offers a precise set of guidelines for the conduct of concurrent operations on shared data structures. Recognizing the difficulties in concurrent systems is necessary for understanding the significance of linearizability. When several threads or processes run concurrently, their actions may interleave in a number of different ways, resulting in unpredictable results and perhaps inconsistent shared data structures. The system becomes unreliable and prone to mistakes as a result of these inconsistencies, which can lead to subtle defects that are challenging to find, identify, and remedy. In order to ensure that the outcomes of concurrent operations on shared data structures appear as though they were carried out sequentially, linearizability imposes a rigid ordering of operations. This consistency model makes it easier to understand how concurrent systems behave, enabling developers to create software that is more dependable and durable. The importance of the linearizability problem also extends to a number of fields and applications, such as databases, distributed systems, and real-time systems, where it is crucial to ensure consistency among numerous nodes or processes. In these systems, ensuring linearizability can assist prevent data loss, preserving data integrity, and ensure that the system functions as intended under varied circumstances. Developers can get a number of additional benefits that improve the overall quality and performance of their software systems by adhering to a rigid consistency model like linearizability.

1. Simplified debugging and testing: Linearizability makes it simpler to think about concurrent systems, which makes it simpler for programmers to test and debug their programs. Linearizability decreases the difficulty of comprehending the many interleavings of concurrent operations and makes testing and debugging procedures simpler by requiring a precise sequencing of activities.
2. Decreased programming complexity: The capacity of linearizability to reduce programming complexity is one of its main benefits. Due to linearizability, which offers an intelligible paradigm for understanding the behavior of concurrent operations and abstracts away many concurrency-related problems, developers can write code as if they were working with sequential operations. This simplification might produce code that is clearer, easier to maintain, and less likely to make concurrency-related mistakes.
3. Improved system predictability: Linearizability assures that operations on shared data structures follow a rigid sequence, which improves the predictability of concurrent systems. This ordering ensures that the system operates consistently and predictably, which makes it

simpler for programmers to analyze the system's behavior and create more dependable and stable applications.

4. Enhanced system resilience: By avoiding the data corruption and consistency issues that can result from concurrent processes, ensuring linearizability can help in the development of more resilient systems. Strong consistency models, like linearizability, enable systems to withstand errors and bounce back from failures, thereby increasing the robustness of the system as a whole.
5. Wide applicability: Databases, distributed systems, and real-time systems are just a few of the domains and applications where linearizability is important because it ensures consistency across numerous nodes or processes. In these systems, ensuring linearizability can assist prevent data loss, preserving data integrity, and ensure that the system functions as intended under varied circumstances.

The famous CAP theorem [16], also known as Brewer's theorem, is relevant for explaining its importance. It is a fundamental concept in distributed computing that highlights the inherent trade-offs that exist between consistency, availability, and partition tolerance in a distributed system. The theorem was first proposed by Eric Brewer in 2000, and it has since become a cornerstone of distributed system design. In essence, the CAP theorem states that it is impossible for a distributed system to simultaneously guarantee all three of the following properties: consistency, availability, and partition tolerance. Consistency refers to the property that all nodes in the system see the same data at the same time, availability means that the system remains operational even in the face of failures, and partition tolerance means that the system can continue to function even if the network is partitioned or split into multiple disjoint parts. The theorem argues that in the event of a network partition or failure, a distributed system must choose between ensuring consistency or ensuring availability. This trade-off is particularly acute in systems that prioritize strong consistency, which is a consistency model that guarantees that all nodes in the system agree on the same value for a given data item at all times.

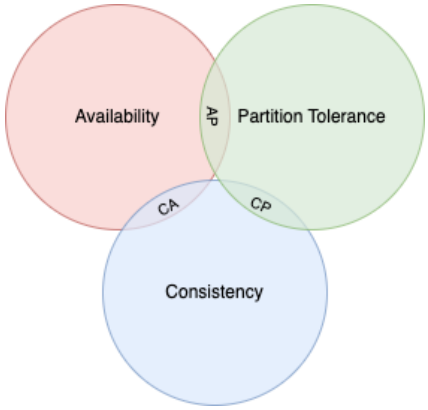

Figure 1. Visualization of CAP Theorem

Strong consistency is a desirable property in distributed systems because it provides a straightforward and intuitive way to reason about data. However, as the CAP theorem shows, achieving strong consistency while also maintaining availability and partition tolerance is impossible. This means that developers must make a trade-off between strong consistency and other desirable properties of a distributed system. Without strong consistency, it can be difficult to reason about data, and inconsistencies between nodes can lead to incorrect results or even system failures.

Achieving strong consistency can be challenging, but it is essential for ensuring that distributed systems are reliable, correct, and easy to reason about. By understanding the trade-offs involved in distributed system design and choosing appropriate strategies for achieving strong consistency, developers can build systems that meet their requirements and provide reliable and consistent performance.

1.4 Review of Significant Research

The linearizability checker is implemented by using various algorithms with different complexity and techniques. One of the fastest linearizability checkers implemented by Anish Athalye named "Porcupine" [12]. Porcupine is a tool for determining whether concurrent data structures are linearizable. A concurrent data structure must satisfy the linearizability requirement in order to act as if there is only one copy of it and for operations to appear to happen atomically in some total er. P-compositionality, the quickest known linearizability testing procedure, is implemented by Porcupine. An approach for determining if concurrent data structures are linearizable called P-compositionality has various advantages over previous linearizability checking algorithms. The fact that it does not involve examining all potential interleavings of operations—which can have exponential time and space complexity and is the root of the O(N!) problem in linearizability checking is one of its main advantages. The fundamental concept behind P-compositionality is to determine whether a concurrent data structure is linearizable by first determining whether each individual operation is linearizable in isolation, and then by determining whether the composition of these operations is linearizable. As the linearizability of each operation is examined separately from other operations, this method removes the need to investigate all potential interleavings of

operations. The approach builds a history graph that depicts the series of events that take place throughout the execution of a concurrent data structure in order to implement P-compositionality. The history graph contains details on each operation's beginning and ending times as well as the relationships between them, such as which activities are carried out concurrently and which are nested inside of others. After creating the history graph, the method determines if each operation is linearizable separately by first creating a partial order of the events in the history graph and then determining whether this partial order satisfies the linearizability requirement. By mapping the events in the partial order to intervals on a timeline using the interval projection technique, this is accomplished. The method then determines if the partial orders for each operation can be merged into a single partial order that meets the linearizability condition. This checks the compositionality constraint. Calculating the transitive closure of the partial orders and ensuring that the resulting order meets the linearizability requirement are used to achieve this. P-compositionality is superior to alternative linearizability checking algorithms in a number of ways. For instance, it saves the need to investigate all potential operation interleavings, which could necessitate exponential time and space complexity. The approach can also be used to determine whether a concurrent data structure is linearizable in a distributed environment where various components of the data structure are spread across various nodes. Top of Form Porcupine creates a data structure called a history graph, which depicts the series of events that take place throughout a run of a concurrent data structure, in order to implement P-compositionality. The history graph shows the beginning and ending timings of each operation as well as the connections between them (e.g., which operations are concurrent and which operations are nested inside others. Using a partial order of the events in the history graph and a check to see if it meets the linearizability criterion, Porcupine verifies the linearizability of each operation separately after building the history graph. By mapping the events in the partial order to intervals on a timeline using the interval projection technique, this is accomplished. In order to verify that the partial orders for each operation can be combined into a single partial order that meets the linearizability condition, Porcupine tests the compositionality condition one last time. Calculating the transitive closure of the partial orders and ensuring that the resulting order meets the linearizability requirement are used to achieve this. Based on the Python programming language, Porcupine has a command-line interface for determining whether a given concurrent data structure is linearizable. Along with visualizations of the history graph and the partial orders, the tool also supports specifying custom data types and processes. Porcupine is a tool that uses the P-compositionality technique to verify the linearizability of concurrent data structures. A history graph is built, each operation's linearizability is verified using interval projection, and the compositionality requirement is verified by computing the transitive closure of the partial orders. Another tool for checking linearizability is known as “Knossos” [13], however Porcupine is faster and has a smaller memory footprint. Porcupine exceeds other tools where it takes advantage of P-compositionality. A research team that specializes in verifying the correctness of distributed systems, Jepsen, created Knossos, a linearizability checker. Knossos is used to check that various distributed databases and other systems are functioning properly under various failure scenarios and network partitions. It is built to work with a wide range of distributed databases and other systems. The Knossos linearizability checker employs a state-space exploration and execution tracing-based technique. To

determine if a distributed system is linearizable, Knossos combines state-space exploration and execution tracing. Dynamic symbolic execution, a method for exploring all possible routes through the system and locating all potential linearization sites for each operation, is used to carry out the state-space exploration. Knossos employs an execution trace to verify that the system's behavior is consistent with a linearization of the operations after the linearization points have been found. The trace is produced by carrying out a series of operations on the system and documenting the series of events that take place (e.g., messages sent and received, locks acquired and released, etc.) The trace is then checked against the set of linearization points to ensure that the observed behavior is consistent with a valid linearization of the operations. Knossos employs a method known as partial order reduction to manage complicated activities and interactions between numerous clients. By locating duplicate event interleavings and removing them from consideration, this approach shrinks the state space. The capacity of Knossos to manage systems with non-deterministic behavior is one of its main features. Concolic testing, a method for doing this, combines dynamic symbolic execution with concrete execution to explore alternative system paths and find every potential linearization point. Overall, the technique proposed by Knossos provides a strong and adaptable method for determining if distributed systems can be linearized. However, as mentioned before, it may not be suitable for all use cases or circumstances and can be computationally expensive.

In their 2010 study titled "Line-up: a full and automatic linearizability checker" [6] Sebastian et al. introduced another linearizability checker. In order to minimize the time needed to confirm the correctness of such systems, this research presented a novel technique to test the linearizability of concurrent systems.

1. Introduction The Line-Up paper's authors presented a novel, fully automatic method for verifying linearizability. The idea behind the suggested approach is to investigate the space of all potential sequential specifications for a specific concurrent system. With this method, the authors are able to both automatically deduce a correct sequential specification for a linearizable concurrent system as well as detect violations of linearizability.
2. Approach The Line-Up technique relies on three main components:

   2.1 Random Testing in Parallel Running a concurrent system with numerous threads that produce random operations is what parallel random testing involves. By looking at the generated execution traces, the goal is to find potential linearizability violations.

