Blockchain-based Collaborative Database for a consortium of Fintech companies

A Project Report

submitted by

SIDDHARTH SINGH

in partial fulfilment of requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS

May 2023

THESIS CERTIFICATE

This is to certify that the thesis titled **Blockchain-based Collaborative Database for a consortium of Fintech companies**, submitted by **Siddharth Singh**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. John Augustine
Research Guide
Professor
Dept. of Computer Science
IIT-Madras, 600 036

Place: Chennai

Date: June 3, 2023

ACKNOWLEDGEMENTS

I express my deep gratitude and heartfelt appreciation to Prof. John Augustine and Prof. Aishwarya Thiruvengadam for their invaluable guidance, constructive criticism, and support throughout this project. Their vast knowledge, keen insight, and attention to detail were instrumental in shaping my understanding of the subject matter and enabling me to do this project to the best of my ability.

I thank Pappu Kumar for the fruitful discussions and encouragement throughout the semester. His constructive criticism and insightful suggestions were essential in helping me identify areas of improvement. I also thank Rashmi Sarode for her valuable suggestions in the writing process of this thesis.

Finally, I thank my family and friends for their immense love and support throughout my academic journey. Their constant motivation and support have been a source of inspiration, and I could not have completed this project without their love and backing.

ABSTRACT

KEYWORDS: Blockchain; Fintech; consortium; FalconDB; collaborative database

We present a collaborative database based on blockchain technology, which has been customized for a consortium of Fintech companies. This project is a part of a larger industry problem addressed by CAMS (Computer Age Management Services Limited) in collaboration with IIT Madras. The goal of the project is to provide a platform which is secure and efficient for the consortium to share information and streamline their operations.

To achieve this goal, we built the core of our database using the design of FalconDB, a blockchain-based database system. We made a few modifications to FalconDB to make it suitable for our specific requirements. Our collaborative database allows the consortium members to securely and transparently share data without the need for intermediaries, reducing the chances of fraud and errors.

This project has the potential to transform the way Fintech companies operate and collaborate with each other. It can significantly improve the efficiency and security of their operations, leading to better customer service and increased trust in the industry.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

CAMS	Computer Age Management Services Limited
KYC	Know Your Customer
DLT	Distributed Ledger Technology
CLT	Centralized Ledger Technology
CLD	Centralized Ledger Database
BaaS	Blockchain as a Service
TSA	Timestamp Authority
UTXO	Unspent Transaction Output
BRD	Blockchain Relational Database
SSI	Serializable Snapshot Isolation
IoT	Internet of Things
BFT	Byzantine Fault Tolerance
ADS	Authenticated Data Structure
API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
SQL	Structured Query Language
OCE	Order-Commit-Execute
MSP	Membership Service Provider
OSN	Ordering Service Node
IPFS	InterPlanetary File System
DHT	Distributed Hash Table

CHAPTER 1

Introduction

A key challenge facing the financial industry today is the need to conduct Know Your Customer (KYC) due diligence for each user that registers with a financial company. KYC involves verifying the identity of the user by collecting and verifying personal information such as government-issued identification documents, address verification, and other relevant information. This process can be time-consuming and expensive, and it is often repeated each time the user registers with a new financial company.

To address this challenge, a group of fintech companies has proposed forming a consortium to create a shared database of user information. By pooling their resources and information, the companies can reduce the cost and time associated with conducting KYC for each new user. This approach would also offer benefits for users, who would only need to go through the KYC process once to gain access to multiple financial services.

However, the creation of a shared database raises concerns around data privacy and security. The consortium would need to establish strict protocols for data sharing and access, ensuring that each member company adheres to industry best practices for safeguarding user information. Additionally, the use of blockchain technology could offer added security and transparency to the consortium's operations, allowing for secure and decentralized management of the shared database.

To create a shared database that meets the needs of a consortium of Fintech companies, there are several key requirements to consider. Two of the most important conditions are:

1. **The database must be distributed.** There are many benefits to using a distributed database instead of a centralized one. One key advantage is that it avoids a single point of failure. By replicating the data across multiple machines, the system can continue to function even if some of the machines fail. This is critical for a large consortium of companies that are dealing with millions of data entries, as managing a centralized database can become very challenging at such a scale. Distributed databases are also better equipped to handle vast amounts of data with

better scalability and fewer performance issues than centralized databases. Additionally, distributed databases provide high availability and data consistency, which are crucial for a shared database used by multiple companies.

2. **Updates must be based on a consensus.** It is important that any updates to the shared database go through a consensus process to ensure that all member companies agree on the changes. This is especially important because these companies are competitors in the same field and may have malicious intentions towards each other. Consensus ensures that each update is validated and approved by a majority of the consortium members before it is reflected in the database. By using a consensus-based approach, the consortium can reduce the risk of fraudulent or malicious updates that could compromise the integrity of the shared data. This can help build trust between the consortium members and ensure the continued success of the shared database over time.

Ensuring the safety and reliability of the shared database for a consortium of fintech companies is paramount. To achieve this, we need to adhere to two main conditions. Firstly, the database should be distributed to avoid a single point of failure, facilitate scalability, and ensure high availability and data consistency. Secondly, updates to the database should be based on a consensus to prevent malicious actors from influencing each other.

Fortunately, researchers worldwide have already paid significant attention to the concept of blockchain-based databases, which offer tamper resistance and improved security.

These databases can be built on either Centralized Ledger Technologies (CLTs) or Distributed Ledger Technologies (DLTs). We will explore both types of databases in detail in Chapter 4.

CHAPTER 2

Background

In recent years, there has been a proliferation of novel ledger technologies aimed at providing decentralized and trustless collaboration, building upon the blockchain foundation laid by Satoshi Nakamoto's Bitcoin white paper [1]. Despite ongoing efforts to address issues and shortcomings through research and development [2; 3; 4], blockchain is still in its nascent stages, with new technologies emerging frequently and countless questions and performance challenges remaining unanswered. Nonetheless, the growing interest and application of ledger databases in various industries, such as global healthcare [5; 6; 7; 8; 9], finance [10; 11; 12; 13; 14], and the Internet of Things [15; 16; 17; 18; 19; 20], present promising opportunities for further development. Leveraging the blockchain data structure and decentralized architecture, ledger technology offers robust security against malicious activities and privacy management. As such, there is a pressing need for ongoing improvements and innovation to meet the diverse and evolving needs of various fields leveraging ledger technology.

Blockchain is a decentralized ledger technology that has traditionally been open and accessible to all, allowing anyone to join and access public data. However, the rise of private enterprises adopting blockchain has resulted in new demands and requirements. To address this, a private network model was developed that preserves the fundamental principles of blockchain while restricting access to the creators and maintainers of the network. This model, known as a permissioned blockchain, meets the specific needs of private entities seeking greater control over their data and network.

Blockchain technologies are decentralised and utilize Decentralized Ledger Technologies (DLT). Their performance and transaction throughput are lower than conventional distributed databases or centralised ledger technologies. While database systems based on blockchain are inherently safer than traditionally distributed databases against malicious attacks, DLT's limitations paved the way for developing a centralised technique that can match the performance of traditional distributed databases. This approach, called Centralised Ledger Technologies (CLT), makes use of the blockchain data structure for storing data and is called a Centralised Ledger Database (CLD) [21]. CLD differs from blockchain in that it is maintained by a trusted third party and does not require a consensus mechanism. It is important to discuss CLD and permissioned blockchain-based databases for secure and tamper-proof databases. CLD and permissioned blockchain-based databases are often categorised as “ledger databases”, which offer trustless collaboration between enterprises and are ideal for corporate use.

CHAPTER 3

Basic Concepts

3.1 Blockchain

Blockchain is most appropriately defined as a Decentralised Ledger Technology (DLT) that uses peer-to-peer connections between nodes to create a decentralised system that supports a trustless environment. The data in this technology is organized into blocks, which are then chained together. The consensus mechanism is crucial for the transaction processing in blockchain, as it provides a way to order transactions between nodes. Blockchain networks can be classified as either permissioned or permissionless [22; 23], depending on whether nodes can join the network without restrictions or are limited to a specific set of maintainers. Permissioned networks are usually used by private enterprises, making Sybil attacks less of an issue. Both types of networks provide secure and tamper-resistant storage of data in a ledger, allowing for easy validation.

Centralised Ledger Databases (CLD) share some key characteristics with blockchain, including the use of a trusted third party to maintain the system, resulting in an increase in centralisation. Unlike decentralised structures, consensus mechanisms are not typically necessary in CLD. Large tech companies that offer Blockchain as a Service (BaaS) [24; 25] often act as these trusted third parties, providing an easy way for smaller businesses to integrate blockchain functions. In terms of security, permissioned blockchain architecture is more secure than permissionless blockchain since it requires identity assumptions. However, CLD is even more secure than permissioned blockchain due to the presence of a trusted third party.

When comparing the security of permissioned and permissionless blockchains, it becomes evident that permissioned blockchains are more secure due to their centralized nature. This is due to the fact that the consensus mechanism used in permissionless blockchains is probabilistic, meaning that agreement among nodes regarding new blocks is reached with a certain degree of probability. While this mechanism ensures consistency in the ledger, it also opens up the possibility of a ledger fork, which occurs when different nodes are having different ledger states. This can lead to uncertainty and inappropriate results from the perspective of the enterprise, as different nodes in the blockchain network would be using different ledger states.

On the contrary, permissioned blockchains use deterministic consensus mechanisms, which always ensure a state of the ledger that is agreed upon globally. This is an important criterion for blockchain-based databases, as having a globally agreed upon ledger state is crucial to ensuring the accuracy and reliability of transactions. Due to this, permissioned blockchains are generally considered more suitable for use in blockchain-based databases.

However, it should be noted that the higher degree of security offered by permissioned blockchains is not without its trade-offs. In particular, permissioned blockchains require identity assumption due to their public nature, which can lead to issues with privacy and confidentiality. Nonetheless, the centralized nature of permissioned blockchains allows for a higher degree of control and oversight, which can be beneficial in certain use cases. Overall, the choice between permissioned and permissionless blockchains ultimately depends on the specific requirements of a given application or use case.

Smart contracts, while a valuable addition to blockchain technology, are not ubiquitous and are not always employed in every blockchain system. A smart contract is a piece of code that can be executed on a blockchain, resulting in alterations to the state of the blockchain. It operates as a program which is deterministic in nature, meaning that its execution always results in the same outcome. Smart contracts are employed in various programming languages, such as Solidity or Java. The operations that can be carried out on a smart contract, as well as the current state of the database, can be managed and controlled through predefined rules.

