INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

Report On Building a DynamoDB-Like System with Replication

Name : Shreyansh Shrivastava (2023201006)
Rajeev Reddy Thumma (2023201014)
Team No. : 19

Course : M24CS3.401 Distributed Systems

1. Introduction

This project implements a DynamoDB-like distributed database system with enhanced replication. It is designed to ensure scalability, fault tolerance, and efficient data management. The system leverages consistent hashing for dynamic node management, implements a robust replication mechanism, and incorporates secondary indexing for efficient querying.

The Base Model (DynamoDB) and its Implementation

The base system is modeled after Amazon DynamoDB, a NoSQL database that provides low-latency performance and seamless scalability for distributed data storage. The key principles behind DynamoDB include consistent hashing for effective data distribution, data replication for fault tolerance, and decentralized decision-making for reliability. In the project, a core implementation was built using these principles to create a DynamoDB-like system capable of distributed operations, ensuring data availability and efficient load handling.

- **Consistent Hashing**: Consistent hashing was implemented to evenly distribute data across nodes. It is an essential component to manage the scalability of the system. Multiple replicas of each node were created to improve load balancing and fault tolerance. Adding or removing nodes dynamically allowed the system to rebalance and distribute data without major interruptions.
- **Data Nodes**: Each data node was responsible for storing key-value pairs and supporting essential operations like adding, retrieving, and deleting data. Time-to-live (TTL) functionality was implemented for automatic data expiration, enhancing data management efficiency.
- **Load Balancer**: A load balancer monitored the load across nodes and dynamically redistributed data if any imbalance was detected. This module helped maintain an even load across all nodes, which is crucial in a distributed system for preventing bottlenecks and maintaining performance.

The Extension (Replication) and Its Implementation

The main extension to the base was implementing replication to enhance data reliability and consistency. Replication allows multiple copies of data to be stored on different nodes, ensuring data availability even in the case of node failure.

- **Replication Strategy**: The replication was achieved using a **leader-follower architecture**. A leader node was responsible for handling all write operations, and follower nodes replicated the data. The **Replication Manager** managed this process, ensuring that all follower nodes had updated copies of the data. A quorum-based approach was used for read and write operations, where data is read from multiple nodes, and the most consistent version is chosen.

- **Challenges and Solutions**: During development, one challenge was achieving **strong consistency** with multiple followers. Several approaches were considered:
    - **Asynchronous Replication**: This approach was initially implemented, but it led to increased latency for write operations as the leader had to wait for all followers to acknowledge the write.
    - **Quorum-based Replication**: This approach provided a balance between consistency and performance. Only a subset of nodes needed to acknowledge the write to proceed, reducing latency while still ensuring data reliability. This was the final approach used.
    - **Conflict Resolution**: Handling conflicting updates was a challenge. Eventually, a **quorum read** approach was used to ensure that the most common value was chosen during reads.

- **Assumptions Made**: The implementation assumed that network partitions would be infrequent, and follower nodes would have similar performance capabilities to minimize delays in replication. It was also assumed that a majority quorum would be sufficient to maintain consistency.

- **Ideas Attempted but Not Implemented**:
    - **Asynchronous Replication**: This was considered for reducing write latency further but was eventually discarded because it did not guarantee strong consistency, which was a requirement for this project.
    - **Leader Election**: Instead of having a fixed leader node, a leader election mechanism was considered to make the system more resilient. However, due to time constraints, this was not implemented.