   2.2 Sequential Specification Inference, a specific concurrent system, this component must deduce the appropriate sequential specification. Based on the notion of constraint logic programming, the authors employ a method known as "constraint-based synthesis" (CLP). In order to identify a solution that satisfies these requirements, a set of constraints describing the behavior of the system must first be generated.

   2.3. Linearizability Checking Linearizability checking in Line-Up is done by comparing the observed behavior of the concurrent system with the inferred sequential specification. If the observed behavior is consistent with the inferred sequential specification, the system is considered linearizable.
3. Evaluation The authors tested Line-Up against queues, stacks, and sets, among other concurrent data structures. They identified previously unknown vulnerabilities in these data structures and

successfully found known linearizability violations in them. For linearizable implementations, they could also automatically deduce the right sequential specifications.

As a result, the Line-Up linearizability checker offers a thorough and automatic method for determining whether concurrent systems are linearizable. This method lessens the work necessary to guarantee the accuracy of concurrent data structures by integrating simultaneous random testing, sequential specification inference, and linearizability checking.

In their 2013 publication titled "Systematic Testing for Concurrency Defects in Erlang Programs" [19], Maria et al. provided yet another linearizability tester for concurrent Erlang applications. Concuerror was created by the authors to find and fix concurrency flaws in Erlang applications, with a particular emphasis on deadlocks, crashes, and violations of linearizability. The authors introduced a methodical testing tool created especially for concurrent Erlang programs. Concuerror uses partial-order reduction techniques to limit the amount of interleavings that need to be studied, which speeds up the testing process. Concuerror systematically investigates the space of potential thread interleavings in a particular Erlang application. Concuerror can identify different concurrency issues, including deadlocks, crashes, and violations of linearizability. Concuerror employs stateless model checking, a method that does not necessitate the explicit creation of a state space, for linearizability checking. Concuerror also creates test cases for the specified Erlang program in an effort to cover as many potential execution scenarios as it can. The program has two input options: user-provided test cases or test cases that are automatically generated. Concuerror was tested by the authors on a variety of concurrent Erlang programs, including distributed databases, servers, and caches. They were successful in finding several concurrent issues in these programs, including deadlocks, crashes, and violations of linearizability. Additionally, they showed that Concuerror's use of partial-order reduction techniques allows it to effectively explore the universe of potential thread interleavings.

In summary, the Concuerror linearizability checker provides a methodical and effective means of identifying concurrency defects, such as violations of linearizability, in concurrent Erlang systems. Concuerror lessens the work necessary to guarantee the accuracy of concurrent Erlang programs by combining methodical investigation, issue identification, and test case generation.

When it comes to the time skew problem, one influential paper is "Time, Clocks, and the Ordering of Events in a Distributed System" by Leslie Lamport. This paper introduces the concept of logical clocks, which are used to order events in a distributed system, even when the clocks on different nodes are not synchronized. The paper also introduces the happens-before relation, which is a partial ordering of events that can be used to reason about causality in a distributed system.

Another significant paper is "The Google File System" by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. This paper describes the design of the Google File System (GFS), which is a distributed file system used by Google. The paper discusses how GFS handles time skew by using a master node to coordinate access to the file system, and by replicating data across multiple nodes to provide fault tolerance.

Finally, "Time, Clocks, and the Ordering of Events in a Distributed System: Revised" by Leslie Lamport provides an updated version of the original paper. This revised paper includes additional material on clock synchronization algorithms, which can be used to reduce time skew in a distributed

system. The paper also includes a discussion of the challenges involved in designing clock synchronization algorithms that are both accurate and efficient.

1.5 Assumptions and Limitations

This study assumes that the problem of time skew, one of the most crucial challenges of distributed systems, has already been solved with one of the existing clock synchronization methods. In a distributed system, time skew can occur due to various factors, such as:

- Clock drift: The clocks in different nodes of the system can have different rates of drift, leading to time differences between them.
- Network latency: The time it takes for a message to travel between nodes in a distributed system can vary depending on the network conditions, leading to variations in the timestamps of the messages.
- Message queuing delays: Messages can be queued in buffers in different nodes before being processed, leading to variations in message delivery times.
- Clock synchronization errors: Even if clocks are synchronized, there can be errors in the synchronization process that can lead to time skew.

Time skew in a distributed system can lead to a range of problems. For instance, inconsistencies in data may arise if nodes in the system employ timestamps to order events or data. Additionally, coordination issues may emerge if nodes need to coordinate their actions based on timestamps or timeouts. In some cases, time skew may also create security vulnerabilities, especially if timestamps are used for security purposes like preventing replay attacks or enforcing time-based access control.

To mitigate time skew, various clock synchronization algorithms can be used, as mentioned earlier. These algorithms ensure that the clocks in different nodes are synchronized within a certain error bound, reducing the likelihood of time skew. In addition to clock synchronization, other techniques such as message ordering and queuing can be used to further reduce the impact of time skew in a distributed system.

Synchronizing clocks is crucial for maintaining coherence and collaboration between various nodes in a network. In distributed systems, each node possesses a clock used for marking events like message transmissions, updates to databases, and user interactions. However, due to factors like clock drift, delays in the network, and time spent on processing, these clocks might lose sync over time, resulting in system inconsistencies and errors. For example, if a pair of nodes in a distributed system display differing clock values, their interpretation of the event sequence might also differ, potentially causing conflicting outcomes. Furthermore, if a node transmits a message bearing an inaccurate timestamp, the recipient node could dismiss the message as it appears improperly sequenced. Clock synchronization is especially vital in applications where time is of the essence, such as financial trading, telecommunication networks, and power grid systems, where even minor timing differences can have significant ramifications.

To avoid these issues, clock synchronization methods are used to ensure that all nodes in the system have a consistent and accurate view of time. This enables them to coordinate their actions, maintain data consistency, and evade conflicts. Here are some of the most common methods and their comparison:

- **NTP (Network Time Protocol)**: NTP is a widely used protocol for clock synchronization that uses a hierarchical approach to synchronize clocks across a network. It operates using a set of reference clocks that are highly accurate and accessible over the internet. The protocol uses a complex algorithm that compensates for network delays and adjusts the clock's frequency to match the reference clocks. NTP can achieve sub-millisecond synchronization accuracy and is widely used in critical applications such as financial trading, telecommunication networks, and power grid systems. One of the key assumptions of NTP is that the network delay is constant, which may not hold true in all scenarios.
- **PTP (Precision Time Protocol)**: PTP is a network-compatible high-precision clock synchronization protocol. PTP aims to achieve higher accuracy in clock synchronization across diverse devices in a network and was created to solve the shortcomings of other synchronization techniques, such as the Network Time Protocol (NTP). It is extensively utilized in control systems, scientific experimentation, industrial automation, and other applications that need for sub-microsecond synchronization accuracy. With a master-slave architecture, PTP synchronizes a selected master clock with slave clocks dispersed throughout the network. PTP is able to effectively maintain clock synchronization in massive systems thanks to this hierarchical structure. The use of hardware-based timestamping, which reduces network latency and enhances PTP's remarkable synchronization accuracy, is one of its distinguishing characteristics. This hardware-assisted method guarantees that the impact of elements like network jitter and delay is minimized, resulting in more accurate and dependable timekeeping. PTP has remarkable features, but it also makes several assumptions that might not always be accurate. One of these is that the network delay is symmetric and constant. This implies that a message should always arrive at its destination after traveling from the master clock to a slave clock. In actual use, network conditions can change, causing shifts in network latency and possible errors in time synchronization. However, in order to lessen the effects of network delay changes and preserve high synchronization accuracy, PTP uses a variety of strategies, such as delay request-response systems.
- **GPS (Global Positioning System)**: The United States Department of Defense created and maintains the satellite-based navigation system known as the Global Positioning System (GPS). A GPS receiver can normally acquire signals from at least four satellites at any given moment because to the network's positioning of 24 to 32 satellites in Earth's orbit. The GPS can deliver extremely accurate time and location information all around the world because to the wide-ranging satellite coverage. Atomic clocks, which are extraordinarily accurate timekeepers, are installed in GPS satellites. The time signals produced by these clocks are transmitted to GPS receivers on Earth. GPS receivers are able to pinpoint their location and time with astounding accuracy by triangulating the time signals they receive from many satellites. For many applications needing high degrees of precision, sub-nanosecond synchronization accuracy is essential, and GPS is capable of delivering it. GPS is widely used in a number of vital sectors, including:
- Scientific research: Experiments that require precise time synchronization, such those in radio astronomy or particle physics, are time-sensitive and greatly benefit from the use of GPS.

- Military operations: GPS is essential to systems for navigation, reconnaissance, and targeting, guaranteeing that military personnel and equipment can function effectively and precisely.
- Telecommunications: The synchronization of communication networks and upkeep of precise timing for a variety of services, such as cellular networks, internet services, and satellite communications, depend on the GPS time signals.Geolocation and navigation: GPS is used often in common applications, such as mobile phones, personal navigation devices, and car navigation systems, enabling users to find their current location, map out routes, and keep track of their movements.Emergency services: GPS technology makes it easier for emergency responders, including police, firefighters, and paramedics, to find and identify persons in need.The GPS receiver must have a clear view of the sky and be able to receive signals from at least four satellites in order for the system to work effectively. Tall structures, mountains, and dense vegetation are examples of obstacles that may potentially interfere with the signal reception and impair accuracy. Additionally, the functioning of the GPS system can be impacted by atmospheric factors like ionospheric disturbances or severe weather, which can impair GPS transmissions.
- **Christian's algorithm**: In a distributed system, Christian's algorithm is a crucial clock synchronization method used to determine and rectify the offset between two nodes' clocks. The measurement of time differences between the nodes, which enables the estimate of clock offsets and subsequent modifications, is the basis of this algorithm. One node, referred to as the client, sends a time request message to another node, referred to as the server, in order to carry out the synchronization. The server responds with its current timestamp after receiving the request. The round-trip delay, which is measured by the client, is the length of time it took for the server to respond. The client can calculate the server's time at the time the response arrives by assuming that the network latency is constant and symmetric, and then it can change its local clock accordingly. The accuracy of synchronization that Christian's algorithm can achieve at the millisecond level makes it a practical solution for many distributed systems that need reasonably accurate timekeeping. This algorithm, however, has some drawbacks and weaknesses. Christian's approach is sensitive to network delays and clock drift since these elements might cause inaccuracies in the server's time estimation and the ensuing clock modifications. The network delay must be constant and symmetric, which means that the amount of time it takes for a message to travel from the client to the server and back must not change. Additionally, the clocks of both nodes must drift at a constant rate. These two conditions are prerequisites for the algorithm to work effectively. However, in practice, network conditions can change and clock drift can be unpredictable, which can result in errors in the time synchronization. Christian's algorithm continues to be a useful technique for clock synchronization in distributed systems despite its drawbacks, especially in situations where the network latency and clock drift are both relatively stable. Alternative methods, such as the Berkeley algorithm or the Network Time Protocol (NTP), may be better appropriate for systems that need higher degrees of synchronization precision or when network delays and clock drift are more unpredictable.