3.2 Distributed Database vs. Blockchain

Several research papers [26; 27; 28; 29] have explored the similarities and differences between blockchain and distributed databases. These two technologies share fundamental similarities but differ in many ways. Blockchain is commonly used in cases where there is no trust among nodes, while distributed databases are used where performance matters more than security. Distributed databases excel in performance by offering increased transaction throughput and reduced latency. In contrast, blockchain experiences higher latency due to its utilization of consensus mechanisms for ensuring strong consistency and its decentralized nature. Despite the trade-off in performance, blockchain's decentralized structure opens up new possibilities for applications. An essential distinction between the two lies in data deletion and modification, as the immutability of blockchain's ledger makes these actions impossible, whereas distributed databases often support such functionalities, albeit they are seldom implemented. Twenty years ago, the CAP theorem was introduced [30] as a framework for categorizing distributed databases according to their consistency, availability, and partition tolerance. It states that it is not possible for a distributed database to achieve all three of these features simultaneously. In blockchain technology, the DCS theorem [31] was proposed to define its property set, comprising decentralization, consistency, and scalability. Only two of these system properties are available at once, and total decentralization cannot be achieved in permissioned blockchain due to its enterprise-associated characteristics. As a result, consistency and scalability must be emphasized in these cases.

CHAPTER 4

Literature Survey

4.1 Centralised Ledger Databases

4.1.1 LedgerDB

LedgerDB [21] developers have pointed out that applications constructed on permissioned blockchains often encounter performance challenges, including limited throughput, increased latency, and substantial storage requirements. Consequently, they propose that centralized systems could be a more advantageous choice when decentralization is not essential, particularly considering the prevailing industry shift towards a single service provider. LedgerDB aims to tackle these concerns by prioritizing high write performance while ensuring satisfactory performance for read and write verification.

The LedgerDB architecture consists of three types of nodes: the ledger master, the ledger proxy, and the ledger server. The ledger master is responsible for managing all the cluster data and events. The ledger proxies and servers are stateless, which enables seamless resolution of issues. The ledger proxy pre-processes client requests and sends them to the appropriate ledger server. The server handles the requests, interacts with the underlying storage layer, and stores the data. The storage layer can be implemented using different technologies like file systems, key-value stores, and more. To enable data replication across various regions and zones, the storage layer utilizes a shared abstraction implemented through the Raft protocol. Additionally, data deletion from the database via the API is feasible without compromising the verification or audit process, resulting in reduced storage needs.

Preserving the threat model of LedgerDB necessitates placing trust in a third-party timestamp authority (TSA). The TSA plays a crucial role in providing accurate and legitimate timestamps for transmitted data. This is accomplished through the implementation of a specialized journal known as a TSA journal, which comprises a ledger snapshot along with an approved timestamp from the TSA. By relying on a TSA, LedgerDB can establish the existence of data before a specific moment in time, effectively reducing the vulnerability window. In a two-way pack protocol, the TSA signs the ledger digest, resulting in its inclusion as a TSA journal within the ledger.

4.1.1.1 Verification

LedgerDB facilitates the verification and authentication of returned journals via its dedicated verify application. The integrity of proofs is maintained through their signature by a specific component. The verification process can be conducted either on the server side or the client side, with the option to utilize trusted anchors to expedite the process. These anchors serve as reference points within the ledger, indicating positions where the verification of the digest has been previously validated.

4.1.1.2 Transaction Processing

In LedgerDB, transactions modify the state of the ledger, resulting in journal entries that are added to the ledger. A block comprises multiple journal entries. LedgerDB uses an execute-commit-index transaction model, as shown in Figure 4.1, instead of the traditional approach of order-then-execute. This model leverages the centralized structure and links the execution and validation phases. Since the network is managed by a single service provider, there is no need for a consensus mechanism to be implemented.

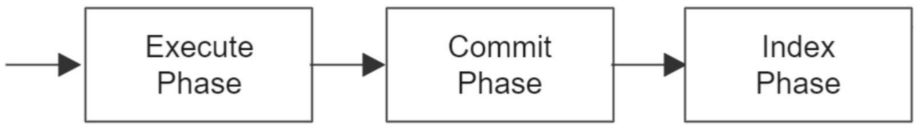


Figure 4.1: Execute–commit-index transaction processing model

The performance challenges of verification models in permissionless and permissioned blockchains, specifically in data management involving smart contracts or UTXO, persist due to factors like the requirement for recursive searches in the UTXO verification model. To tackle this issue, LedgerDB introduces user-defined labels called “clues” that incorporate business logic. Each asset is assigned a distinct ID and can be ordered using these clues. This approach guarantees that each state of an asset is linked through a single clue index, deviating from traditional key-value models.

Execute Phase: After the client initiates a transaction, it undergoes verification and authorization checks to ensure the necessary permissions for the intended operations. After the successful completion of the verifications, the transaction is saved in the storage layer, which effectively minimizes expenses associated with storage and communication. Once executed, the transactions are sent to the relevant ledger server, regardless of whether they were successful or not.

Commit Phase: The journals from the Execute Phase are collected by the ledger server and then processed as a group in a globally ordered manner. The transactions are subsequently sent to the storage system for permanent storage, with only the successfully executed transactions being retained. Clients are provided with a receipt that serves as an acknowledgement of their transactions, irrespective of their outcome, and serves as a status update for their respective transactions.

Index Phase: The commit phase involves the generation of specific indexes for the data stored, which serve the purpose of retrieval and verification. These indexes encompass the bAMT accumulator, the clue index, and block information. Unlike the digest of an individual block, the block information provides a comprehensive overview of the entire ledger. This facilitates the simultaneous verification of all ledger information, as it is derived from this consolidated summary created at a specific point in time. Once the indexes are established, a confirmation receipt is issued to the client. The index phase holds significant importance in upholding the accuracy and integrity of the data stored within LedgerDB.

4.2 Permissioned Blockchain Based Databases

4.2.1 Blockchain Relational Database

The Blockchain Relational Database (BRD) [32] shares a comparable conceptual framework to that of Hyperledger Fabric [33], albeit functioning at a more granular level as an integral component of PostgreSQL. This has led to a significant reliance on PostgreSQL as a relational database. The system incorporates 4,000 lines of C code into PostgreSQL and has one of the most advanced transaction management approaches within the domain of blockchain-driven databases. Empirical studies conducted by researchers have demonstrated the system's remarkable capacity for handling a substantial volume of transactions. In this particular system, PostgreSQL serves as the underlying framework, housing smart contracts as stored procedures. Database nodes function independently, undertaking transaction processing and engaging in decentralized consensus to determine transaction sequencing. Transaction processing occurs through two distinct methods: the initial, albeit less efficient, ordering approach, and the concurrent execution approach coupled with simultaneous ordering. Notably, the system encompasses three distinct node types.

1. **Client:** Within the system, a designated administrator assumes responsibility for overseeing client network access management. The digital certificates of clients are stored on the database peers, facilitating secure and authenticated network entry. Clients possess the capability to subscribe to a notification channel to receive real-time updates regarding the progress of their transactions.
2. **Database peer node:** In a blockchain system, nodes play a crucial role in storing and managing important data. Every node within the network retains an individual copy of the ledger, assuming the duty of executing smart contracts. Furthermore, these nodes are entrusted with the validation and incorporation of fresh blocks into the blockchain.
3. **Ordering service:** It is a crucial component of the blockchain network that facilitates consensus among the orderer nodes. In case the ordering service is ordered nodes, it is mandatory for them to be owned by different organisations to prevent conflicts of interest. This service is highly flexible and can be easily customized to suit the requirements of different organizations, and is designed to be independent of any specific technology or platform.

4.2.1.1 Serializable Snapshot Isolation (SSI)

Serializable Snapshot Isolation is an innovative approach that enhances the Snapshot Isolation (SI) [34] technique to attain serializability. This is achieved by leveraging a Multi-Version Serialization [35] History Graph, which effectively identifies SI anomalies. In this graph, each transaction is depicted as a node, while edges symbolize the execution order between transactions. Notably, three distinct types of dependencies can establish these edges.

- rw-dependency
- ww-dependency

- wr-dependency

Within the Multi-Version Serialization History Graph employed by Serializable Snapshot Isolation (SSI), anomalies are effectively detected by identifying cycles present in the graph structure. The existence of cycles signifies a departure from the expected serial execution order. To rectify these anomalies, SSI meticulously tracks read-write (rw) and write-read (wr) dependencies, promptly aborting them during the commit phase. This process is made possible by maintaining two separate lists for each transaction.

SSI Based on Block Height: A novel adaptation of SSI has been developed, which leverages block height for parallel transaction execution and consensus-driven ordering. SSI does not guarantee identical committed transaction states across all database nodes by default. However, to ensure consistent execution with identical committed data on all nodes, every row within a table contains creator and deleter block numbers, indicating the block number responsible for row creation and deletion. Submitted transactions include the designated block height and number for execution. While this SSI variant is exclusive to a specific approach, it is seamlessly integrated into implementing the order-then-execute methodology for provenance queries, ensuring appropriate serializability. To maintain this serializability, any comparable transactions resulting in phantom reads or stale data reads are promptly aborted.

SSI variant—block-aware abort during commit: This technique ensures consistency across all nodes by aborting the same set of transactions on each node. A specialized table known as the "abort rules table" is implemented to identify transactions that require aborting, utilizing a variant of SSI. In the context of this SSI variant, concurrent transactions that successfully commit during or after the current block can potentially lead to phantom reads or access to stale data. To address this concern, the SSI variant tracks and monitors such occurrences during the commit phase, ensuring the appropriate handling of these transactions.

4.2.1.2 Transaction Processing

In establishing a blockchain-based relational database among nodes with varying degrees of trust, two distinct strategies come into play. The initial approach, coined as "order-then-execute", mandates the involvement of a consensus service that meticulously arranges transactions before database peer nodes simultaneously execute them. Contrarily, the second technique follows a distinct path where database peer nodes execute transactions sans prior knowledge of their order, leaving the determination of order to an ordering service during execution. While the latter method boasts enhanced performance compared to the former, it necessitates more comprehensive modifications and implementation within the relational database, thereby engendering a tradeoff.

Order then Execute: In the Order then Execute approach, the client transactions are processed in a specific sequence. Initially, a sorting process is facilitated by an ordering service to arrange the transactions. Subsequently, the transactions are executed and confirmed on the database peer node before being recorded during the checkpoint phase. This model's visual depiction can be observed in Figure 4.2.

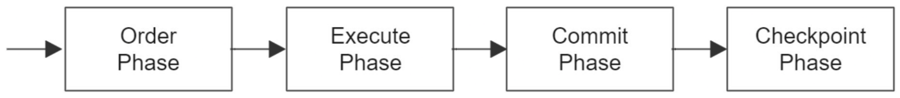


Figure 4.2: Order-then-execute transaction processing model

Order Phase: During the order phase, clients submit their transactions to any of several ordering service nodes. At intervals (e.g., every one second), these nodes work to reach a consensus on the order of the transactions, resulting in the creation of a new block. The ordering service then broadcasts the block to all nodes.

Execute Phase: After the ordering phase, the execute phase begins, during which each database node confirms the correct sequence of the current block and its origin from the ordering service. Each transaction in the block is assigned a thread and executed independently on each node with its corresponding arguments. To ensure consistency across all nodes, the SSI variant, using the 'abort during commit' technique, guarantees the same committed state. The next block is processed only after the current block is committed.