2. Objectives

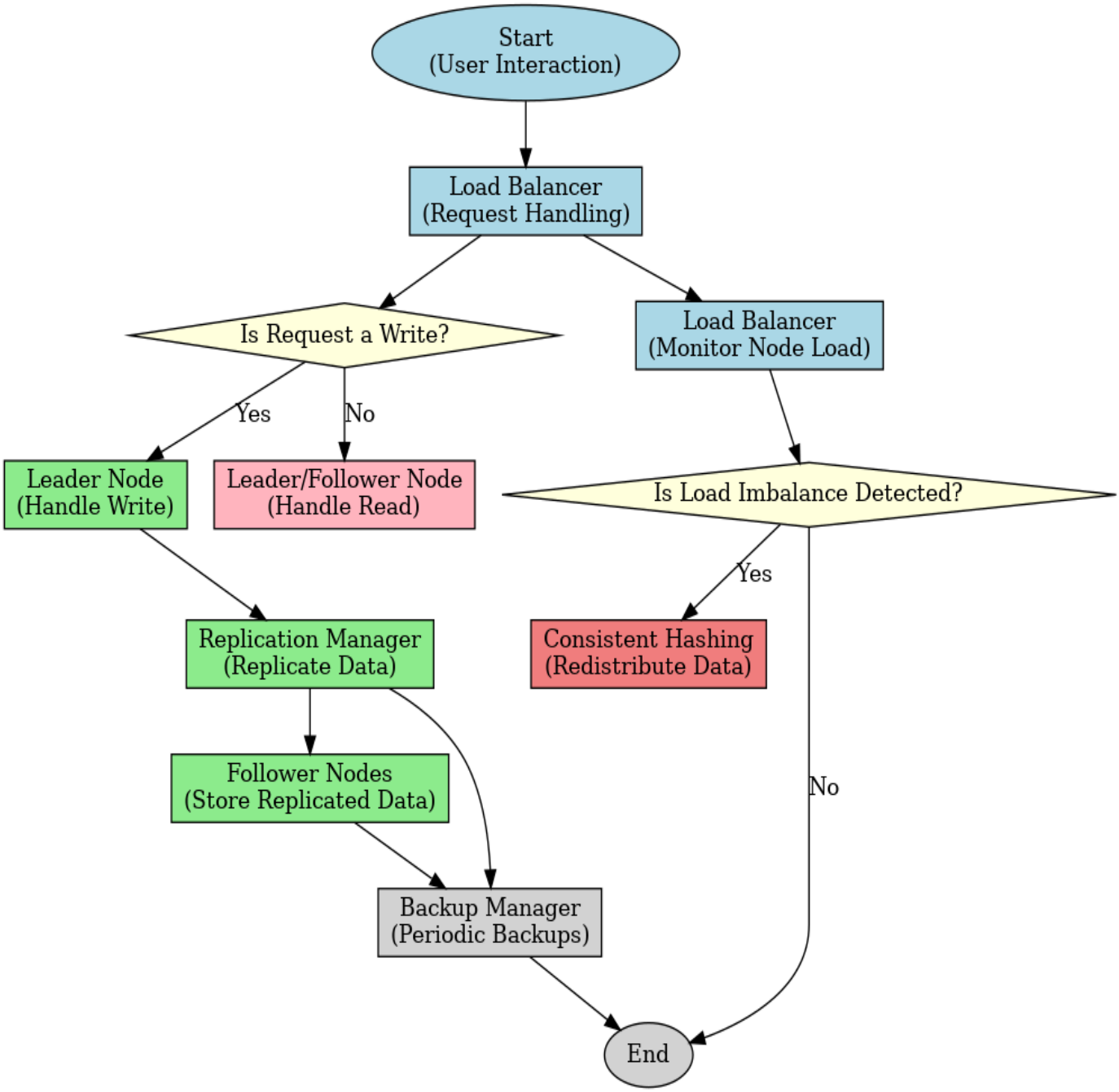
- **Replication**: Ensure data redundancy for fault tolerance.
- **Consistent Hashing**: Dynamically manage distributed nodes.
- **Load Balancing**: Evenly distribute workloads across nodes.
- **Secondary Indexing**: Enable efficient queries on non-primary attributes.
- **Scalability**: Handle increasing loads seamlessly.