- **Berkeley algorithm**: The Berkeley algorithm is a centralized clock synchronization technique created to use a master clock to synchronize the clocks of all nodes in a network. This technique is particularly good at synchronization precision of less than one millisecond, which makes it useful for low-latency applications like streaming multimedia and online gaming. The Berkeley algorithm's central component is a master clock, which manages the network's nodes' synchronization activities. By gathering clock readings from all involved nodes, the master clock starts the synchronization process. The average time for the entire network is then calculated using these readings. The master clock determines the time difference between the average time and each individual node's clock reading after calculating the average time. Each node is then sent adjustment values, which are used to alter their clocks. This procedure aids in reducing the differences in all nodes' clocks, ensuring a high level of synchronization across the network. The efficacy of the Berkeley algorithm depends on a few fundamental presumptions. It starts off by assuming that the network delay is symmetric and constant. This indicates that the amount of time it takes for messages to travel in both directions between the master clock and the nodes is constant and equal. The algorithm also counts on a constant rate of clock drift. The method can precisely synchronize the clocks inside the network by taking these considerations into account. The Berkeley algorithm has numerous drawbacks despite its advantages. For instance, because it depends on a centralized master clock, it is susceptible to single points of failure. The algorithm's performance may also be affected by unpredictable clock drift rates or fluctuating network latency, which could result in errors throughout the synchronization process.

The choice of the method depends on the specific requirements of the application, such as the desired synchronization accuracy, network topology, and latency constraints. NTP and PTP are commonly used in industrial and critical applications that require high accuracy, while GPS is used in applications that require extreme accuracy. Christian's algorithm and Berkeley algorithm are simpler and less accurate but can be suitable for low-latency applications.

A limitation of the implementation presented on this paper could be that it’s designed for NoSQL databases. Also, the current implementation is compatible with Redis database only, but it can be easily extended. Traditional SQL databases are typically designed to run on a single server, with all the data stored in a single machine. As a result, they usually are not good candidates for scaling out horizontally (adding more servers to the database cluster) as it can be challenging and expensive due to several factors. Firstly, SQL databases are designed to maintain strong data consistency, which means that every transaction must be executed on a single server to ensure that all data is in sync. This makes it difficult to distribute the data across multiple servers without sacrificing consistency. Secondly, SQL databases often rely on a single server to provide high availability. If the server goes down, the entire database becomes unavailable. Scaling out horizontally requires additional infrastructure to ensure high availability, which can add complexity and cost. Lastly, SQL databases are designed to handle complex data relationships, such as foreign key constraints, which can make it difficult to split data across multiple servers. In contrast, NoSQL databases are designed to scale out horizontally from the start. They use a distributed architecture that allows data to be partitioned across multiple servers, making it easier to handle large volumes of data and high write loads.

NoSQL databases also typically sacrifice some degree of data consistency to achieve better scalability and availability, using techniques like eventual consistency to ensure that data is eventually consistent across all servers. NoSQL databases are often a better fit for applications that require high scalability and availability. They are used in applications such as social media platforms, e-commerce, and gaming. In contrast, traditional SQL databases are better suited for applications that require strong data consistency and complex data relationships, such as financial and ERP systems.

2 LITERATURE REVIEW

These days, numerous benchmarking tools have been designed to measure the consistency offered by the data stores in some way. YCSB++ [18] is an example of such systems. The work from Anderson et al., which similarly uses operation traces and runs offline checkers against them, is the most similar to ours. Their inspectors look for violations of atomicity, regularity, and safety. Regularity and read-after-write consistency are analogous to linearizability and atomicity, respectively. Safety is a novel relaxation of read-after-write consistency that allows an arbitrary value to be returned when there are concurrent writes. Every single one of these benchmarking programs creates a synthetic workload, gathers a global trace, and then assesses irregularities in that synthetic global trace. This research, on the other hand, looks at real, sampled traces. We can examine a large-scale industrial system by using sampling traces. We gain insights into the anomalies in the current eventually consistent system and the advantages of stronger consistency by using real traces. In a 24-hour period, Amazon tracked the frequency with which its finally consistent Dynamo system returned multiple (concurrently written) versions of a shopping cart and discovered that 99.94% of queries were met with a single version. Because Dynamo employs sloppy quorums for writes that might not always intersect, several versions are conceivable. Facebook avoids this kind of divergence by having a single master per shard that serializes all updates. Instead of focusing on divergence, our approach examines an alternative component of eventual consistency by examining breaches of numerous consistency models. Expected constraints on staleness are provided by Probabilistically Bounded Staleness (PBS) for replicated storage that employs sloppy quorums in the Dynamo fashion. It employs replication delays to parameterize a model and use those results to forecast how frequently and how stale reads would return values. The study was partially motivated by PBS, which only considers PBS k-staleness and PBS monotonic reads, is based on synthetic models, and is restricted to sloppy quorums. It examines a production single-master-per shared system, is based on actual measurements, and considers a wide range of theoretical and real-world consistency models. Several industry journals have described the consistency model that their systems offer. They provided us with information, and we measured anomalies using most of these models. The model that Facebook's system follows internationally is eventual consistency, which is provided by Amazon's Dynamo Spanner from Google offers rigorous serializability. As a special case of rigorous serializability without transactions that offers a lower bound, we measure anomalies under linearizability. Read-after-write consistency is provided by Facebook's Tao system and per-object sequential consistency is provided by Yahoo's PNUTS system. (Also called per-record timeline consistency). Both models' abnormalities are measured.

Consistency is a critical aspect of large-scale storage systems, as it determines how predictable a system's behavior is, particularly in the presence of concurrency and failures. Linearizability, as stated by Herlihy and Wing, represents the strongest form of consistency in a concurrent system [3]. Nevertheless, implementing linearizability in large-scale systems can be challenging due to performance trade-offs, leading many production systems to opt for reduced consistency levels that offer low latency and high throughput [1]. This decision impacts programming complexity, as stronger consistency can prevent anomalies, or unexpected behavior observable to users.

Lu et al.'s [1] study on Facebook's TAO system is a pioneering investigation into the frequency of anomalies in large, production-scale web services, focusing on the impact of consistency levels on real-world system behavior. The research demonstrates that a linearizable system would produce different outcomes for 0.0004% of reads to vertices, highlighting the advantages of stronger consistency in preventing anomalies. This work stands out among other studies, as it provides valuable insights into the occurrence of anomalies in real-world systems and the advantages of stronger consistency. The study also introduces a consistency monitoring method that records ε-consistency, a novel consistency metric that is effective for tracking system health.

On the other hand, Lee et al. [3] concentrate on the implementation of linearizability in large-scale systems, specifically in the RAMCloud storage system. Their work introduces RIFL, a method that enables easy transformation of non-linearizable operations into linearizable ones, which is designed for large systems and low-latency settings. By implementing RIFL in RAMCloud, basic operations like writes and atomic increments become linearizable, with RIFL only adding 530 ns to the 13.5 s base latency for durable writes. Furthermore, Lee et al. developed a new multi-object transaction mechanism for RAMCloud using RIFL, simplifying the transaction implementation and surpassing the main-memory H-Store database system for the TPC-C benchmark, committing straightforward distributed transactions in under 20 milliseconds. By comparing the practical research by Lu et al. [1] and the implementation-focused work by Lee et al., it becomes evident that both studies contribute to the understanding of the implications of consistency choices in large-scale storage systems from different perspectives.

Ajoux et al. [2] present a broader view of the challenges associated with adopting stronger consistency at scale. They identify obstacles that prevent the deployment of causally or strongly consistent data models, providing context on the difficulties of integrating consistency across numerous stateful services, addressing high query amplification, accommodating key items, and offering net benefits to consumers. This work complements the research by Lee et al. [3], which focuses on technical solutions for achieving linearizability in large systems. Ajoux et al. highlight the practical issues that hinder the adoption of stronger consistency in real-world systems, contributing to the overall discussion on consistency trade-offs in large-scale storage systems. The challenges highlighted by Ajoux et al. also provide valuable context for understanding the significance of the monitoring method proposed by Lu et al., as well as the practical implications of the RIFL method introduced by Lee et al.

Guerraoui and Ruppert [4] disagree with the popular belief that linearizability is always a safety property, offering a different point of view on the topic of linearizability. They demonstrate that linearizability is not a safety property for objects with infinite nondeterminism, contributing to the

overall discussion on consistency and its implications in concurrent systems. This work adds depth to the understanding of the nature of linearizability, complementing the practical research by Lu et al. [1] and the implementation-focused work by Lee et al. [3].

Horn and Kroening [5] expand on the foundational principles of linearizability by introducing the concept of P-compositionality, which extends the application of Herlihy and Wing's localization principle to operations on the same concurrent data type. By developing a novel linearizability checker, Horn and Kroening provide an efficient method for verifying the correctness and consistency of concurrent data types. Their work complements the research by Lee et al. [3] on implementing linearizability in large-scale storage systems and Lu et al.'s [1] exploration of anomaly occurrences in practice.

Linearizability is a specific form of thread safety where all operations of a concurrent component appear to take effect instantly at some point between their call and return [6]. Burckhardt et al. delve into linearizability in concurrent application development, stressing the need for thread-safe components that can function properly when called simultaneously by multiple client threads. They introduce Line-Up, a complete and automatic linearizability checker for deterministic components. This tool is based on the idea that by systematically listing all sequential behaviors of a component and determining whether each concurrent behavior is equivalent to a specific sequential behavior, an automatic linearizability checker can be built. Line-Up does not require manual abstraction, semantic specification, test suites, or source code access, making it an efficient tool for verifying linearizability.

Liu et al. [7] propose another approach for verifying linearizability based on refinement relations from abstract specifications to concrete implementations. Their technique can use linearization points when provided and has been integrated into the PAT model checker to verify multiple concurrent object implementations, including the first scalable nonzero indicator algorithm. To address the state space explosion problem, they develop and apply symmetry reduction, dynamic partial order reduction, and a combination of the two for refinement checking.