Commit Phase: Following the completion of transaction execution, the commit phase commences. During this phase, transactions have the potential to be either committed or aborted. The order in which transactions are included in the block mirrors the order in which they are committed. This synchronicity guarantees consistency across all database nodes with regard to the commit order. Just as in the execute phase, the SSI variant known as “abort during commit” is employed to ensure that transactions are executed based on the same committed state.

Checkpoint Phase: The commit phase follows the execution of all transactions and determines if they are to be committed or aborted. The order of transactions in the block matches their commit order, ensuring consistency across all database nodes. Much like the execute phase, the SSI variant referred to as ‘abort during commit’ is harnessed to ensure uniformity in the committed state of transactions.

Execute and Order in Parallel: Demonstrating superior performance compared to the aforementioned approach, the execute and order phases in this method operate in parallel. Simultaneously, ordering is performed by the ordering nodes, while execution takes place on the database nodes. Subsequently, the commit and checkpoint phases follow suit. A visual representation of this model is depicted in Figure 4.3.

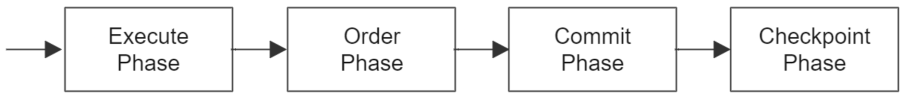


Figure 4.3: Execute and order in parallel transaction processing model

Execute Phase: In the execute phase, clients follow the established transaction submission process as previously described. In this particular scenario, upon receiving the transaction request, the respective database node dispatches the request to other nodes and the ordering service for background ordering. Each transaction received by a database node is assigned a dedicated thread, facilitating the forwarding and execution of the transaction. Consequently, execution and ordering processes occur concurrently. This parallel action results in the inclusion of a shared transaction block height, ensuring uniformity in the committed data across all database nodes.

Order Phase: In contrast to the order-then-execute approach, the order phase in this method receives transactions from database nodes via an ordering service. This allows for simultaneous ordering while transaction execution occurs on the database nodes. The remaining processes follow the same steps as in the order-then-execute approach.

Commit Phase: The commit phase of the order-then-execute approach involves executing all transactions in a block and determining whether they should be committed or aborted. However, there are two key differences in this phase due to the parallel architecture of the approach.

1. If a block contains unexecuted transactions, they are executed before continuing to the commit phase.
2. Blind updates are not allowed because transactions may be executed at different snapshot heights, leading to inconsistencies.

Checkpoint Phase: The checkpoint phase in the approach of order-then-execute is the same as the checkpoint phase in this approach. No differences exist between the two approaches in this regard.

4.2.2 BigchainDB

BigchainDB [36] is an open-source blockchain-based database that was released in 2016 and has since been continuously improved. Despite experiencing a decline in its active programming community in recent times, this blockchain database continues to be widely recognized and highly regarded in the research domain. BigchainDB focuses on examining transaction models for blockchain databases from an owner-controlled assets perspective, providing solutions to supply chain management, intellectual property rights management, and data governance, among others. It is particularly useful in cases where comparable assets, such as products or people, need to be managed. However, some studies [37] have suggested that implementing smart contract features can provide better opportunities for supply chain management, and there are concerns about the high resource usage of BigchainDB in such cases due to the use of IoT devices [38].

The fundamental principle underlying BigchainDB revolves around delivering database-like attributes encompassing low latency, high transaction rates, and ample capacity. This is achieved through the utilization of the MongoDB database, where each node maintains its local replica. BigchainDB seamlessly integrates the Tendermint consensus mechanism [39], known for its Byzantine Fault Tolerant (BFT) nature [40]. This makes the resulting system BFT as well, providing greater security and resilience. Despite its decreasing community, BigchainDB remains a notable solution in the blockchain-based database research field.

4.2.2.1 Owner-Controlled Assets

Like many other blockchain systems, BigchainDB employs the concept of owner-controlled assets, which means that only the asset owner(s) can transfer the asset, and not the node operators. Unlike other blockchain systems, BigchainDB allows for the creation of multiple assets, each of which can store arbitrary data in JSON format. The only limitation is the size of the transaction, which can be adjusted. These assets can be divided into smaller units but cannot be less than one unit. Additionally, BigchainDB allows for the storage of arbitrary JSON documents as data and metadata.

The transaction model employed by BigchainDB incorporates a metadata field, offering users the ability to append supplementary information to transactions while preserving the integrity of the original asset data. This metadata field proves particularly valuable within TRANSFER transactions, as it allows for the inclusion of additional details regarding the transferred asset. Notably, the TRANSFER and CREATE operations stand out as pivotal transaction operations within BigchainDB, as they facilitate the creation and modification of data.

BigchainDB encompasses a comprehensive set of five distinct transaction operations in total.

- CREATE
- TRANSFER
- VOTE
- VALIDATOR_ELECTION
- CHAIN_MIGRATION_ELECTION

The final three operations are related to the consensus mechanism used by the system, while the CREATE and TRANSFER operations are the most relevant for creating and updating asset data.

Create: BigchainDB offers a CREATE transaction feature that facilitates the registration of assets, allowing users to include both data and metadata of their choice. This transaction type supports the registration of various types of assets, regardless of their divisibility or indivisibility, and allows for the assignment of multiple owners, including the possibility of no owner. Only the designated owner/s have the authority to utilize the registered asset. Furthermore, the required consensus for validating a transaction as legitimate can be customized to specify the number of owners who must agree.

Transfer: In order to transfer an asset, a user must meet a certain condition and have permission to do so. This means that only certain users are allowed to transfer ownership of an asset or modify its metadata.

4.2.2.2 Transaction Processing

BigchainDB employs a simple transaction processing method known as order-then-commit, which is illustrated in Figure 4.4. This approach resembles the order-then-execute transaction processing paradigm. In this process, transactions are received and arranged by the Tendermint system, after which each server commits the block to its local MongoDB database based on the response from Tendermint. Subsequently, each server relays a response back to its respective Tendermint component.

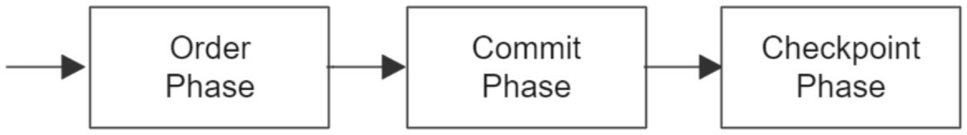


Figure 4.4: Order-then-commit transaction processing model

Order phase: Ordering transactions in BigchainDB follows a process where clients communicate with the system via APIs and the Gunicorn web server handles incoming HTTP requests. Once a message is received, it is passed on to the Tendermint instance. Transactions in the memory pool are included in a block after validation, and consensus is achieved via the Tendermint Broadcast API. All servers in the network vote on the validity of the block before it is successfully created.

Commit phase: During the commit phase, after achieving consensus among all Tendermint components, BigchainDB proceeds to store the block in the local MongoDB database. Notably, unsuccessful transactions are not retained in BigchainDB, and no details regarding them are exposed through APIs. In contrast, both successful and unsuccessful transactions are encompassed within the current blocks managed by Tendermint.

Checkpoint phase: In the checkpoint phase of transaction processing in BigchainDB, each server transmits a distinct response message to its corresponding Tendermint component to signify the successful completion of block commitment to the local MongoDB database.

4.2.3 FalconDB

FalconDB [41] is a decentralised network where nodes can function as either server or client, accommodating clients with limited hardware resources. The network is designed to operate efficiently even with low storage, computation, and bandwidth requirements. The storage of the entire database is exclusively maintained by the server nodes, whereas the clients retain only the headers of the blocks. Authentication Data Structures (ADS) are securely stored within the database on the server nodes to assist clients in verifying their requests sent to the server nodes. By leveraging the ADS, a digest is generated to represent the current content of the database. This digest serves as a means for clients to authenticate the outcomes provided by the server nodes. Furthermore, each block header contains the digest corresponding to the respective state, thereby allowing clients to access the digest associated with the present database content. This design allows FalconDB to tolerate up to 1/3 of malicious nodes, and the network can still function even if all server nodes are malicious, except for one.

FalconDB is platform-agnostic and can support any blockchain platform. It can also incorporate a variety of ADS solutions to meet the specific requirements of the platform. FalconDB is designed to function under an incentive model to ensure that honest replies are provided for queries and update requests. While server nodes are not focused on generating profits, a shared cost environment can be established based on usage. In the event of a dishonest node, a financial penalty is imposed to discourage such behavior.

4.2.3.1 Authenticated Data Structure

Detecting and mitigating potential risks posed by malicious server nodes in outsourced databases is an essential aspect of secure and optimal data management. A promising solution in this domain is Authenticated Data Structures (ADS) [42], which empowers clients with the capability to authenticate server results and validate both query and update requests. ADS encompasses five fundamental functions that facilitate querying, validation, and the generation of a digest representing the current state of the database.

To ensure the accurate execution of a request, it is imperative to conduct a thorough verification and validation process on the received proofs and results from the server node. This validation process is deemed successful only if the server has diligently executed the necessary ADS functions. Consequently, ADS emerges as a critical component within outsourced database systems, aiming to uphold the integrity, security, and efficiency of operations, while effectively mitigating potential malicious attacks on server nodes.

ADS offers a reliable way for clients to verify the validity of a response from a server node. This helps to minimize the risk of malicious activity, such as tampering with data or attempting to deceive clients. By providing a secure and authenticated data structure, ADS is an essential tool in ensuring the integrity and accuracy of data stored in outsourced databases. Additionally, ADS can be used to maintain the privacy of sensitive information, as it enables data to be accessed only by authorized parties, and can provide evidence of unauthorized access.

Limitation of ADS: Authenticated Data Structures (ADS) have some limitations that are addressed by blockchain architecture.

1. The initial constraint pertains to the risk of encountering issues caused by an outdated or malicious digest, as it is crucial for users to possess the most up-to-date and valid digest. However, this concern is effectively mitigated by employing a Byzantine Fault Tolerant (BFT) consensus mechanism, which guarantees the synchronization of the digest across all clients.
2. The subsequent constraint involves the potential for attackers to manipulate data by inserting or deleting records, rendering them untraceable by ADS, thereby eluding identification. Nevertheless, this challenge is effectively countered by the inherent properties of immutability and transparency offered by blockchain technology. Through the immutability feature, data remains unalterable, and any modifications are permanently recorded on the blockchain, enabling straightforward traceability and identification of malicious actors.

One of the limitations of a distributed ledger database is that it can suffer from synchronization issues among clients. This issue can be addressed by implementing a Byzantine Fault Tolerance (BFT) consensus mechanism, which can help ensure that the digest is synchronised across all clients. Another limitation of distributed ledger databases is the potential for malicious actors to manipulate the data. This issue can be addressed through the immutable and transparent properties of the blockchain, which can make the data tamper-proof and easily traceable.