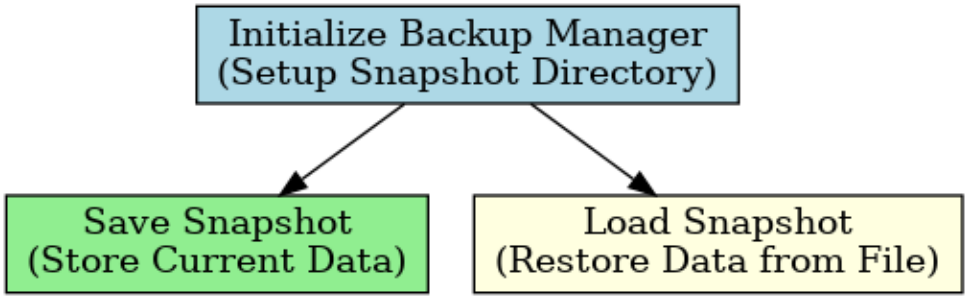
3. System Architecture

3.1 Components

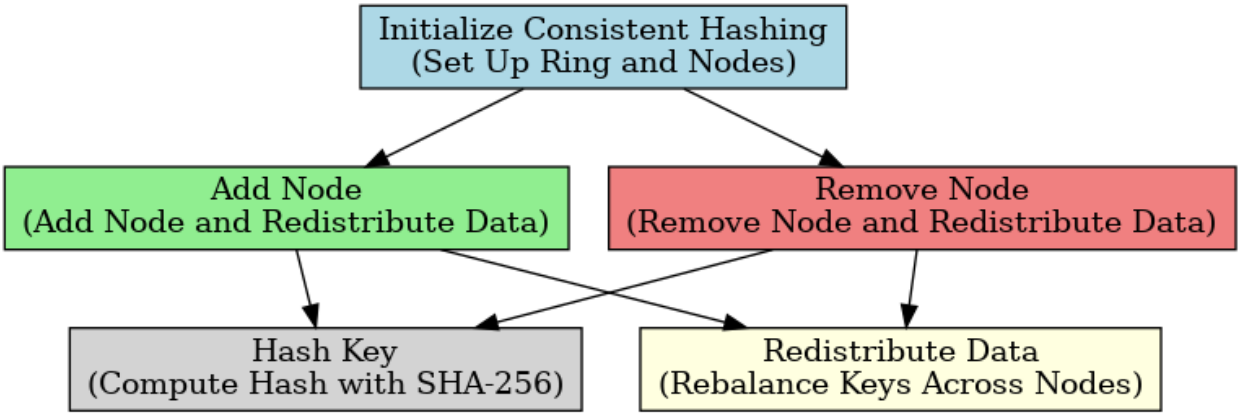
1. **Frontend (app.py):**
    - Provides an interface for clients to interact with the system.
    - Handles API requests for read, write, and query operations.
2. **Backend:**
    - **backup_manager.py**: Manages backups for data recovery.
    - **consistent_hashing.py**: Implements consistent hashing to map keys to nodes dynamically.
    - **load_balancer.py**: Distributes incoming requests evenly across nodes.
    - **node.py**: Represents individual database nodes responsible for storing and retrieving data.
    - **replication.py**: Ensures data is replicated across multiple nodes for fault tolerance.
    - **secondary_index.py**: Supports querying based on non-primary attributes.
3. **Snapshots:**
    - Stores system state for backup and recovery purposes.