In the context of distributed storage systems, Viotti and Vukolić [9] provide a comprehensive survey of more than 50 consistency notions, ranging from linearizability to eventual and weak consistency. They offer a partial ranking of these consistency predicates and map their semantics to real-world applications and research prototypes. Their work focuses on non-transactional semantics, which apply to actions on single storage objects, and complements existing research on transactional database semantics.

Singh et al. [8] address the challenge of achieving high availability in distributed services, which often rely on replication and are housed in large, shared, geographically different data centers. They propose Zeno, an eventually consistent Byzantine-fault tolerance protocol that trades consistency for higher availability. Zeno prioritizes liveness over strong safety guarantees, offering eventual consistency instead of linearizability. Their evaluation of the Zeno prototype demonstrates improved availability compared to conventional BFT protocols.

Van Renesse and Schneider [10] introduce the chain replication protocol for large-scale storage services, which provides high throughput, availability, and strong consistency guarantees. The protocol assumes fail-stop servers that are linearly ordered to form a chain. Query and update

requests are handled serially at a single server (the tail), ensuring strong consistency. A master service is employed to detect server failures and reconfigure the chain accordingly.

Terrace and Freedman [11] present CRAQ, an improvement on chain replication that maintains strong consistency while enhancing read throughput. CRAQ achieves this by distributing load across all object replicas and supporting a weaker consistency model. The system scales linearly with the chain size, and for read-mostly workloads, most read requests are handled solely by non-tail nodes. In write-heavy workloads, version queries, which are lighter-weight than full reads, allow the tail to process requests at a higher rate before saturation.

In summary, the reviewed literature presents various approaches to verifying and ensuring consistency, particularly linearizability, in concurrent and distributed systems. These approaches include automatic linearizability checkers, optimized refinement checking, and chain replication protocols. Researchers have also conducted extensive surveys, explored trade-offs between consistency models, and developed innovative fault tolerance protocols to address the challenges associated with maintaining consistency in distributed systems.

3 METHODOLOGY

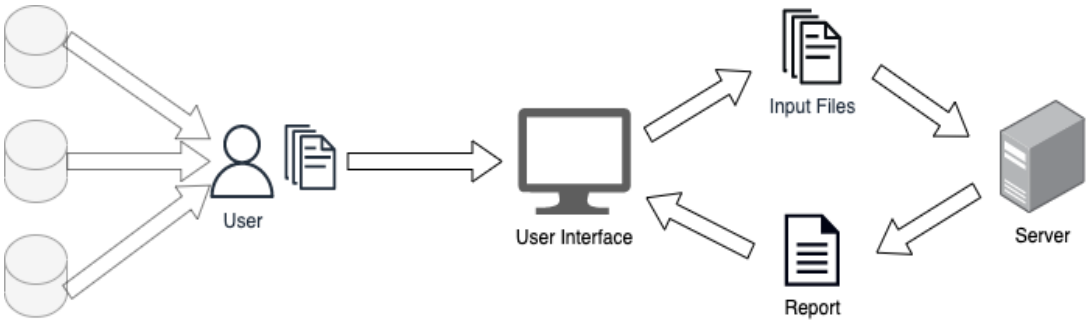

Figure 2: Architecture of the System

The system's operation commences when the user submits query history files obtained from multiple database instances. To arrange the queries in sequential order by their execution date, the files will be merged into a single history file. There is no restriction on the number or size of the input files. Given that the input files may contain confidential data, the upload process must be exceptionally secure. After the user uploads the input files, the system verifies them for any consistency violations and generates a report that is displayed to the user. The report explicitly highlights the locations where consistency violations, if any, have occurred.

The system is composed of two subsystems (see Figure 2), namely the User Interface and the Server. The former is the component that operates on the user's machine, and it is responsible for collecting the input files from the user, transferring them to the server in a secure manner, and displaying the server-generated report to the user. Meanwhile, the Server is the component that undertakes computation-intensive procedures. It oversees merging the input files that were provided, verifying their strong consistency, creating a report, and returning it to the User Interface. One of the available options for designing the system was to exclude the Server component. Although this

would mitigate some security issues such as secure file transmission, it would necessitate a significant amount of processing power on the client machine, which may not always be feasible.

The format of the input files, which contain the records of executed queries on database instances, is as follows:

```
<Database Instance ID>

<Timestamp> || <Query> || <Result>

<Timestamp> || <Query> || <Result>

…
```

You can also see an example input below:

```
<redis-03>

2022-10-14T22:11:18Z || SET NAME HASAN || OK

2022-10-14T22:11:20Z || GET NAME || HASAN
```

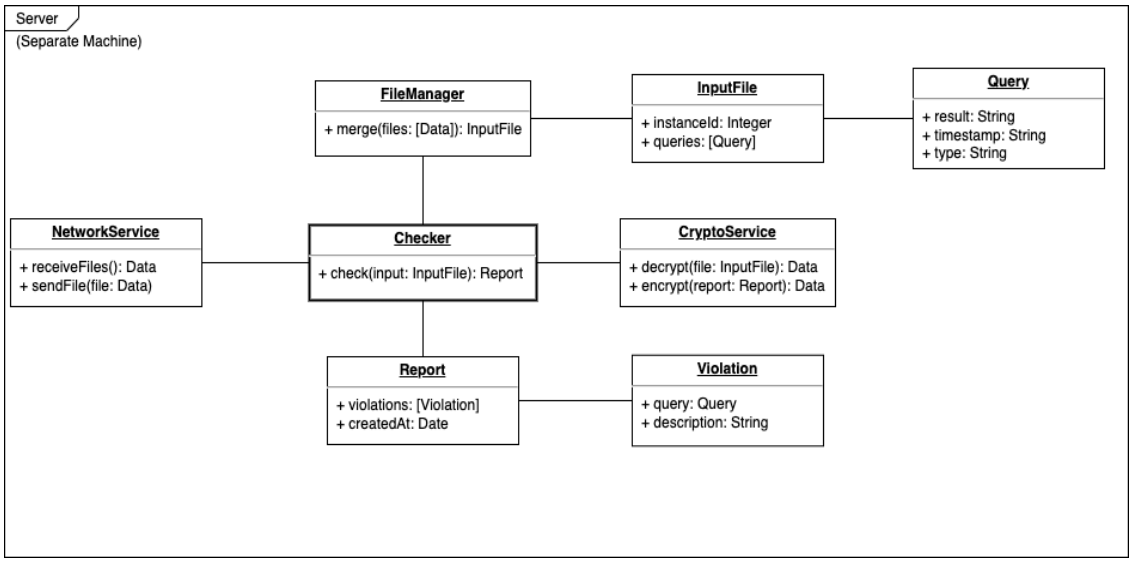

Figure 3: Server Architecture

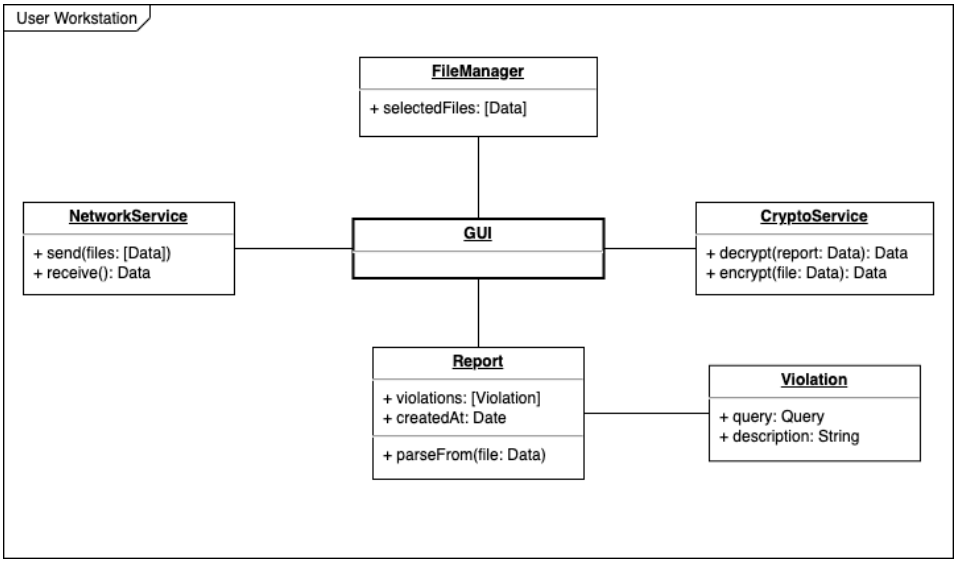

Figure 4: Client Architecture

The main logic of the system, which performs the linearizability check can be simply explained as follows: The system starts by creating an empty HashMap data structure to hold key-value pairs that represent distinct identifiers and the data they correspond to. After that, it reads and parses an input file that contains a string of SET, UPDATE, and GET queries together with their corresponding keys and values.Depending on the type of inquiry, the system takes different steps after processing each query. It uses the put() method to add or modify the key-value pairs in the HashMap for SET and UPDATE queries. When a GET request is made, the HashMap is searched for the supplied key and the corresponding value is either returned, or a designated value is returned if the key cannot be located. The system compares the returned value from the HashMap against the expected value supplied in the input file after processing a GET query. If the values agree, there are likely no breaches of linearizability, and the algorithm moves on to the subsequent inquiry. A linearizability violation has occurred, and the system records it if the values do not agree.When a violation is found, the system adds a thorough description to a report object, detailing the offending query, the intended and actual values, and any additional pertinent information. This report can then be output by the system for additional examination and troubleshooting. The linearizability check system runs a number of queries, looks for any violations of linearizability, and confirms that concurrent data structures uphold the linearizability property. For concurrent systems to remain consistent and correct, this is essential.

Figure 5 offers a visual description of the main logic through an activity diagram.