4.2.3.2 Transaction Processing

FalconDB presents an innovative approach that promotes the utilization of a reliable service even for clients with limited hardware capabilities. The model operates on a system of incentives, motivating servers to deliver honest services while imposing penalties for any dishonest conduct. This incentivization framework is built upon smart contracts deployed on platforms like Ethereum, enabling seamless communication and deposit handling between clients and servers. Within this model, a fee is required to be paid either by the client or the server. In the event of malicious behavior by a server, all funds stored in the smart contract account will be forfeited. These smart contracts serve as integral components of the system, facilitating secure and incentivized interactions between clients and servers.

- Service fee contract: clients provide compensation to servers for query and request processing services.
- Authentication contract: temporarily restricts access to the server's account until it provides verifiable evidence of the requested smart contract.

FalconDB is composed of two main layers: the authentication layer and the consensus layer. Understanding the functioning of FalconDB requires a detailed look at how a query is processed through both layers.

Consensus layer: In FalconDB, the consensus layer handles the processing of queries sent by clients to the server nodes through the service fee smart contract. Servers receive a fee for processing these queries and return the results to the client upon execution. Once the BFT consensus protocol is agreed upon, the new block is committed to the blockchain, and the updated digest becomes available to all clients.

Authentication layer: The authentication layer in FalconDB allows clients to verify the success of their requests by paying an additional fee to the server through the authentication contract. This generates an ADS proof that validates the digest for the client.

4.2.4 ChainifyDB

ChainfyDB [43] is a transaction processing system that aims to allow organizations and companies to execute transactions in a heterogeneous way. Unlike traditional transaction processing systems, ChainifyDB connects various database managers with different implementations. The divergence in transaction execution is caused by different interpretations of the SQL standard or the use of different data types. To address this issue, a new model called the Whatever-Ledger Consensus model was created.

The Whatever-Ledger Consensus model enables the chaining of heterogeneous infrastructures by dividing the transaction processing into two phases: the whatever phase and the ledger-consensus phase. In the whatever phase, each organization processes a batch of inputs according to their own rules, without any assumptions. Then, in the ledger-consensus phase, the system endeavours to establish agreement on the transactions at hand.

In order to establish consensus, ChainifyDB employs a flexible voting algorithm that can be customized according to the desired number of organizations needed to reach an agreement. A trigger mechanism defined in SQL 99 is required to create digest tables for every block, which ensures tamper-proofing and consensus achievement. In the scenario where both the designated number of organizations and the individual organization share identical hashes, the individual organization proceeds to append a block to its own ledger.

Overall, the Whatever-Ledger Consensus model introduced by ChainifyDB offers a resolution to the problem of divergent interpretations of the SQL standard and discrepancies in data types encountered during the processing of transactions. It allows organizations and companies to execute transactions in a heterogeneous way and achieve consensus using a lightweight voting algorithm.

ChainifyDB consists of a distributed network comprising multiple servers operated by different organizations. The network architecture includes an OrderingServer, which is not considered fully trusted, along with a set of Kafka nodes responsible for broadcasting the blocks generated by the OrderingService. Organizations can operate on multiple servers, with a minimum requirement of one server for active participation. The servers in the network serve different functions such as ChainifyServer, AgreementServer, ExecutionServer, and ConsensusServer. Communication between ChainifyServer and AgreementServer facilitates the collection of necessary agreements for transactions, which are defined by policies specifying the required consensus among participating organizations. ExecutionServer handles the execution of transactions within received blocks from the OrderingServer, utilizing triggers to generate corresponding hashes. ConsensusServers rely on these hashes to establish communication among themselves.

4.2.4.1 Transaction Processing

ChainifyDB's transaction processing model, known as Whatever-Ledger Consensus, involves several servers that work together to process transactions. These servers include the ChainifyServer, AgreementServer, ExecutionServer, and ConsensusServer. The initial step involves the reception of the signed transaction by the ChainifyServer from the client, followed by its interaction with the AgreementServer to gather the necessary agreements. Upon successful completion, the OrderingServer proceeds to generate a block, which is subsequently transmitted to a Kafka node and subsequently received by all ExecutionServers. The ExecutionServers execute the transactions in the block, generate the corresponding block hash, and send it to the ConsensusServer.

The ConsensusServer is responsible for determining if a consensus has been achieved or not based on the received hash. If consensus is attained, the block is appended to the ledger. However, in the case that a consensus has not been achieved, a recovery process is initiated. This transaction processing model is designed to accommodate heterogeneous infrastructures, where different organisations execute transactions differently, leading to different interpretations of the SQL standard and the use of different data types. The Whatever-Ledger Consensus model allows each organisation to process a batch of inputs as they see fit in the whatever phase, with no assumptions made. This is followed by the ledger-consensus phase, where a consensus on the transactions is attempted using a lightweight voting algorithm.

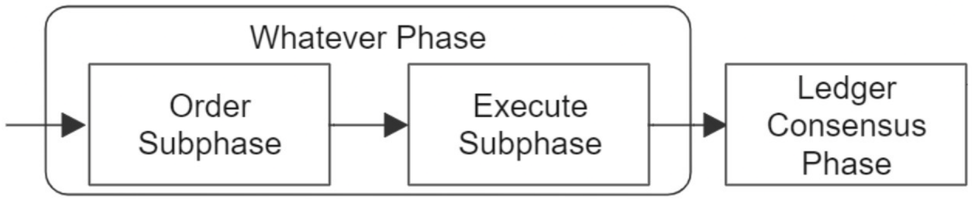


Figure 4.5: Whatever-Ledger Consensus transaction processing model

The default execution of transactions in ChainifyDB is sequential, which has limitations in terms of efficiency and performance. To address this issue, parallel execution of transactions is implemented. However, this approach requires a clear parallel model to avoid inconsistencies in the commit-order. ChainifyDB's parallel architecture encompasses numerous compact batches of transactions that can be securely executed concurrently. To ensure consistency, conflicts between transactions are identified during the execute subphase.

Whatever Phase: Order Subphase: In the Whatever Phase, the Order subphase is responsible for creating blocks by globally ordering and grouping input transactions in a batch. Unlike the OCM model, there is no subsequent round of consensus following the finalization of the ordering process.

Execute Subphase: During the execute subphase, each valid transaction within the block is locally executed in the database. However, unlike the OCE model, this execution is not guaranteed to be deterministic on all nodes.

Ledger Consensus Phase: The Ledger Consensus Phase is the stage where consensus is reached based on the effects produced in the Whatever Phase.

4.2.5 BlockchainDB

BlockchainDB [44] presents an innovative method for integrating blockchain technology into conventional data management methodologies. It involves the integration of blockchains as a fundamental layer for the purpose of data storage in combination with a database layer to enable data sharing. This approach ensures that native storage layers, such as existing blockchain technologies, can be used for decentralized storage, while shared tables can be accessed via the database layer. This architectural choice eliminates the need for nodes to store all data within the blockchain network locally. In contrast, the storage of shards is decentralized among a variable number of peers, which serves as a significant aspect observed in Ethereum 2.0 [45] and Rapidchain [46]. Leveraging data sharing does not compromise the security or fundamental principles of the blockchain. It offers advantages such as increased scalability and flexibility, as data can be distributed across a larger network of nodes. This approach allows for more efficient use of resources and a reduction in the overall costs associated with blockchain-based data management. Furthermore, it enables the development of new applications that leverage the power of blockchain technology without being constrained by its limitations. Overall, BlockchainDB represents a promising direction for the future of blockchain-based data management.

The BlockchainDB network consists of two types of participating nodes: full peers and thin peers. While thin peers do not store any shards and have limited resources, they can participate in the blockchain network through the shared table. Conversely, full peers undertake the responsibility of overseeing databases and retaining a minimum of one replica of a shard. Clients can interact with the BlockchainDB network through a limited put-get interface, which allows them to perform data retrieval and modification operations without requiring knowledge of the underlying blockchain technology. The database layer utilizes this interface, while the storage layer is responsible for storing all data on blockchains. Additionally, clients can verify the database layer's methods using off-chain verification techniques. This two-layer architecture ensures that nodes in the BlockchainDB network do not need to store all data locally, maintaining the network's scalability and security.

Database Layer: The Database Layer of BlockchainDB is positioned above the Storage Layer and provides a put-get interface to clients. The Shard Manager is responsible for locating the required shard of a table, which may be stored locally or remotely on another peer's storage, depending on the partitioning and replication schemes. In order to ensure consistent behaviour and handle concurrent transactions, the Transaction Manager works alongside the Shard Manager.

Storage Layer: In BlockchainDB, the storage layer is built on top of existing blockchain technology and is responsible for storing all data that interacts with the BlockchainDB. It generates and grants access to shards through a clearly defined interface that follows the key-value data model. These interfaces are designed to be blockchain-agnostic, and backend connectors are used to ensure stable and predefined interfaces for data management through smart contracts. To leverage the basic read and write functionalities, these smart contracts need to be installed within the BlockchainDB system. The storage layer allows for the concurrent processing of data across different blockchains, where each blockchain serves as a separate shard. Examples of supported blockchain technologies include Ethereum and Hyperledger Fabric.

4.2.5.1 Transaction Processing

The transaction model and consensus mechanism of BlockchainDB are dependent on the selected blockchain technology. Therefore, this section will provide an overview of the transaction path, starting from the client and passing through the database and storage layers, until the transaction is received by the chosen blockchain, presented in an abstract manner.

Database Layer: Within the database layer of BlockchainDB, clients engage in communication with their trusted peers by means of put and get requests. These requests are subsequently received and processed by the transaction manager. The manager handles transactions with a customizable consistency level, and the requests are first sent to the ShardManager, which determines the correct shard based on the provided key, before forwarding it to the storage layer.

Storage Layer: In the storage layer of BlockchainDB, the shard's location required for a given key is known to the system. Therefore, the storage layer has the capability to interface with the blockchain network and execute the required operations, such as read, write, or asynchronous operations, using the well-defined interface.

Offchain verification: Clients can ensure the correctness of their transactions in BlockchainDB by employing offchain verification. This can be done either online or offline. Online verification is conducted through the verify operation, which provides assurance to clients that the results of their requests are valid. On the other hand, offline verification is performed by deferring the verification of multiple operations until they can be executed simultaneously. By doing so, clients can avoid the overhead associated with verifying each request individually.

4.2.6 Hyperleder Fabric

Hyperledger Fabric [33], developed under the Linux Foundation's Hyperledger projects, is a blockchain framework which has gained popularity recently and is being used and tested by almost 400 organizations and companies. Fabric provides the capability to run distributed applications using widely-used programming languages such as Go, Java, and Node.js. This functionality allows applications to be executed on a globally distributed blockchain computer, effectively transforming it into a pioneering distributed operating system [47] for permissioned blockchains. Despite its intricacy, Fabric is versatile enough to be employed in both complex and straightforward applications [48].