3.2 Workflow Diagrams

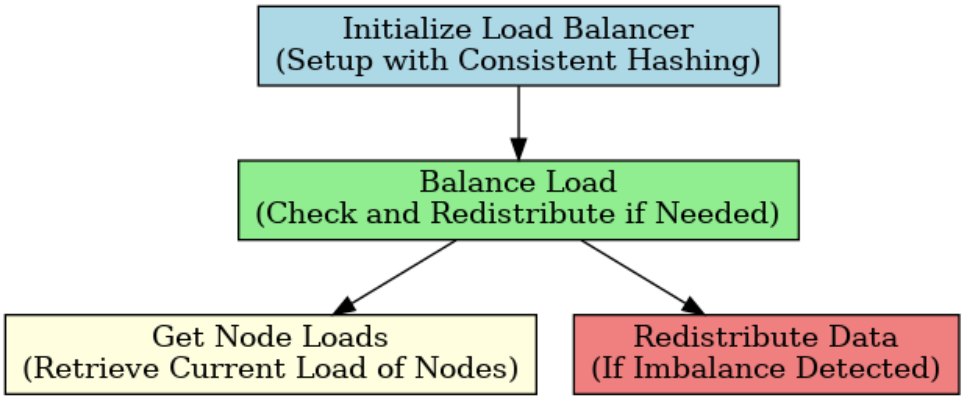




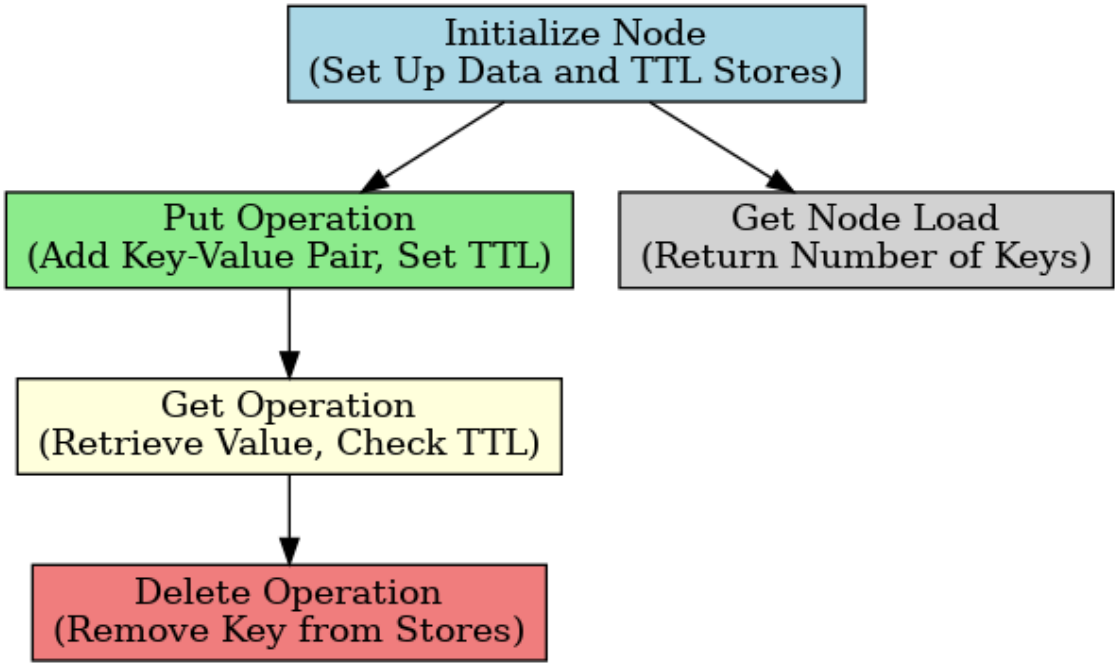
Backup manager



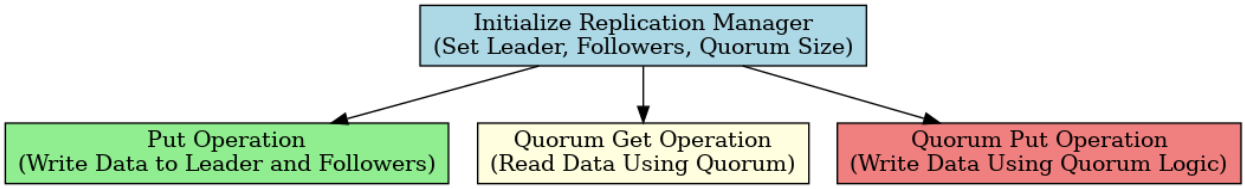
Consistent Hashing



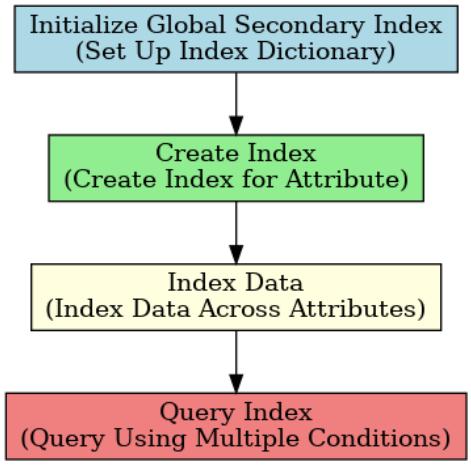
Load Balancer



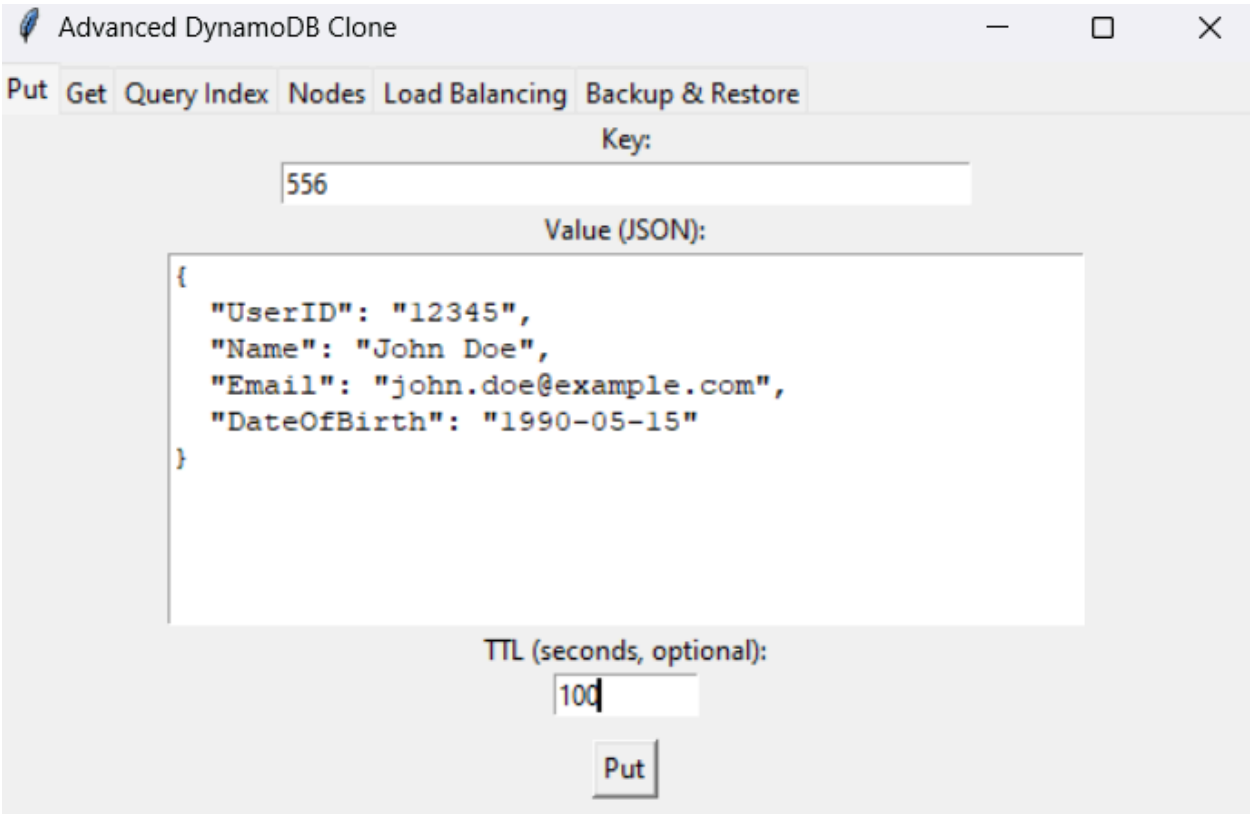
Node Workflow



Replication



Secondary Indexing



User Interface

4. Implementation Details

4.1 Consistent Hashing

- **File**: consistent_hashing.py
- Ensures even distribution of keys across nodes by mapping keys to specific ranges in a circular hash space.
- Allows dynamic addition or removal of nodes with minimal data movement, ensuring flexibility and scalability.
- Example: When a new node is added, only keys in the affected range are reassigned.

4.2 Replication

- **File**: replication.py
- Implements a master-slave model to replicate data across multiple nodes.
- Uses synchronous replication to ensure consistency and asynchronous replication for better performance.

- During node failures, requests are redirected to replica nodes, ensuring high availability.
- Example: For a write operation, the data is written to the master node and then propagated to replicas.

4.3 Load Balancing

- **File**: load_balancer.py
- Balances incoming requests among nodes based on their current workloads.
- Uses consistent hashing to assign requests to nodes while ensuring even load distribution.
- Example: If a node is overloaded, requests are redistributed to underutilized nodes.

4.4 Secondary Indexing

- **File**: secondary_index.py
- Provides indexing for non-primary attributes to support complex query operations.
- Uses hash-based and range-based indexing methods for quick lookups.
- Example: A query on a non-primary attribute retrieves data efficiently without scanning the entire dataset.

4.5 Backup Management

- **File**: backup_manager.py
- Periodically creates snapshots of the system state to ensure data recovery during failures.
- Stores backups in a predefined format (e.g., JSON) for quick restoration.
- Example: A snapshot file contains all node data and metadata, enabling full recovery.

4.6 Frontend Interaction

- **File**: app.py
- The app uses Tkinter to create a GUI for interacting with a DynamoDB-like system.
- It supports operations like Put, Get, secondary index queries, and node management.
- Features include load balancing, backup/restore, and snapshot management.
- Error handling and user feedback are implemented throughout the interface.