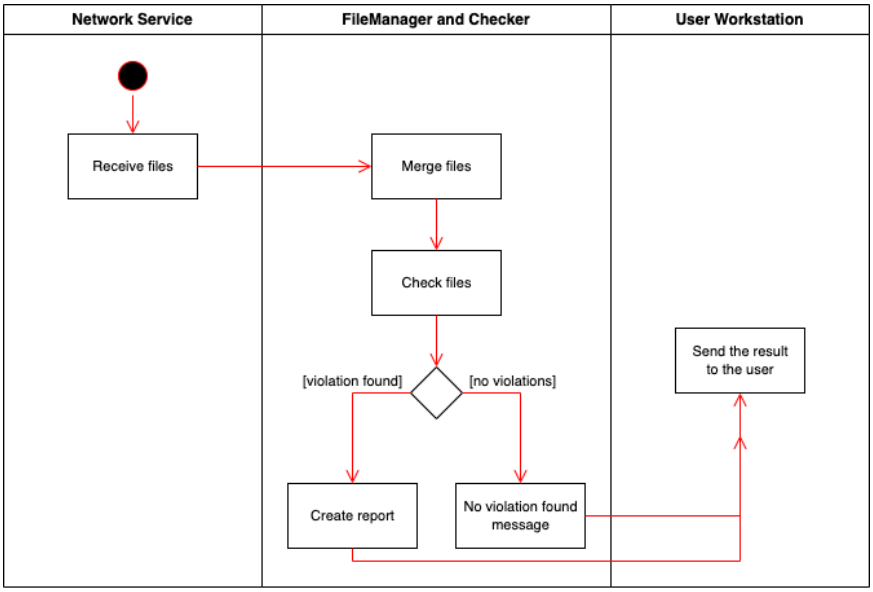

Figure 5: Activity Diagram

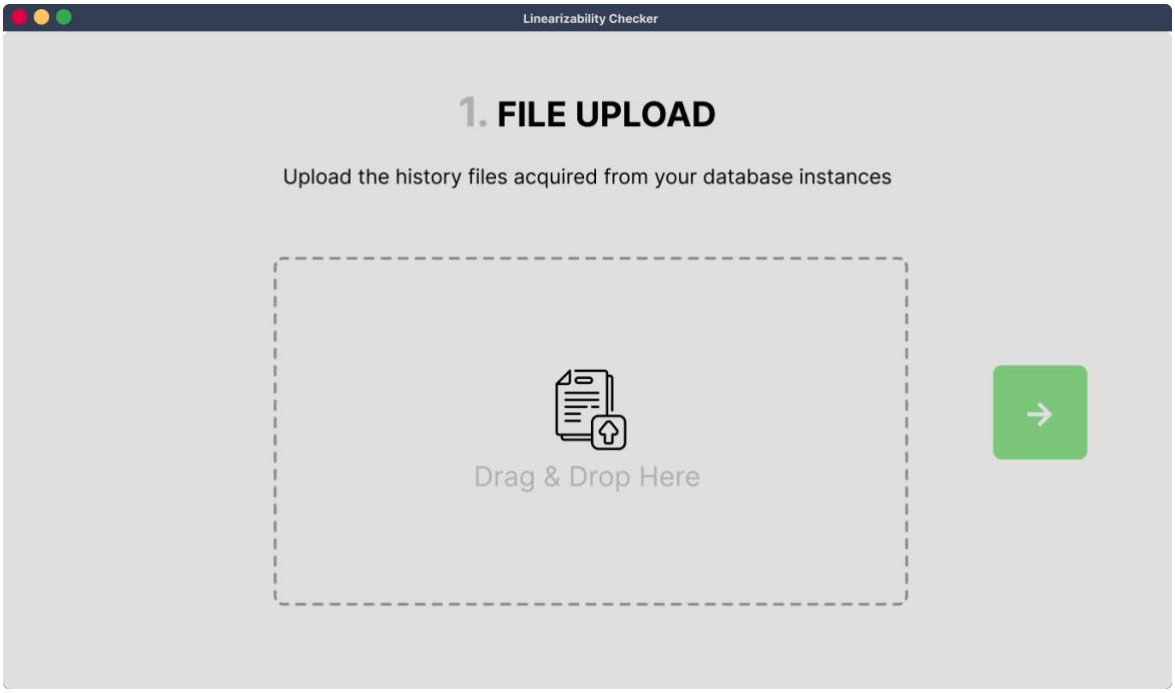

Figure 6: UI design of the first step

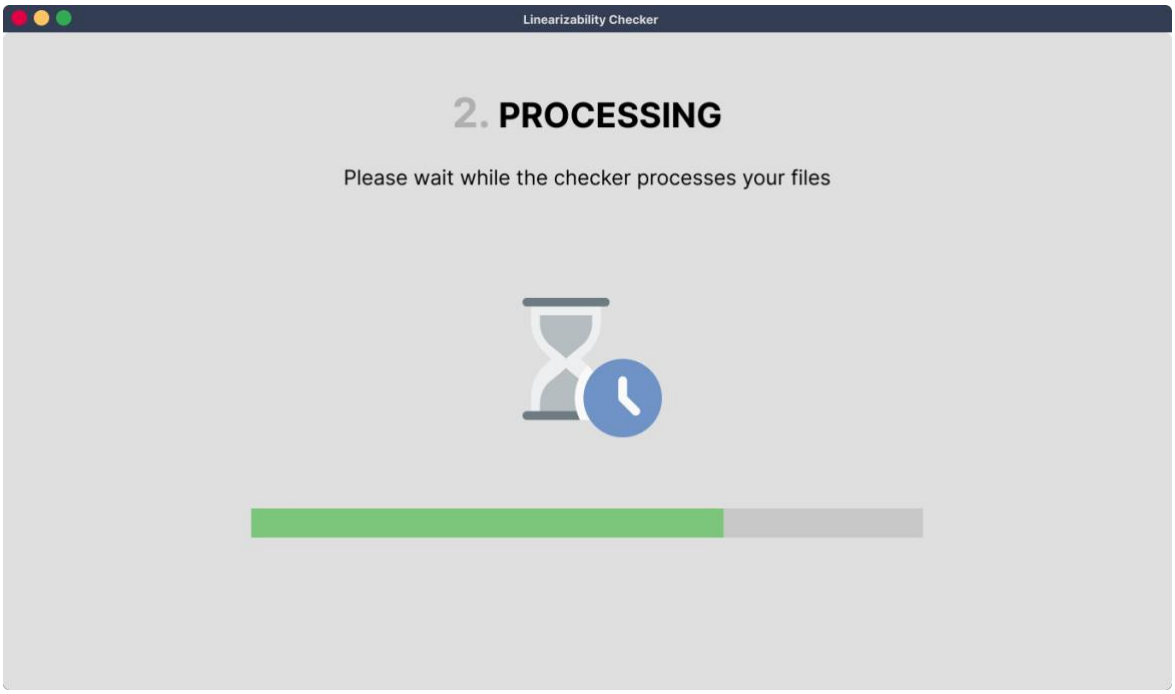

Figure 7: UI design of the second step

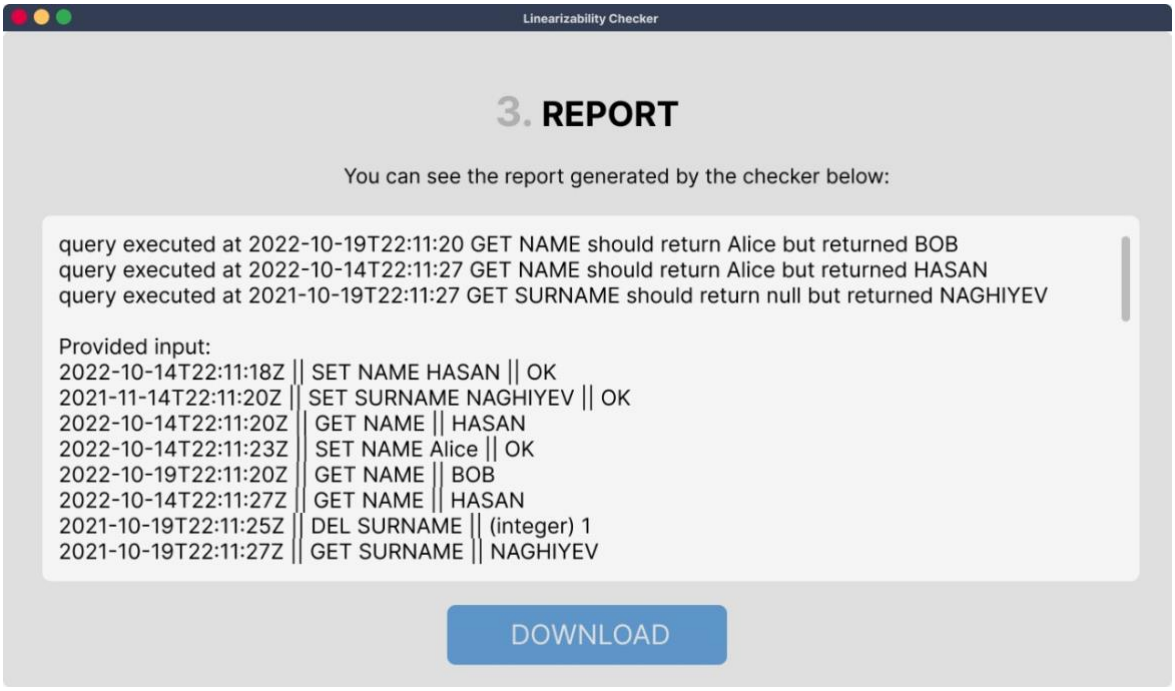

Figure 8: UI design of the third step

The UI that is displayed to the user walks the user through 3 steps:

1. The user is prompted to submit the database history files that have been collected from various instances. (see Figure 6) A green button is used to proceed to the next stage once the upload has been completed.
2. The user is asked to wait while the server processes the files and uploads them (see Figure 7).
3. The program presents the generated report. The user can read the report from the program window and, optionally, download the report (see Figure 8).

The system solves the time skew problem with an implementation of NTP [16], a protocol that is widely used to synchronize clocks in a distributed system. Initially created during the 1980s, it has evolved into a standard for synchronizing clocks on the Internet and within various distributed systems.

The NTP protocol operates in a hierarchical manner, with a few highly accurate servers at the top of the hierarchy providing time to other less accurate servers, and so on. This stratified system guarantees the effective and precise distribution of time throughout the network. To achieve clock synchronization, NTP employs a mix of algorithms and procedures, which include:

1. Reference clock selection: NTP selects one or more reference clocks, which are extremely precise time sources for the other clocks within the network. The choice of reference clocks is based on various criteria, such as accuracy, reliability, and availability. NTP uses a

weighted average algorithm to merge the time from multiple reference clocks and to assign a confidence level to each clock.