Hyperledger Fabric is a versatile blockchain platform designed to solve a wide range of problems for organizations. Transactions are recorded on separate ledgers called channels, with access rights determined by a group of members. One of its most beneficial features is the ability to create complex networks, making it useful from a business perspective. Additionally, it has numerous pluggable options, including support for various Membership Service Providers (MSPs) and consensus mechanisms.

Hyperledger Fabric employs smart contracts to manage the business logic and manipulate the ledger's state, which stores both current and historical states. Administrators can create groups related to smart contracts using chaincode, which is run in Docker containers. The chaincode's local states can be accessed through the use of GetState, PutState, and DelState functions.

The Membership Service Provider (MSP) plays a crucial role in validating the identity and authorization of participants within a Hyperledger Fabric blockchain network. The network's nodes communicate with each other using the gRPC framework. A Hyperledger Fabric network comprises three distinct categories of nodes:

- **Clients** - Clients have the ability to submit transactions to peers for execution and subsequently forward them to the ordering service.
- **Peers** - Peers maintain a complete copy of the ledger and carry out transaction execution based on the specified policy.
- **Ordering Service Nodes (OSNs)** - The OSN has a singular responsibility of sequencing transactions. It does not participate in the validation and execution phase, nor does it possess any information about the application state.

Regarding data storage in the Hyperledger Fabric, the world state and the ledger play critical roles. At any given time, the world state reflects the current state of the ledger and functions as its underlying database. Typically, the world state database comprises two key-value store databases, with LevelDB as the default database. While CouchDB is capable of rich queries, it has greater opportunities for queries than LevelDB. When using CouchDB as the world state database, the ledger data is stored in the form of JSON. The transaction log is another essential component of the ledger, which records the pre and post-values stored in the ledger database being utilized by the blockchain network.

Hyperledger Fabric's consensus mechanism can be customized to meet the specific requirements of a blockchain network. Raft is the recommended crash fault-tolerant ordering service starting from version 2, offering a simpler setup compared to the previous Kafka-based ordering service. In Raft, leader nodes are chosen per channel and their decisions are replicated in follower nodes. The transaction model employed by Hyperledger Fabric differs from the conventional order-validate-execute model. Instead, it utilizes the execute-order-validate model, which shares similarities with the model utilized in the BRD. [49]

4.2.6.1 Transaction Processing

Fabric's transaction model differs from the traditional order-validate-execute model. Instead, as seen in Figure 4.6, it employs an execute-order-validate approach. This model involves three distinct stages that can be executed on different nodes within the blockchain network. In order to execute and verify the transaction's authenticity, the client first transmits it to the endorsing peers. The ordering service then arranges the transactions, after which validation based on predefined rules takes place.

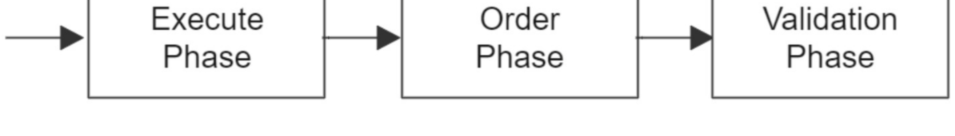


Figure 4.6: Execute-order-validate transaction processing model

Execute Phase: Clients submit their transactions to endorsing peers for independent execution, without the need for peer synchronization or communication. After the execution phase, each node can generate its own endorsement, which includes the read-write sets from the execution simulation. The client is responsible for gathering the required number of endorsements specified by the chaincode, and subsequently sending the transaction to the ordering service. [49]

Order Phase: The ordering service receives the transaction when the client has obtained the required number of endorsements, and consensus is reached on the transaction order. To improve efficiency, the ordering service can combine several transactions into a single block. [49]

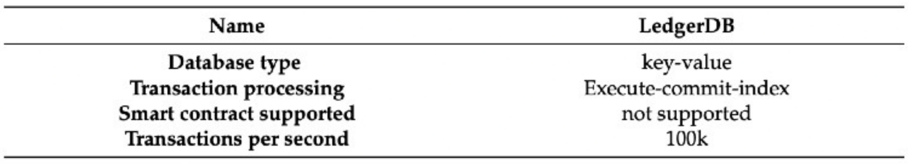
Validation Phase: After the ordering service has organized and grouped the transactions on a block, it is forwarded to the peers, which then validate it through three sequential steps. [49]

1. Firstly, the peers assess the validity of all transactions simultaneously according to the endorsement policy. Transactions that pass this evaluation are marked as valid and will have an effect on the database.
2. Secondly, the peers compare the keys in the current state of the ledger with those in the readset to determine if there are any read-write conflicts. Transactions that pass this check are still marked as valid, while those that fail become invalid.
3. Lastly, the blockchain state is modified based on the validation results, and the block is appended to the local ledger.

CHAPTER 5

Why FalconDB?

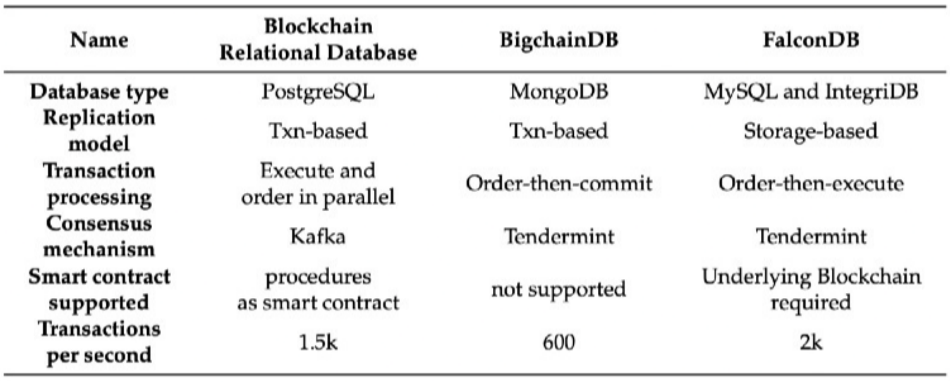
In all the ledger databases discussed above, one issue is that in most of them, key-value databases are more dominant than the SQL databases [49]. But, SQL databases being a standard, are an ideal choice for companies as they would not have to put in much effort in migrating to other databases as most of the modern databases are SQL compliant.



Name	LedgerDB
Database type	key-value
Transaction processing	Execute-commit-index
Smart contract supported	not supported
Transactions per second	100k

Table 5.1: Overview of discussed Centralised Ledger Technology

One major problem in the Centralised Ledger Databases (CLDs) is that trust is required between the participating entities. However, mutual trust is hard to assume in our problem where the participators are in the same field and are probably cut throat competitors of each other.

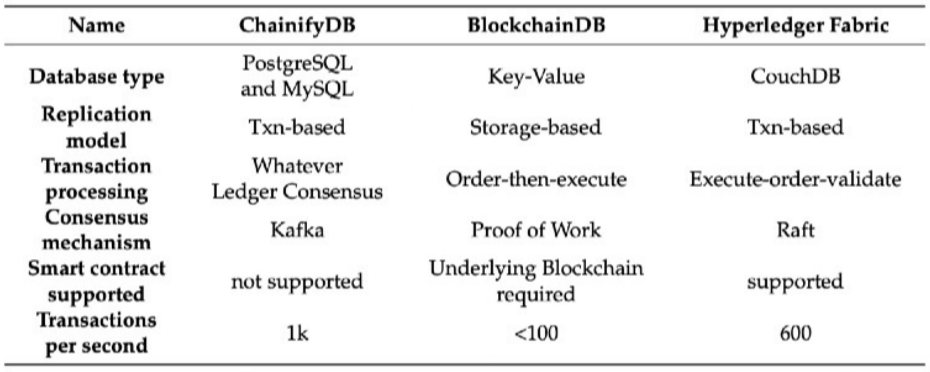


Name	Blockchain Relational Database	BigchainDB	FalconDB
Database type	PostgreSQL	MongoDB	MySQL and IntegriDB
Replication model	Txn-based	Txn-based	Storage-based
Transaction processing	Execute and order in parallel	Order-then-commit	Order-then-execute
Consensus mechanism	Kafka	Tendermint	Tendermint
Smart contract supported	procedures as smart contract	not supported	Underlying Blockchain required
Transactions per second	1.5k	600	2k

Table 5.2: Overview of discussed permissioned blockchain technologies

So, a blockchain-based database suits our needs better. Among the discussed blockchain-based databases, BlockchainDB follows the Proof of Work consensus, which is undesirable as it is a very time and energy-consuming process.

Some databases do not support smart contracts, which is also undesirable as no support for smart contracts implies that applications cannot be built on the blockchain.



Name	ChainifyDB	BlockchainDB	Hyperledger Fabric
Database type	PostgreSQL and MySQL	Key-Value	CouchDB
Replication model	Txn-based	Storage-based	Txn-based
Transaction processing	Whatever Ledger Consensus	Order-then-execute	Execute-order-validate
Consensus mechanism	Kafka	Proof of Work	Raft
Smart contract supported	not supported	Underlying Blockchain required	supported
Transactions per second	1k	<100	600

Table 5.3: Overview of discussed permissioned blockchain technologies

We then compared the transaction throughput (in terms of transactions per second) of the different blockchain-based databases. Finally, we arrived at the decision to adopt FalconDB's model to build our collaborative database.

CHAPTER 6

Preliminaries

A blockchain system usually comprises numerous nodes that lack trust among themselves. These nodes collaborate to maintain a global set of shared states and conduct blockchain transactions to change them. The blockchain nodes can reach an agreement on the transactions and their sequence despite the presence of potentially deceitful nodes, under the presumption that most of them are honest.

In a simplified version of the blockchain network, there is a chain of blocks, each consisting of a header H = (M, V) and block content C. The header comprises metadata M and verification data V, including the block height, the hash value of the previous block, and the hash value of the block content [41].

The initial block in a blockchain network is called the genesis block, and it has a fixed place in the protocol. Any other block (H, C) is considered valid only if the previous block (H', C') is valid and satisfies the below-mentioned conditions:

- the hash value of the previous block is equal to the lastBlockHash in the header of the current block;
- the height of the current block is one greater than that of the previous block;
- the hash of the content of the current block is equal to the dataHash in the header;
- pre-defined function called validate returns a value of 1.

To maintain stability, a functional blockchain system must incorporate verification data V and a validate function that prevent the arbitrary generation of blocks. Occasionally, multiple valid blocks may exist at the same height, leading to a fork. To ensure global state consistency among nodes, most blockchain protocols designate a primary branch as the main chain and ensure all nodes work on that chain.

There are two types of blockchain systems: permissionless and permissioned. The former allows any node to enter or exit the network, while the latter restricts access to a group of members only. All nodes undergo authentication, and their identities are shared and acknowledged by the other participating nodes in a permissioned blockchain. Given that FalconDB is designed as a collaborative database, it is presumed that the identities of all participants are established. Therefore, this report will primarily concentrate on permissioned blockchains.