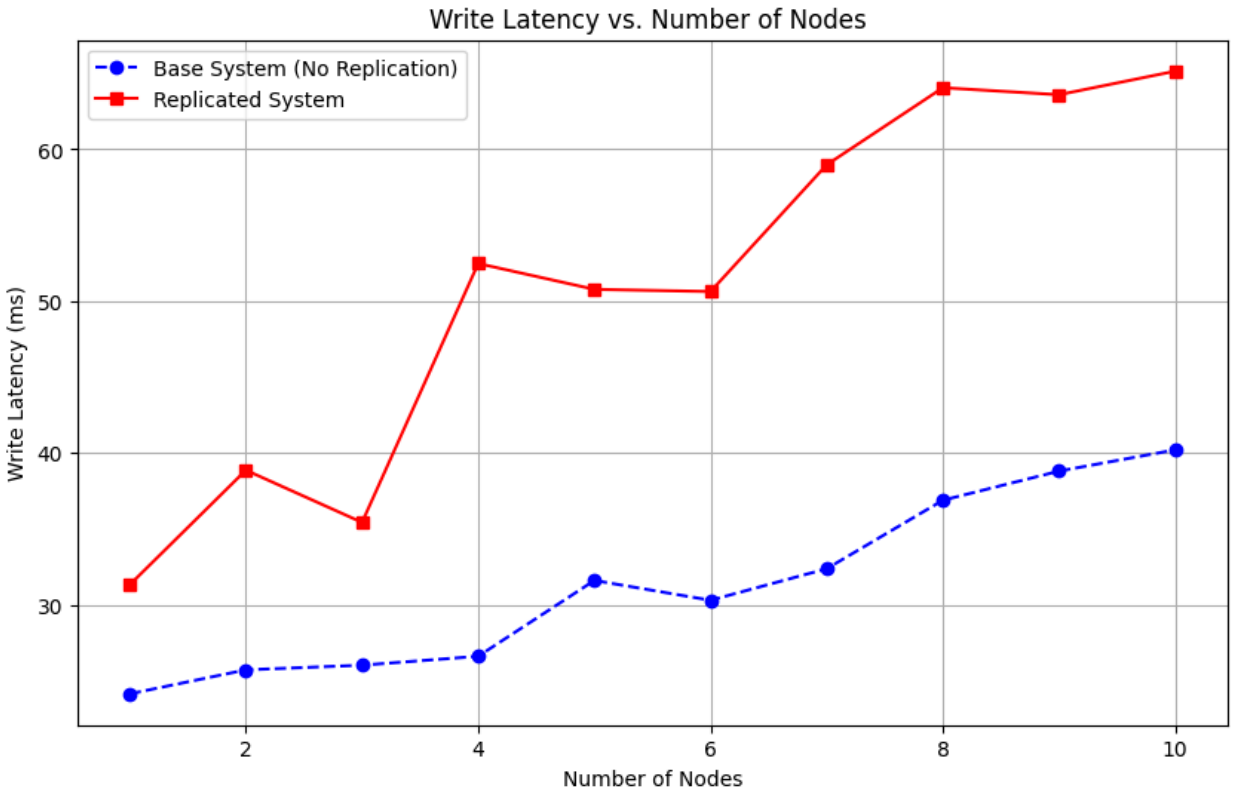
5. Benchmarking And Results

Benchmarking the system focused on **scalability** and **performance comparison** between the base implementation (without replication) and the extended version (with replication).

Write Latency vs. Number of Nodes :