2. Clock filtering: NTP uses a filtering algorithm to remove outliers and smooth the clock data. NTP uses a variation of the median filter, which considers the deviation of each sample from the median and removes samples that deviate beyond a certain threshold. This approach helps to reduce the impact of transient errors or fluctuations in the clock data.
3. Clock steering: NTP uses a steering algorithm to adjust the clock frequency and phase to keep the clock synchronized with the reference clock. The phase-locked loop (PLL) technique used by NTP modifies the clock's frequency and phase in response to variations from the reference clock. In order to continuously reduce the temporal difference between the clock and the reference clock, the PLL algorithm employs feedback to change the clock frequency and phase.
4. Clock stratum: The reference clock is located at stratum 0, and NTP divides clocks into strata based on their separation from the reference clock. The quantity of "hops" needed to go to the reference clock establishes the stratum of a clock. A clock at stratum 1 is in direct synchronization with the reference clock, while a clock at stratum 2 is in synchronization with another clock at stratum 1, and so on. To avoid synchronization loops and guarantee that the system's time hierarchy is upheld, NTP makes use of the clock stratum.

These algorithms work together to ensure accurate and reliable clock synchronization in a distributed system. By selecting a suitable reference clock, filtering out outliers, adjusting the clock frequency and phase, and maintaining the time hierarchy, NTP can achieve high levels of accuracy and stability in clock synchronization.

NTP incorporates multiple security measures to defend against threats, including authentication and encryption. To verify the source of the time, it can use either public key infrastructure (PKI) or symmetric key cryptography. Additionally, the protocol provides a number of operational modes, such as broadcast, multicast, and unicast. A client requests time from a server in unicast mode, whereas the server sends time to a group of clients in multicast and broadcast modes.

Numerous distributed systems, such as the Internet, financial networks, and telecommunications networks, now depend heavily on NTP . The precision and reliability it offers have rendered it an indispensable instrument for guaranteeing well-coordinated and synchronized operations within these systems.

For proper evaluation of the system, we first came up with a test environment and plan. The test environment includes physical resources, production data, and users. To determine the required physical resources and data volume, developers were surveyed to get a better insight into the required amount of data and physical resources (see A.1)

Based on developer responses, the test environment consists of several key components. First, a test server should be established, with Linux being the preferred choice for easier virtualization. Some tests need to be executed on a remote workstation, which requires a test server that can support various applications. Ideally, this server should have 16 GB of RAM and an adequate number of cores or threads for a smooth testing experience.

Then, an appropriate SDK must be installed, with JDK being the choice in this particular case. A database server also needs to be configured. We prefer Redis for our implementation. Developers suggest that a single node should be enough for testing purposes.

It is important to generate test data for the test environment, which typically involves replicating production data. This allows testers to identify issues similar to those on a live production server without jeopardizing production data. The process involves creating scheduled tasks (named cron jobs in UNIX) that replicate data to a shared testing environment while hashing sensitive information, such as salaries and passwords, to avoid compromising the privacy of real users.

Additionally, network configurations should be set up to accept requests from authorized clients. This might involve whitelisting IP or MAC addresses to ensure only testing team members have access to the test environment. Finally, the backend application can be launched on the server and clients can use the front-end to test the program.

4 RESEARCH RESULTS AND ANALYSIS OF RESULTS

We discuss the findings and analyses of a query execution monitoring system built on the Java platform for a master's thesis. In order to simulate a normal database management system, the system reads a database file that contains a list of queries that have been executed. The system's goal is to identify and communicate inconsistencies between anticipated and actual query results. Using Java and its built-in libraries, we created the monitoring system with an emphasis on query log processing, parsing, and analysis. The Query class, ReadFileService class, and utility class LocalDateTimeConverter make up the system's three basic building blocks. A query's execution time, query type (GET, SET, or DEL), and actual result are all represented by the Query class. The ReadFileService class reads and analyzes the input file, finds inconsistencies between anticipated and actual query results, and alerts the user to any violations. Last but not least, the utility class LocalDateTimeConverter transforms dates represented in strings into LocalDateTime objects.Our examination of the monitoring system reveals its potency in spotting anomalies in query execution. The system successfully identified discrepancies in the query results when tested using a sample database file, and these were provided in a clear and plain style. A database management system's integrity and dependability depend on its capacity to recognize and report query execution problems. Additionally, the system's modular structure makes it simple to integrate with other parts, making it useful in bigger database management systems. However, there are some restrictions with the monitoring system that needs to be solved in future studies. First of all, the system currently treats input parsing exceptions or edge situations as if they don't exist and assumes a faultless input file format. The robustness of the system would increase with better error handling and input validation. Second, the system can only currently handle the query types GET, SET, and DEL. The applicability and usefulness of the tracking system in real-world scenarios would expand with support for more query types and database management systems.

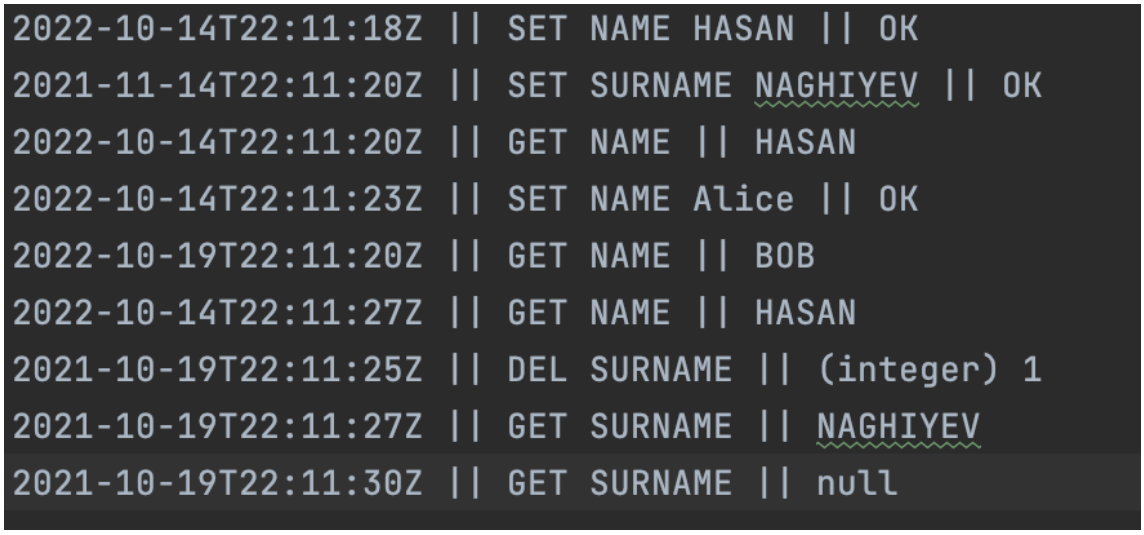

Figure 9: Sample Input File Format

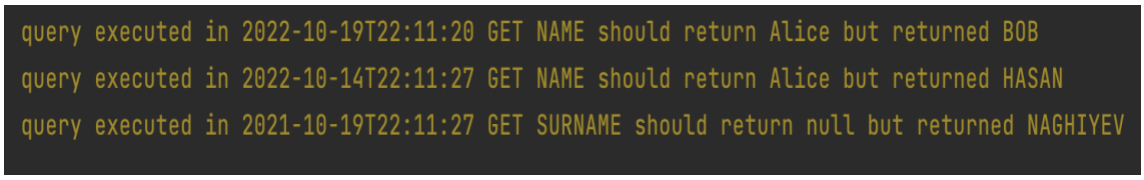

Figure 10: Sample Output File Format

In figure 10, the output of the monitoring system provides a clear representation of detected discrepancies in query execution. It prints each inconsistency on a separate line, using a consistent format to describe the erroneous query, its execution time, the expected result, and the actual result.

In the provided example, the system detects three discrepancies:

1. The first inconsistency is detected when the system encounters a GET query for the key "NAME" at 2022-10-19T22:11:20. The system expects the value "Alice" as the result since the last SET query for the key "NAME" was executed at 2022-10-14T22:11:23 and set the value to "Alice". However, the actual result of the query is "BOB".
2. The second inconsistency occurs with a GET query for the key "NAME" at 2022-10-14T22:11:27. Again, the system expects the value "Alice", but the actual result is "HASAN".
3. The third discrepancy is found in a GET query for the key "SURNAME" at 2021-10-19T22:11:27. The system expects a "null" value since a DEL query for the key "SURNAME" was executed at 2021-10-19T22:11:25, removing the key-value pair from the map. However, the actual result is "NAGHIYEV".

In this paper, we have conducted a comprehensive set of tests to validate the robustness and reliability of the system under various scenarios. These tests were designed to ensure that the system

behaves as expected under both ideal and non-ideal conditions, demonstrating its adaptability and resilience.

1. We first examined the system behavior when a valid database connection was established, and each database instance contained log files. In this scenario, the system successfully processed the log files and detected any inconsistencies as expected, proving its effectiveness in a standard environment.
2. We also tested the system when a valid database connection was established, but one of the instances was missing log files. In this case, the system was able to handle the missing files gracefully, continuing its operation without errors and providing appropriate feedback to the user.
3. The system's behavior was evaluated when an invalid database connection was provided. In this scenario, the system displayed proper error messages, ensuring that users are aware of the issue and can take corrective action.
4. We further tested the system when a valid database connection was established, but each database instance contained log files in different formats that were not considered by the system. In this case, the system was able to detect the unexpected file formats, generate appropriate error messages, and continue processing the remaining files.
5. The system's behavior was also assessed when a valid database connection was established, and the log files were of a significantly large size. In this scenario, the system demonstrated its ability to handle large data volumes without performance degradation or errors, proving its scalability.

Additionally, we performed tests on the input files to evaluate the system's ability to handle various file formats and sizes:

1. We tested the system when all input files were in the expected format and size. In this case, the system performed as intended, processing the files without any issues.
2. We evaluated the system behavior when all input files were in the expected format but had unexpected sizes. In this scenario, the system demonstrated its resilience by adapting to the varying file sizes and continuing its operation without errors.
3. The system was tested when all input files were in different formats but had the expected size. In this case, the system was able to detect the unsupported file formats, generate appropriate error messages, and continue processing the supported files.
4. We assessed the system behavior when all input files were in different formats and unexpected sizes. In this scenario, the system demonstrated its adaptability by identifying the unsupported file formats and sizes, providing relevant error messages, and continuing to process the supported files.
5. Finally, we tested the system behavior when input files had reading restrictions. In this case, the system was able to detect the access limitations, generate proper error messages, and continue processing the remaining files without restrictions.