6.1 InterPlanetary File System (IPFS)

The InterPlanetary File System (IPFS) [50] is a distributed file system that facilitates the storage and sharing of data across a peer-to-peer network. The IPFS protocol is designed to address the shortcomings of the traditional client-server model by creating a global namespace that uniquely identifies each file using content-addressing. In other words, files are not identified by their location, but by a hash of their content, which allows them to be stored and accessed from anywhere on the network. This makes IPFS a powerful tool for building decentralized applications that require large-scale data storage and sharing.

Unlike traditional file storage and sharing methods that rely on a central server, the InterPlanetary File System (IPFS) is a decentralized peer-to-peer network that allows users to both host and receive content similar to BitTorrent. IPFS is built on a distributed system where user-operators collectively store and share files, resulting in a robust and resilient network. Each user-operator holds a segment of the overall data, ensuring data redundancy. The content in IPFS is identified by its unique content address, and users within the network may serve files using the corresponding content address. Using a distributed hash table (DHT), any node containing the requested content can be discovered and approached by other peers in the network, enabling efficient and decentralized file sharing.

IPFS differs from BitTorrent because its objective is to establish a cohesive worldwide network implying that in the event of two users publishing a data block with identical hashes, the peers retrieving the content from the first user will share data with those retrieving it from the second user. Instead of using protocols for static webpage delivery, IPFS employs gateways that can be accessed via HTTP.

There are two types of IPFS networks - public and private. In a public IPFS network, anyone around the globe who has an IPFS connection can add the files to the network or access the files added to the network. In the private network, only the members of that particular network can add or access the files in that network.

A private IPFS network suits our purpose because the companies that are members of the consortium would not like to have their customer's data accessible to anyone outside the consortium.

CHAPTER 7

Summary of the Research Work

7.1 Design

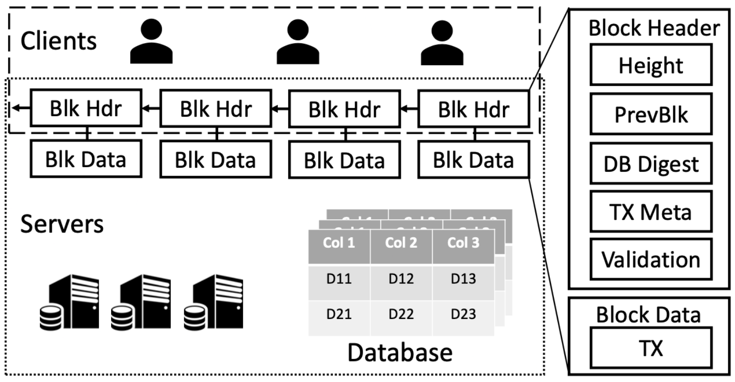


Figure 7.1: System overview

There are two types of entities in our design:

1. **Server nodes:** The blockchain data and database are maintained by server nodes, which are operated by various entities. These nodes are accountable for handling client requests and updates, verifying newly generated blockchain blocks, and producing evidence for query results.

2. **Client nodes:** The clients send requests to the server nodes and support the collaborative database. They have the ability to read or write certain portions of the database based on their access privileges. The client nodes do not store the blockchain or the database locally.

Both servers and clients maintain the blockchain network. Each block in the blockchain contains a transaction that involves one or more database reads or writes. Additionally, the ADS digest of the database version is included in the block header, which reflects the database content after the corresponding transaction in the block has been executed [41]. However, the ADS layer has not been added as part of our work. Implementing the ADS layer into the current design can be taken as part of the future development of the project.

In preparation for consensus, every server and client participating in the network deposits funds to a smart contract, which manages the incentive mechanism. In addition, all nodes in the network agree upon a privilege function. This function specifies the type of updates that a specific client is authorized to make and is adaptable. For instance, if a client becomes malicious, the others can revoke all of its access to the database [41]. A smart contract can easily implement this function, and we do not delve into the specifics in this report.

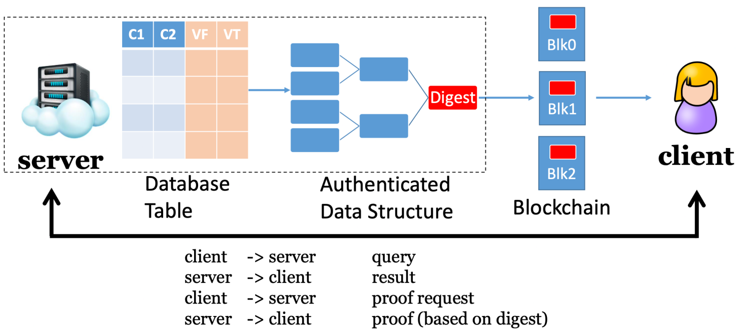


Figure 7.2: Simplified query workflow of FalconDB

The workflow for a query is depicted in Figure 7.2. A client node can initiate a query by transferring the query fee from its deposit to a server node and receiving the result in return. If the client is not satisfied with the result, it can challenge it by requesting an authentication from the smart contract. To do this, the client needs to pay an additional fee to the server. Then, the server is required to generate an ADS proof that can be validated by the digest. If the server fails to provide this proof, it will lose all fees in its smart contract account, including its initial deposit and revenue earned from clients. [41]

In addition, clients can use the ADS interface to send updates to the server. After communicating with the server, the client will suggest a new block to the network, which includes the update, interactive log, identity of the updater, and the new digest in the block header. The blockchain nodes will then validate the update's validity and the accuracy of the new digest. The block will be added to the blockchain once a consensus is reached among all blockchain nodes using the BFT consensus protocol. [41]

7.2 Role of IPFS

In situations like Know Your Customer (KYC) data, where scanned documents like Aadhar Cards, PAN Cards, passports, and driving licenses need to be stored, it can quickly become cumbersome to store these large image files on the blockchain. Also, storing large data on the blockchain makes it slow down. Therefore, it is necessary to store these files on a separate network that is decentralized and only stores their hashes in the blockchain transactions and database.

We have implemented the use of IPFS, a decentralized network that allows clients and servers to store and retrieve data, as a solution to this problem. With this approach, the clients can first store the scanned images on the IPFS network and then include the hashes returned by IPFS in the read or update queries submitted to the server instead of the image files themselves.

Also, we use a private IPFS network which consists of the members of the consortium only. This is essential as companies would like to minimize the risk of their customer's data becoming accessible to others.

7.3 Workflow

In this section, we will explain sequentially the steps that happen to execute a query. There are three major components that communicate with each other to get a query executed successfully - Tendermint Node, Application, and Database.

The application is developed by us internally and customized to our needs. Tendermint gives the flexibility to develop the application in the language of our choice. The application communicates with Tendermint Core through gRPC (Google Remote Procedure Call), a high-performance, language-agnostic framework for developing remote services. The gRPC user interface can be implemented using Protobuf (Protocol Buffers) to define the service interfaces and message types. Protobuf provides a language-agnostic and efficient way to define the structure of the data exchanged between the client and the server.

The application communicates with the MySQL database through JDBC (Java Database

Connectivity), an API that provides a standard way for Java applications to interact with databases. It allows developers to connect to a database, execute SQL queries, retrieve and manipulate data, and manage database transactions using Java code.

We use the SQL database in our system. It has entries to store a customer's name, Aadhar Card and Passport details. The Aadhar number field is set as the primary key of the database because it is unique for each citizen of India. The database has the following entries:

1. Name
    - Type - String
    - Stores the name of the customer
2. AadharNo
    - Type - Integer
    - Stores the Aadhar number of the customer
3. AadharHash
    - Type - String
    - Stores the hash that IPFS returns upon adding the scan of the customer's Aadhar card to the IPFS network
4. PassportNo
    - Type - Integer
    - Stores the Passport number of the customer
5. PassportHash
    - Type - String
    - Stores the hash that IPFS returns upon adding the scan of the customer's passport to the IPFS network

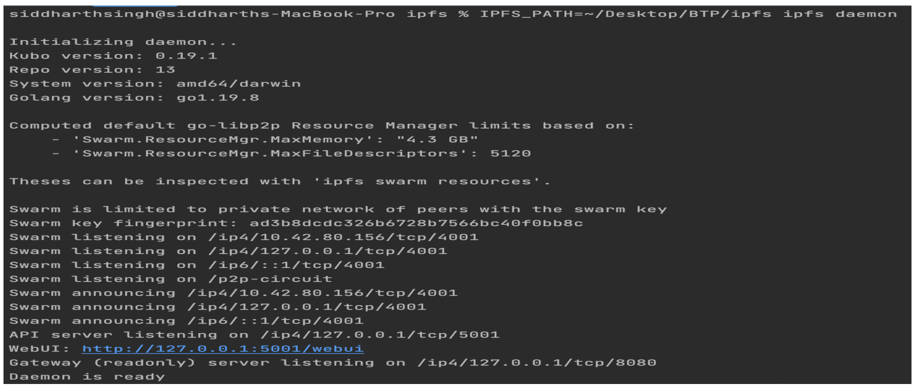
There are three types of queries in our system that the clients can issue to the server nodes - ADD, UPDATE, and DELETE.

1. **ADD**
    - Syntax: ADD Name AadharNo AadharHash PassportNo PassportHash
    - Semantics: Adds the entry of a user having the mentioned details in the database
2. **UPDATE**
    - Syntax: UPDATE AadharNo PassportNo PassportHash
    - Semantics: Updates the passport number and scan of the passport of the customer having the specified Aadhar Number
3. **DELETE**

- Syntax: DELETE AadharNo
- Semantics: Deletes the entry corresponding to the customer with the specified Aadhar Number

Let's follow the steps involved in adding a user's entry to the database.