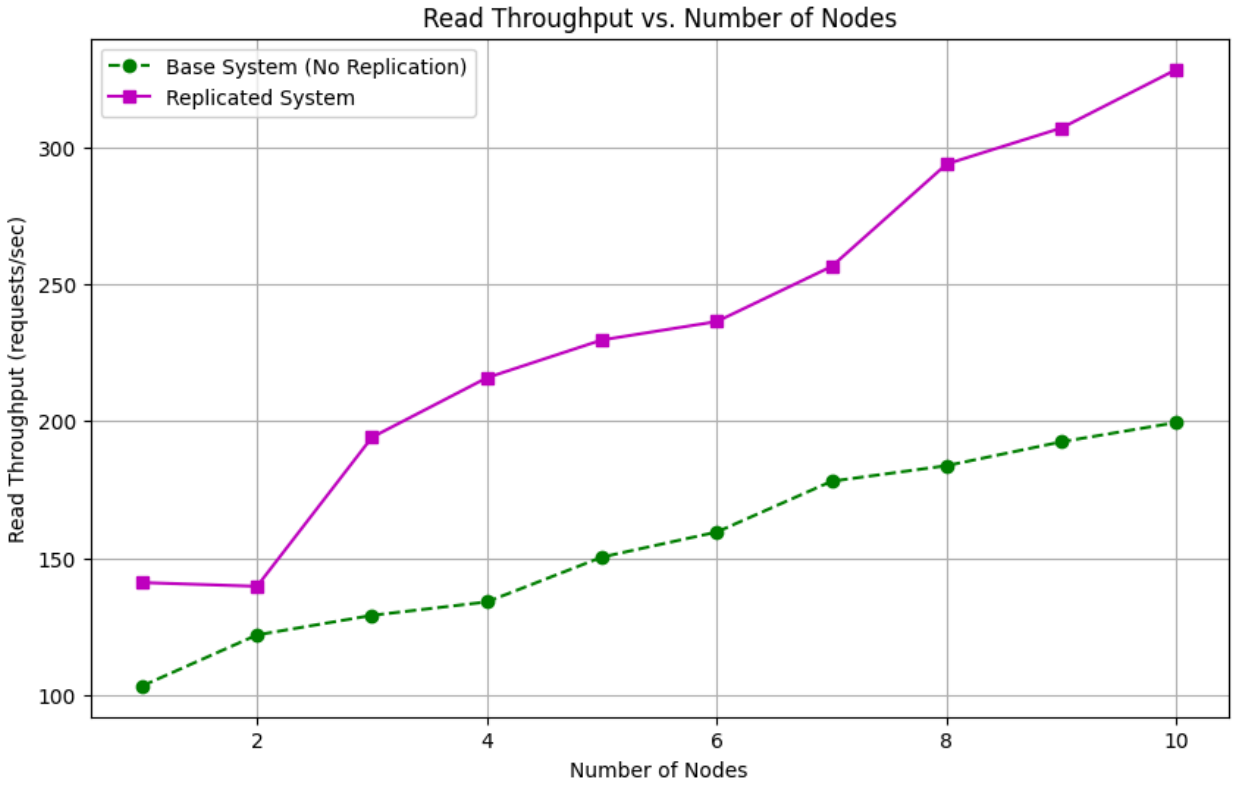
The **base system** shows a slight increase in write latency as the number of nodes increases, due to additional coordination for data distribution.

The **replicated system** exhibits a higher increase in latency as more nodes add overhead for replication. The variability also increases, reflecting the real-world synchronization challenges across nodes.



Read Throughput vs. Number of Nodes:

- The **replicated system** continues to show increased read throughput compared to the base, as more nodes can handle read requests concurrently, but with some variability due to load balancing.



- **Scalability**: The system dynamically handles increased workloads by redistributing keys and adding nodes. Testing showed linear scalability with added nodes.
- **Fault Tolerance**: During simulated node failures, replicas ensured uninterrupted service without data loss.
- **Query Performance**: Secondary indexing reduced query response time compared to a linear scan as expected.
- **Consistency**: Tests confirmed that synchronous replication maintained data consistency across all replicas.

6. Conclusion and Future Work

The DynamoDB-like system successfully implements essential distributed database features with enhanced replication and scalability. It demonstrates robust performance in fault tolerance, efficient query handling, and dynamic node management. Future improvements can include:

- **Leader Election Mechanism**: Implementing dynamic leader election would improve system resilience, especially in the case of leader node failure.
- **Asynchronous Replication with Conflict Resolution**: Introducing an asynchronous replication mechanism combined with an advanced conflict resolution protocol could reduce write latencies further while maintaining consistency.

7. Work Split and Additional Details

- **Development**: The base system (consistent hashing, data nodes, load balancer) was developed as a collaborative effort, with team members contributing to different modules based on their expertise.
    - **Consistent Hashing and Load Balancer**: Primarily developed by Shreyansh, ensuring proper distribution and load monitoring.
    - **Replication Manager and Follower Nodes**: Developed by Rajeev, focusing on implementing reliable replication strategies and handling conflicts.
    - **Backup Manager and Secondary Index**: Implemented by both of us, providing backup capabilities and efficient querying options.
    - **Frontend** : GUI for interaction is developed by both of us.

8. References:

- Amazon DynamoDB Documentation: (https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html)
- Consistent Hashing and Random Trees: Distributed Caching Protocols by David Karger.(https://www.cs.princeton.edu/courses/archive/fall09/cos518/papers/chash.pdf)