In addition to the previously mentioned tests, we also conducted further tests to assess the system's behavior when handling objects in the list and interacting with the Redis database:

1. We examined the system behavior when all objects in the list were not null and the content of each object was successfully saved to Redis. In this scenario, the system functioned as expected, efficiently saving the objects to the Redis database without any issues.
2. We tested the system behavior when one or more objects in the list were null and the content of each object could not be saved to Redis. In this case, the system was able to identify the null objects, generate appropriate error messages, and continue processing the remaining non-null objects. This demonstrated the system's capability to handle unexpected input data and continue its operation without errors.
3. We evaluated the system behavior when all objects in the list were not null, but a database connection problem occurred while attempting to save the objects to Redis. In this scenario, the system was able to detect the connection issue, provide relevant error messages, and continue its operation. This test showcased the system's resilience in handling connectivity issues with the Redis database.

These tests confirmed the robustness, reliability, and adaptability of the system under a wide range of conditions, demonstrating its potential for practical applications in managing and maintaining distributed database systems.

In this study, synchronizing the times of many database servers proved to be a big challenge. The analysis and validation of query results in the Java-based query execution monitoring system we created strongly depends on precise timekeeping. The application may fail to take into account the differences in server times when comparing predicted and actual query results, which might result in inaccurate results or false violation detections. The timestamps provided in the database files are presumed by our Java code to be correct and reliable on all servers. Maintaining exact time synchronization in a distributed system with several servers is challenging. Servers may encounter internal timekeeping inconsistencies due to clock drift, latency, or other issues. These differences may result in inconsistent query execution logs, which would then cause our monitoring system's analysis to be unreliable.Incorporating a time synchronization system like NTP into our Java code is one way to solve this problem. The influence of time differences on the analysis process can be reduced by adding NTP to ensure that the timestamps in the database files are synced to a common time reference. Our monitoring system can more reliably compare query results and precisely identify infractions with a single time base. Furthermore, we can add more error-checking and validation tools to the Java code to increase the resilience of our system. With the use of these techniques, the analysis procedure can be modified in order to account for potential time differences or inconsistencies between servers. For instance, the system could alter the anticipated query results to take into account known time differences between servers or use statistical methods to estimate potential inconsistencies. We can greatly enhance the precision and dependability of our Java-based query execution monitoring system by tackling the issue of time synchronization between database servers. In turn, this will make it possible to spot actual violations, guarantee the accuracy of the data, and improve the distributed system's overall performance. In order to correctly and consistently synchronize time among the nodes in this cluster of three, we developed Network Time Protocol (NTP). The task of synchronizing each node's time with a trustworthy global time source, such as an atomic clock or GPS time signal, was assigned to one node, known as the master. The other two

nodes were set up as clients and used the master node to synchronize their clocks. In order to guarantee data integrity and coordinate events throughout the distributed system, a consistent time basis must be maintained by all nodes in the cluster. This hierarchical technique makes sure of this. Using NTP in this way provides several advantages. First off, because the master node receives its time directly from a very accurate global source, it offers a high level of accuracy in time synchronization. This guarantees that the time foundation for the entire cluster is precise and trustworthy, which is necessary for time-sensitive operations and maintaining a consistent understanding of the system's occurrences. Second, because only the master node needs to be configured to connect to an external time source, the hierarchical structure makes it easier to manage and maintain time synchronization inside the cluster. The system's overall complexity can be decreased by having the client nodes just rely on the master node for time synchronization. In a distributed system like this, using NTP also helps to avoid problems that could arise from clock drift or time inconsistencies between nodes. As a result of inconsistent data and event processing caused by clock drift, the system's dependability and performance may suffer. The system makes sure that all nodes stay in sync and work with a single, precise time basis by implementing NTP and appointing a master node to synchronize with a global time source. This significantly improves the distributed system's robustness and dependability, enabling it to execute time-sensitive operations and keep data integrity across the cluster.

To configure NTP in the three-node cluster, we followed several steps to ensure accurate time synchronization between the master and client nodes. The detailed steps are as follows:

1. NTP software installation: On all three nodes, the NTP software package was installed using the proper package manager for the operating system (e.g., apt-get for Debian-based systems or yum for RHEL-based systems). Each node was given the equipment needed to take part in the NTP synchronization procedure in this step.
2. Master node configuration: On the master node, we made changes to the NTP configuration file, which is normally found at /etc/ntp.conf. We made sure that the master node synchronizes its time with trustworthy external sources by adding the global NTP server pool addresses or specific NTP server addresses using the server directive to the configuration file. Additionally, we added the broadcast directive followed by the local network's subnet address to the master node to designate it as a local NTP server.
3. Client node configuration: We modified the NTP configuration files on the client nodes. We added the IP address or hostname of the master node using the server directive in place of specifying external time sources. The client nodes synchronize their time with the master node thanks to this setting.
4. Firewall configuration: We modified the firewall settings on each node to permit NTP traffic between them. To enable NTP communication between the master and client nodes, we opened the required port (UDP port 123 by default).
5. Starting and enabling the NTP service: Following configuration, we launched the NTP service on each node using the proper system command (such as systemctl start ntp or service ntp start). We also set the NTP service to launch automatically at system startup, guaranteeing ongoing time synchronization even after a reboot.

6. Time synchronization verification: Using the ntpq -p command, we examined each node's synchronization status to ensure that NTP had been successfully configured. This command shows a list of NTP peers together with information about their synchronization, enabling us to confirm that the client nodes are synchronizing with the master node and that the master node is synchronizing with the external time sources.

These steps allowed us to successfully configure NTP on the three-node cluster, with the client nodes synchronizing their time with the master node and one master node synchronizing its time with external time sources. By ensuring accurate and constant timekeeping throughout the whole cluster, this procedure improved the distributed system's dependability and data integrity. In conclusion, the Java-based query execution monitoring system shown in this Master's thesis demonstrates its value in identifying and disclosing query execution irregularities. The modular design of the system and simplicity of interface with other components make it a helpful tool for confirming the dependability and integrity of database management systems. Future development should address the system's flaws, as well as its strengths and usefulness in in real-world circumstances.

5 SUMMARY AND FUTURE WORK

In this paper, we presented a program that is limited to working with Redis database files. Nevertheless, it can be extended to support more databases. The current implementation of the app already converts the files acquired from Redis database to a generic format before analyzing them. This conversion can be performed for any other database file as well, using various methods, such as the Adapter pattern. The Adapter pattern allows two incompatible interfaces to work together by creating an adapter class that implements the Redis file structure and exposes an interface that is compatible with the program. This adapter class can then be used to connect to other types of databases or data sources. By using this pattern, we can create a consistent interface between the program and different data sources, making it easier to manage and maintain. Furthermore, using the Adapter pattern improves the program's maintainability. The program's code doesn’t need to be changed every time to accommodate different data sources. As new data sources become available, we can create new adapters without modifying the existing code. This means that the program's core logic remains stable and does not require significant modifications to accommodate new data sources. For example, if we wanted to connect the program to a MongoDB database, we could create an adapter class that maps the MongoDB data to a generic structure that can be interpreted by the system. This adapter class would then be used by the program to access the MongoDB data. Additionally, since each adapter can be developed independently, it will be easier to test and maintain the overall system. While the Adapter pattern is a suitable solution to extend program for different types of databases, it is not the only one. Several other patterns and approaches can be used to achieve the same goal. Furthermore, the P-compensation technique with heuristics can be used to enhance our existing approach, which depends on NTP for time synchronization. By resolving the shortcomings of NTP and enhancing the overall performance and reliability of the distributed system, this algorithm seeks to offer a more reliable and precise time synchronization solution.

The N-complete problem arises when a single-time synchronization mechanism tries to synchronize every node in a network, increasing latency and decreasing accuracy. The P-compensation technique is made to function in large-scale distributed systems which helps in avoiding this issue. The nodes in the system are arranged into a hierarchical structure using P-compensation, with each level of the hierarchy in charge of synchronizing a particular subset of nodes. This group contributes to the workload distribution for time synchronization, which lowers overall latency and improves time synchronization accuracy across the network. Heuristics and the P-compensation method work together to better enhance the time synchronization procedure. When choosing the ideal parent nodes for each level of the hierarchy or modifying the synchronization intervals based on the state of the network, heuristics are utilized to make informed selections. The P-compensation method may react to changes in the network environment by adding heuristics, ensuring that time synchronization stays precise and effective despite network congestion or other difficulties.

By implementing the P-compensation algorithm with heuristics in our distributed Redis system, we can achieve several key benefits:

1. Enhanced accuracy: The distributed system's timekeeping is more accurate due to the P-compensation algorithm's assistance in minimizing the effects of network latency on time synchronization.
2. Scalability: The P-compensation algorithm's hierarchical node organization enables effective synchronization in large-scale networks, ensuring that the system can continue to function well as it expands.
3. Adaptability: The P-compensation algorithm may adapt its synchronization method based on the current network conditions thanks to the use of heuristics, ensuring optimum performance in a variety of settings.

In conclusion, the P-compensation method with heuristics may greatly improve temporal synchronization in our distributed Redis system, leading to a more dependable and high-performing system. This method addresses NTP's shortcomings and offers a more scalable and flexible time synchronization solution in distributed applications.Our distributed Redis system will incorporate a real-time inconsistency detection and correction technique as another topic of future effort. We can guarantee that the system maintains a high level of data consistency and dependability, even in a dynamic and distributed setting, by continually checking the system for discrepancies and updating the database with accurate values as necessary.

To achieve this goal, we will adopt the following strategies:

1. **Real-time monitoring**: Set up a monitoring module to track the distributed Redis system's state continually and spot any discrepancies across the nodes. To find discrepancies and potential conflicts in the system, this module can use a variety of strategies, such as comparing the data and timestamps of various nodes.
2. **Conflict resolution**: Create a system that automatically resolves any inconsistencies that are found. This can be done by employing a set of pre-established guidelines or heuristics that assess the appropriate course of action depending on the unique characteristics of the

discrepancy. To determine which value should be accepted as correct, the system may, for instance, use a majority voting approach or give preference to the most recent data.

3. **Database update**: The system will automatically update the impacted nodes to make sure they are storing the proper data after the conflict resolution algorithm has established the correct value. This procedure can be carried out in a coordinated fashion to reduce the effect on system performance as a whole and to stop additional inconsistencies from occurring throughout the update.
4. **Adaptive learning**: Utilize machine learning strategies to enhance the system's capacity to recognize and fix discrepancies over time. The system can discover patterns and trends that point to potential causes of inconsistencies by examining previous data and system behavior. This enables it to take proactive measures to fix these issues before they become serious difficulties.
5. **Fault tolerance and self-healing**: Build fault tolerance and self-healing capabilities into the system to ensure that it can continue to work effectively even in the presence of mistakes or other issues. Implementing redundancy, replication, or backup systems may be necessary to guarantee that the system can quickly recover from any errors or data loss.