1. Start the IPFS node.

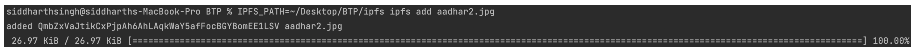


```
siddharthsingh@siddharths-MacBook-Pro ipfs % IPFS_PATH=~/Desktop/BTP/ipfs ipfs daemon

Initializing daemon...
Kubo version: 0.19.1
Repo version: 13
System version: amd64/darwin
Golang version: go1.19.8

Computed default go-libp2p Resource Manager limits based on:
    - 'Swarm.ResourceMgr.MaxMemory': "4.3 GB"
    - 'Swarm.ResourceMgr.MaxFileDescriptors': 5120

Theses can be inspected with 'ipfs swarm resources'.

Swarm is limited to private network of peers with the swarm key
Swarm key fingerprint: ad3b8dcdc326b6728b7566bc40f0bb8c
Swarm listening on /ip4/10.42.80.156/tcp/4001
Swarm listening on /ip4/127.0.0.1/tcp/4001
Swarm listening on /ip6/::1/tcp/4001
Swarm listening on /p2p-circuit
Swarm announcing /ip4/10.42.80.156/tcp/4001
Swarm announcing /ip4/127.0.0.1/tcp/4001
Swarm announcing /ip6/::1/tcp/4001
API server listening on /ip4/127.0.0.1/tcp/5001
WebUI: http://127.0.0.1:5001/webui
Gateway (readonly) server listening on /ip4/127.0.0.1/tcp/8080
Daemon is ready
```

Figure 7.3: IPFS node gets started

2. Add the scans of the Aadhar card and Passport to the (private) IPFS network. In case of an UPDATE query, you just have to add the Passport scan. This step is omitted in a DELETE query,

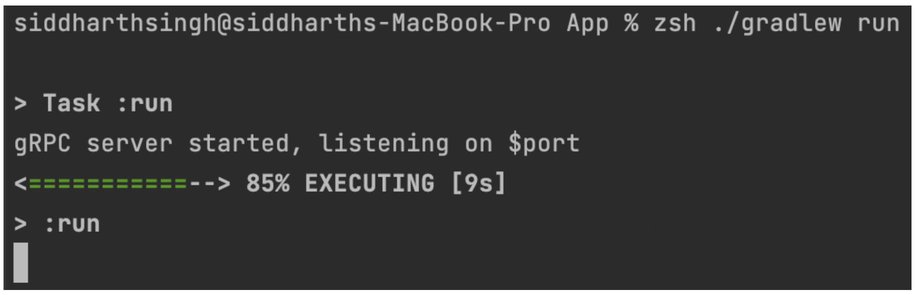
3. Above step will return the hashes of the scans uploaded.



```
siddharthsingh@siddharths-MacBook-Pro BTP % IPFS_PATH=~/Desktop/BTP/ipfs ipfs add aadhar2.jpg
added QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV aadhar2.jpg
 26.97 KiB / 26.97 KiB [========================================================================================================================] 100.00%
```

Figure 7.4: Add the Aadhar scan to IPFS network

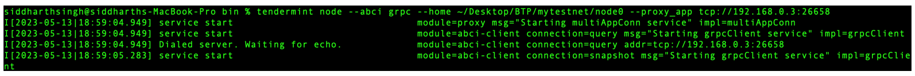
4. Start the application.



```
siddharthsingh@siddharths-MacBook-Pro App % zsh ./gradlew run

> Task :run
gRPC server started, listening on $port
<===========--> 85% EXECUTING [9s]
> :run
```

Figure 7.5: The custom application gets started

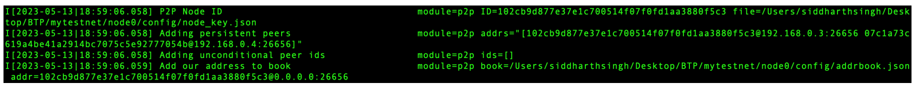
5. Start the Tendermint node.



```
siddharthsingh@siddharths-MacBook-Pro bin % tendermint node --abci grpc --home ~/Desktop/BTP/mytestnet/node0 --proxy_app tcp://192.168.0.3:26658
I[2023-05-13|18:59:04.949] service start                                module=proxy msg="Starting multiAppConn service" impl=multiAppConn
I[2023-05-13|18:59:04.949] service start                                module=abci-client connection=query msg="Starting grpcClient service" impl=grpcClient
I[2023-05-13|18:59:04.949] Dialed server. Waiting for echo.             module=abci-client connection=query addr=tcp://192.168.0.3:26658
I[2023-05-13|18:59:05.283] service start                                module=abci-client connection=snapshot msg="Starting grpcClient service" impl=grpcClie
nt
```

Figure 7.6: Tendermint node gets started

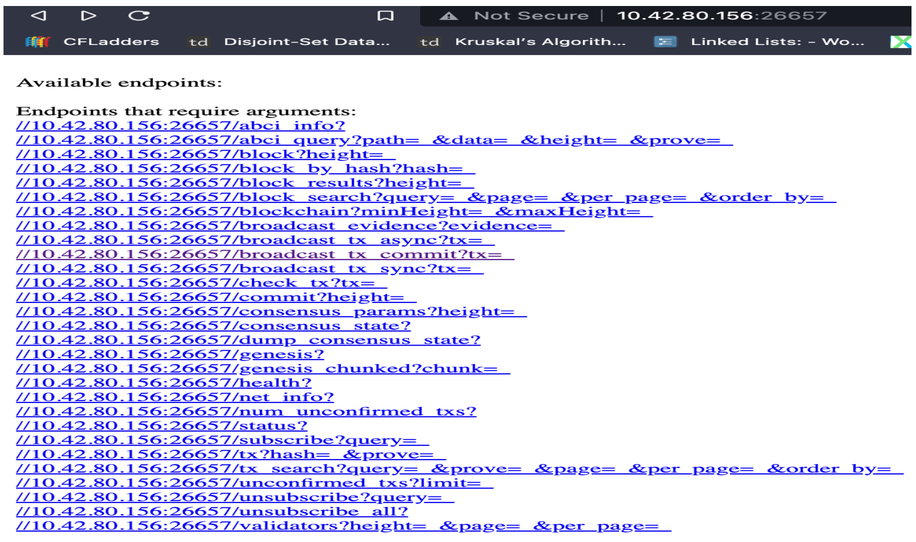
6. The Tendermint node dials the address of the peers mentioned in the config file and, upon getting a response from them, adds them to its address book. These peers participate in the consensus mechanism.



```
I[2023-05-13|18:59:06.058] P2P Node ID                                module=p2p ID=102cb9d877e37e1c700514f07f0fd1aa3880f5c3 file=/Users/siddharthsingh/Desk
top/BTP/mytestnet/node0/config/node_key.json
I[2023-05-13|18:59:06.058] Adding persistent peers                    module=p2p addrs="[102cb9d877e37e1c700514f07f0fd1aa3880f5c3@192.168.0.3:26656 07c1a73c
619a4be41a2914bc7075c5e92777054b@192.168.0.4:26656]"
I[2023-05-13|18:59:06.058] Adding unconditional peer ids              module=p2p ids=[]
I[2023-05-13|18:59:06.059] Add our address to book                    module=p2p book=/Users/siddharthsingh/Desktop/BTP/mytestnet/node0/config/addrbook.json
 addr=102cb9d877e37e1c700514f07f0fd1aa3880f5c3@0.0.0.0:26656
```

Figure 7.7: Peer nodes get added

7. Pass the query to the *broadcast_tx_commit* endpoint of the Tendermint node server. Below is the list of all endpoints. *broadcast_tx_commit* is marked in purple.

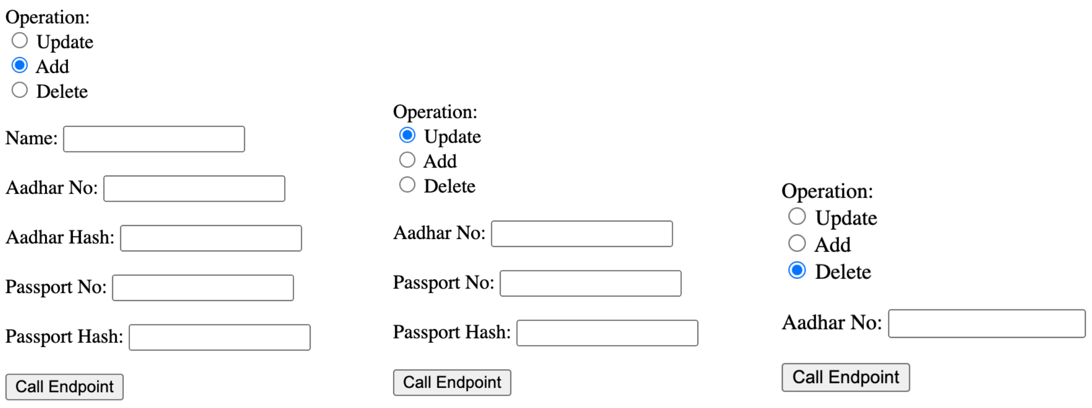


Available endpoints:

Endpoints that require arguments:
//10.42.80.156:26657/abci_info?
//10.42.80.156:26657/abci_query?path=_&data=_&height=_&prove=_
//10.42.80.156:26657/block?height=_
//10.42.80.156:26657/block_by_hash?hash=_
//10.42.80.156:26657/block_results?height=_
//10.42.80.156:26657/block_search?query=_&page=_&per_page=_&order_by=_
//10.42.80.156:26657/blockchain?minHeight=_&maxHeight=_
//10.42.80.156:26657/broadcast_evidence?evidence=_
//10.42.80.156:26657/broadcast_tx_async?tx=_
//10.42.80.156:26657/broadcast_tx_commit?tx=_
//10.42.80.156:26657/broadcast_tx_sync?tx=_
//10.42.80.156:26657/check_tx?tx=_
//10.42.80.156:26657/commit?height=_
//10.42.80.156:26657/consensus_params?height=_
//10.42.80.156:26657/consensus_state?
//10.42.80.156:26657/dump_consensus_state?
//10.42.80.156:26657/genesis?
//10.42.80.156:26657/genesis_chunked?chunk=_
//10.42.80.156:26657/health?
//10.42.80.156:26657/net_info?
//10.42.80.156:26657/num_unconfirmed_txs?
//10.42.80.156:26657/status?
//10.42.80.156:26657/subscribe?query=_
//10.42.80.156:26657/tx?hash=_&prove=_
//10.42.80.156:26657/tx_search?query=_&prove=_&page=_&per_page=_&order_by=_
//10.42.80.156:26657/unconfirmed_txs?limit=_
//10.42.80.156:26657/unsubscribe?query=_
//10.42.80.156:26657/unsubscribe_all?
//10.42.80.156:26657/validators?height=_&page=_&per_page=_

Figure 7.8: List of all endpoints provided by Tendermint

8. We also designed a frontend UI (User Interface) through which the client can issue the request to servers. Below are the snapshots of that.