We want to develop a distributed Redis system that can sustain high levels of data consistency and dependability in real time by putting these techniques into practice. This strategy will improve the system's overall performance and stability and offer a strong framework for supporting a variety of use cases and applications in distributed environments.

Predictive inconsistency detection, a proactive strategy that seeks to spot prospective discrepancies in a distributed database system before they really arise, would be another future endeavor. This approach uses past data, system behavior, and a variety of analytical tools to predict the possibility of future discrepancies. By spotting prospective problems and resolving them before they have an influence on the system's performance or data integrity, predictive inconsistency detection can increase the dependability and efficiency of a distributed Redis system. Here is a thorough explanation of how to accomplish predictive inconsistency detection:

1. **Data collection and preprocessing**: The distributed Redis system must first be queried for pertinent historical data in order to begin implementing predictive inconsistency detection. This data may also comprise query logs, system metrics, and other performance indicators that offer insight into the system's activity over time. The obtained data can be prepared for further analysis by using data preprocessing techniques such as data cleansing, standardization, and transformation.
2. **Feature extraction and selection**: After the historical data has been preprocessed, pertinent features can be extracted to describe the system's properties that may cause discrepancies to occur. The most important properties that have a strong link with inconsistency occurrences can be found by using feature selection approaches like filter methods, wrapper methods, or embedding methods.
3. **Model development**: With the help of the selected features, machine learning techniques or statistical models can be trained to predict the likelihood of inconsistencies in the distributed Redis system. Depending on the nature of the issue and the data at hand, techniques like

decision trees, support vector machines, neural networks, or time series analysis may be applied.

4. **Model evaluation and validation**: The predictive model should be verified and evaluated using the appropriate performance measures, such as accuracy, precision, recall, or F1-score, after it has been developed.Cross-validation procedures may be employed to ensure the model's generalizability and resilience in numerous scenarios.
5. **Real-time monitoring and prediction**: The distributed Redis system can incorporate the verified predictive model for real-time monitoring and forecasting. The system is capable of collecting and processing data continuously, updating the model as needed, and making forecasts regarding the possibility of future discrepancies.
6. **Proactive inconsistency prevention**: Proactive steps can be done to avoid the occurrence of inconsistencies in the distributed Redis system based on the predictions made by the model. To ensure data consistency and system performance, this may entail changing system configurations, redistributing data or workloads, or implementing other preventative measures.

A distributed Redis system can improve its dependability, effectiveness, and overall performance by integrating predictive inconsistency detection, which allows it to proactively identify and address any errors.

A linearizability checker must be made more scalable in order to keep up with the scale and complexity of contemporary concurrent systems. The checker's capacity to manage larger input files and a greater number of concurrent operations can be enhanced in a number of ways to accomplish this. First, the linearizability checker's efficiency and computational complexity can be greatly improved by optimizing the underlying methods that it uses. When confirming the correctness of complex systems, exploring new techniques or improving current ones might result in quicker processing times and more precise findings. Second, strategies for parallelization can be investigated so that the checker can handle several queries at once. The linearizability checker can share the workload across several processing units by utilizing multi-core processors or distributed computing resources, further enhancing performance and cutting down on the total time needed for the verification process. Third, sophisticated data structures and memory management strategies can be used to lower memory usage. Larger input files and more complicated processes can be supported by the checker thanks to efficient data structures' ability to store and process data more compactly. Furthermore, advanced memory management strategies can reduce memory overhead, ensuring the checker functions effectively even when dealing with large datasets. Fourth, the linearizability checker can be made more effective overall by including adaptive techniques. These tactics can be tailored to the unique properties of the system being tested, optimizing the efficiency of the checker based on the types of operations and data structures involved. This could involve identifying patterns or trends that can be used to speed up the verification process using heuristics or machine learning approaches. The user interface can be improved and given more customization choices so that customers can adapt the linearizability checker to their own requirements, thus increasing scalability. Giving users the option to modify the checker's behavior in accordance with the requirements and constraints of the system being tested can increase performance. As a result, parallelization,

advanced data structures, memory management techniques, adaptive strategies, and user interface enhancements all help a linearizability checker scale more effectively. By taking into account these factors, the checker can more effectively manage the growing complexity and size of contemporary concurrent systems, guaranteeing the accuracy of a larger range of applications.

We talked future works about optimization, predictive inconsistency detection and etc, however, visualization of model is as much important as its features since the client interacts with UI. The user experience can be greatly improved by creating visualization tools for linearizability checks, which can also help engineers better comprehend the complex dynamics of concurrent systems. These tools can provide an intelligible depiction of the order of operations, the ability to visualize the state of data structures across time, and the ability to more quickly spot potential linearizability violations, among other advantages.

Developers should be able to monitor the progress of concurrent operations and their interactions with the data structures using a complete visualization tool. Developers can better grasp the relationships between operations and possible sources of linearizability violations by displaying the execution timeline. This knowledge can aid in locating troublesome spots in the code and direct the debugging procedure. Visualization tools should allow users to view the state of data structures at different points throughout execution in addition to displaying the order of operations. Developers can quickly traverse across various historical eras, watch the development of the data structures, and identify anomalies or inconsistencies by offering an interactive interface. This level of specificity can be very helpful in identifying and fixing linearizability problems. A strong visualization tool should also include capabilities for locating and emphasizing possible violations of linearizability. The tool can help developers find problematic circumstances by using automated analysis approaches, such as pattern recognition or machine learning algorithms, which will save time and effort while debugging. Developers should have the option to alter the way information is presented to suit their needs in order to increase the usefulness of visualization tools. To successfully communicate the information, this may entail altering the timeline's granularity, filtering particular operations or data structures, or selecting from a variety of graphical techniques. Visualization tools can be integrated with current testing and development frameworks to speed up debugging and encourage the use of linearizability checkers. These tools can help discover and resolve linearizability concerns more quickly by giving developers a clear, visual depiction of the system's behavior, ultimately resulting in the creation of more dependable and strong concurrent systems.

6 BIBLIOGRAPHY

[1] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential consistency: measuring and understanding consistency at Facebook. In Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15). Association for Computing Machinery, New York, NY, USA, 295–310. https://doi.org/10.1145/2815400.2815426

[2] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. 2015. Challenges to adopting stronger consistency at scale. In Proceedings of the 15th USENIX conference on Hot Topics in Operating Systems (HOTOS'15). USENIX Association, USA, 13.

[3] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. 2015. Implementing

linearizability at large scale and low latency. In Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15). Association for Computing Machinery, New York, NY, USA, 71–86. https://doi.org/10.1145/2815400.2815416

[4] Rachid Guerraoui and Eric Ruppert, 2014. Linearizability Is Not Always a Safety Property. In International Conference on Networked Systems. In: Noubir, G., Raynal, M. (eds) Networked Systems. NETYS 2014. Lecture Notes in Computer Science(), vol 8593. Springer, Cham. https://doi.org/10.1007/978-3-319-09581-3_5

[5] Alex Horn, Daniel Kroening. Faster Linearizability Checking via P-Compositionality, 2015. 35th International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE), Grenoble, France. pp.50-65, https://doi.org/10.1007/978-3-319-19195-9_4

[6] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: a complete and automatic linearizability checker. In Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10). Association for Computing Machinery, New York, NY, USA, 330–340. https://doi.org/10.1145/1806596.1806634

[7] Yang Liu, Wei Chen, Yanhong A. Liu, Jun Sun, Shao Jie Zhang, and Jin Song Dong. 2013. Verifying Linearizability via Optimized Refinement Checking. IEEE Transactions on Software Engineering 39, 7 (2013), 1018-1039. DOI: https://doi.org/10.1109/tse.2012.82

[8] Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, and Petros Maniatis. 2009. Zeno: eventually consistent Byzantine-fault tolerance. In Proceedings of the 6th USENIX symposium on Networked systems design and implementation (NSDI'09). USENIX Association, USA, 169–184.

[9] Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. ACM Comput. Surv. 49, 1, Article 19 (March 2017), 34 pages. https://doi.org/10.1145/2926965

[10] Robbert van Renesse and Fred B. Schneider. 2004. Chain replication for supporting high throughput and availability. In Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6 (OSDI'04). USENIX Association, USA, 7.

[11] Jeff Terrace and Michael J. Freedman. 2009. Object storage on CRAQ: high-throughput chain replication for read-mostly workloads. In Proceedings of the 2009 conference on USENIX Annual technical conference (USENIX'09). USENIX Association, USA, 11.

[12] Anish Athalye. 2017. Porcupine: A fast linearizability checker in Go. Retrieved March 10, 2023 from https://github.com/anishathalye/porcupine

[13] Kyle Kingsbury. 2017. Knossos. Retrieved March 11, 2023 from https://github.com/jepsen-io/knossos

[14] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03). Association for Computing Machinery, New York, NY, USA, 29–43. https://doi.org/10.1145/945445.945450

[16] David Mills, Jack Burbank, William Kasch. 2010. Network Time Protocol (NTP) Version 4: Protocol and Algorithms Specification. RFC 5905. Internet Engineering Task Force (IETF).

[17] Eric Brewer. 2012. CAP twelve years later: How the ‘rules’ have changed. Computer, 45(2), pp. 23–29. Available at: https://doi.org/10.1109/mc.2012.37

[18] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. 2011. YCSB++: benchmarking and performance debugging advanced features in scalable table stores. In Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11). Association for Computing Machinery, New York, NY, USA, Article 9, 1–14. https://doi.org/10.1145/2038916.2038925

[19] Maria Christakis, Alkis Gotovos and Konstantinos Sagonas. 2013. Systematic Testing for Detecting Concurrency Errors in Erlang Programs. IEEE Sixth International Conference on Software Testing, Verification and Validation, Luxembourg, Luxembourg, 2013, pp. 154-163. https://doi.org/10.1109/ICST.2013.50

A APPENDICES

A.1 Developer Survey Questions

1. What are the minimum hardware requirements for the testing team to execute test suites?
2. Do the users need any particular setting such as VPN to use the system?
3. What would be an acceptable amount of time for the linearizability checker to produce a result?