Operation:
○ Update
● Add
○ Delete

Name:

Aadhar No:

Aadhar Hash:

Passport No:

Passport Hash:

(a) ADD

Operation:
● Update
○ Add
○ Delete

Aadhar No:

Passport No:

Passport Hash:

(b) UPDATE

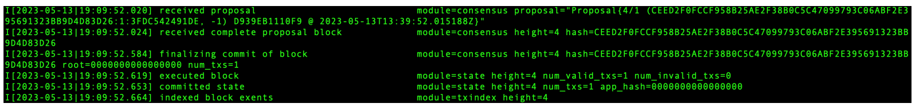
Operation:
○ Update
○ Add
● Delete

Aadhar No:

(c) DELETE

Figure 7.9: Frontend User Interface

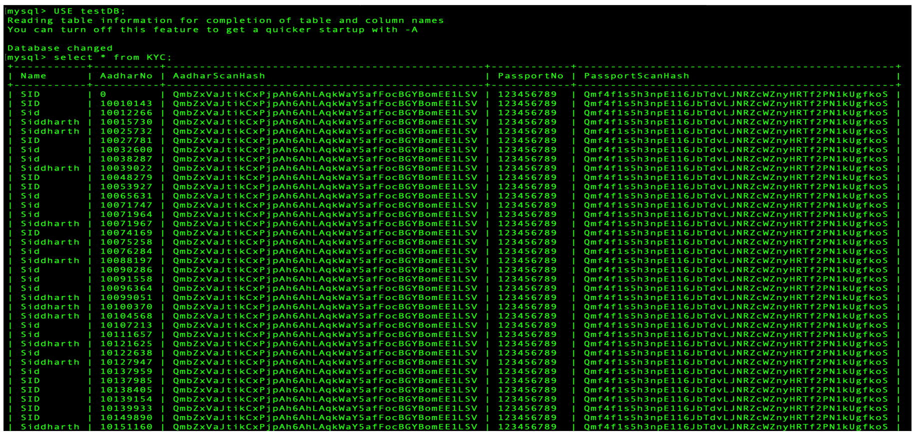
9. After the consensus algorithm terminates successfully, a consensus is achieved on the transaction, and it is committed to the blockchain.



```
I[2023-05-13|19:09:52.020] received proposal                         module=consensus proposal="Proposal{4/1 (CEED2F0FCCF958B25AE2F38B0C5C47099793C06ABF2E3
95691323BB9D4D83D26:1:3FDC542491DE, -1) D939EB1110F9 @ 2023-05-13T13:39:52.015188Z}"
I[2023-05-13|19:09:52.024] received complete proposal block          module=consensus height=4 hash=CEED2F0FCCF958B25AE2F38B0C5C47099793C06ABF2E395691323BB
9D4D83D26
I[2023-05-13|19:09:52.584] finalizing commit of block                module=consensus height=4 hash=CEED2F0FCCF958B25AE2F38B0C5C47099793C06ABF2E395691323BB
9D4D83D26 root=0000000000000000 num_txs=1
I[2023-05-13|19:09:52.619] executed block                            module=state height=4 num_valid_txs=1 num_invalid_txs=0
I[2023-05-13|19:09:52.653] committed state                           module=state height=4 num_txs=1 app_hash=0000000000000000
I[2023-05-13|19:09:52.664] indexed block exents                      module=txindex height=4
```

Figure 7.10: Transaction gets committed to the blockchain

10. Entry gets added to the database in the case of an ADD query, or it gets updated and deleted in case of UPDATE and DELETE query, respectively.



```
mysql> USE testDB;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from KYC;
+-----------+----------+------------------------------------------------+------------+------------------------------------------------+
| Name      | AadharNo | AadharScanHash                                 | PassportNo | PassportScanHash                               |
+-----------+----------+------------------------------------------------+------------+------------------------------------------------+
| SID       | 0        | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| SID       | 10010143 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Sid       | 10012266 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Siddharth | 10015730 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Siddharth | 10025732 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| SID       | 10027781 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Sid       | 10032600 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Sid       | 10038287 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Siddharth | 10039022 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| SID       | 10048279 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| SID       | 10053927 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Sid       | 10065631 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Sid       | 10071747 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Sid       | 10071964 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Siddharth | 10071967 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| SID       | 10074169 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Siddharth | 10075258 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Sid       | 10076284 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Siddharth | 10088197 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Sid       | 10090286 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Sid       | 10091558 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Sid       | 10096364 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Siddharth | 10099051 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Siddharth | 10100370 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Siddharth | 10104568 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Sid       | 10107213 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Sid       | 10111657 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Siddharth | 10121625 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Sid       | 10122638 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Siddharth | 10127947 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Sid       | 10137959 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| SID       | 10137985 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| SID       | 10138405 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| SID       | 10139154 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| SID       | 10139933 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| SID       | 10149890 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
| Siddharth | 10151160 | QmbZxVaJtikCxPjpAh6AhLAqkWaY5afFocBGYBomEE1LSV | 123456789  | Qmf4f1s5h3npE116JbTdvLJNRZcWZnyHRTf2PN1kUgfkoS |
```

Figure 7.11: Database gets updated

7.4 Analysis

The proposed FalconDB-based KYC system makes use of Tendermint as the underlying blockchain, MySQL as the database, and IPFS to store large image files such as the scans of Aadhar cards and passports. The system has various benefits which are as follows:

1. **Immutability and Auditability:** The integration of a blockchain platform such as Tendermint guarantees data immutability, establishing a high level of trust in the KYC system by preventing unauthorized alterations or tampering. Additionally, it facilitates transparent transaction auditing and enables comprehensive monitoring of data changes, enhancing overall accountability and transparency.

2. **Decentralization and Trust:** By leveraging the decentralized structure of the blockchain, the system eliminates the concentration of control in a single entity, promoting trust and confidence among participants. Through a distributed consensus model, transactions undergo validation and agreement from multiple network nodes, fostering enhanced trustworthiness and reliability in the ecosystem.

3. **Enhanced Security:** The utilization of blockchain technology incorporates strong security protocols through cryptographic algorithms and a distributed architecture. This ensures the safeguarding of data stored on the blockchain by employing

encryption techniques, effectively preventing unauthorized access. Furthermore, the adoption of IPFS for storing sensitive documents, such as Aadhar card images, adds an additional layer of security and privacy, reinforcing the overall security measures implemented within the system.

4. **Data Integrity:** The utilization of the blockchain's immutability guarantees the integrity of KYC data, as it becomes an indelible and verifiable record once stored. This fortifies the prevention of data manipulation, providing a trustworthy and irrefutable source of truth for KYC verification purposes.

5. **High Performance and Scalability:** The utilization of Tendermint's consensus algorithm, along with FalconDB and MySQL, delivers exceptional performance and scalability in terms of data storage and retrieval. This empowers the system to effectively handle a substantial volume of KYC records and transactions, ensuring efficient processing and management.

6. **Interoperability:** The system's architectural design allows seamless integration with external systems and applications by utilizing gRPC to communicate with the Tendermint Core. This fosters interoperability with diverse services or modules necessary for KYC processes, ensuring smooth communication and compatibility.

7. **Compliance with Regulations:** The system can be tailored to adhere to regulatory frameworks and data protection laws, incorporating features that enable compliance. Through the provision of an auditable trail of transactions and data modifications, the system aids in meeting regulatory obligations and showcases adherence to compliance requirements.

8. **Reducing Redundancy:** To enhance data storage efficiency, the system optimizes resource utilization by leveraging FalconDB and MySQL in conjunction with the blockchain. This involves storing non-sensitive or non-critical data in databases, effectively reducing redundancy and optimizing the utilization of resources.

9. **Flexibility and Customization:** The system's modular architecture offers flexibility and customization options tailored to specific business needs. It facilitates the seamless incorporation of new features, integration with external systems, and adaptation to evolving regulatory environments, ensuring the system remains adaptable and aligned with changing requirements.

The proposed system, despite its potential benefits, presents certain challenges and limitations that necessitate careful consideration. It is imperative to acknowledge and address these challenges and limitations to ensure the successful implementation and operation of the system. The following are notable challenges and limitations that require attention:

1. **Complexity:** The system's diverse components and technologies contribute to its inherent complexity throughout the development, integration, deployment, and

maintenance processes. Effectively managing and coordinating these components can present challenges, necessitating specialized expertise and dedicated resources for seamless operation and upkeep.

2. **Learning Curve:** The system comprises various technologies, including Tendermint, FalconDB, and IPFS, each with its own learning curve. Acquiring and sustaining the required skills and expertise for all components can present a challenge for both the development and operational teams, emphasizing the importance of continuous learning and skill development within the organization.

3. **Scalability:** Ensuring the system's ability to accommodate growing volumes of KYC records and transactions, while maintaining optimal scalability, can pose challenges. Successful management requires meticulous planning and configuration to sustain performance levels as the user base expands.

4. **Integration Challenges:** The integration of multiple components, each with unique protocols, data formats, or APIs, demands meticulous coordination and effective communication. Ensuring seamless interoperability among Tendermint, MySQL and IPFS presents notable challenges that require careful attention and comprehensive integration strategies.

5. **Performance and Latency:** System performance and latency can be influenced by various factors, including network congestion, blockchain size, and data retrieval efficiency. Guaranteeing satisfactory performance levels, particularly during periods of high usage, necessitates meticulous optimization and vigilant monitoring to ensure optimal system responsiveness.

6. **Security Risks:** Although blockchain technology provides inherent security features, it is not immune to potential risks such as smart contract vulnerabilities or attacks on the consensus algorithm. Proactive measures such as continuous monitoring, regular security audits, and timely updates play a crucial role in mitigating these risks and safeguarding the system's integrity.

7. **Data Privacy:** Protecting sensitive personal information in adherence to data protection regulations is of paramount importance. Implementing robust security measures, employing encryption techniques, and enforcing stringent access controls are essential components in safeguarding data privacy and ensuring comprehensive data protection.

8. **Adoption and User Experience:** The successful adoption of a new KYC system may necessitate user education and concerted efforts to promote acceptance. Delivering a streamlined and user-centric experience, which encompasses clear instructions and streamlined processes, plays a pivotal role in ensuring user satisfaction and fostering widespread adoption of the system.

9. **Cost Considerations:** When deploying and managing a system comprising multiple components, it is crucial to consider the associated costs. These may include licensing fees, infrastructure requirements, ongoing maintenance, and the need for skilled personnel. Striking a balance between the benefits gained and the costs incurred is paramount to ensure cost-effectiveness and maximize the system's overall value.

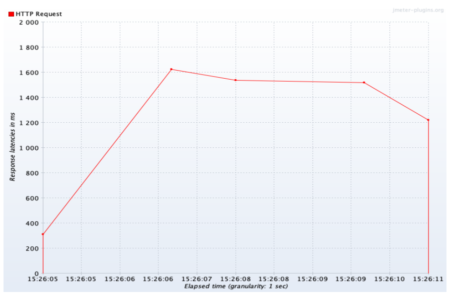
7.5 Experiments

We conducted the performance tests of the system using Apache JMeter. Apache JMeter is an open-source tool developed by the Apache Foundation. It is used to perform load tests and performance tests on a wide variety of software, although it was initially used to perform testing on web applications.

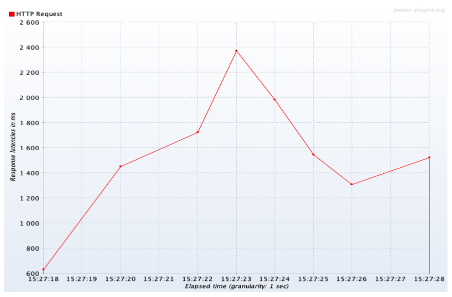
We tried different values for the number of threads during the experiments. The number of threads simulates the number of users trying to access the system concurrently.

Our setup involved two server nodes running on two different machines - one Mac and another Linux system. The Mac had a 2.3 GHz Dual-Core Intel Core i5 processor, and the Linux system had a 2.9 GHz Dual-Core Intel Core i7 processor. More systems can be seamlessly incorporated into our setup, but due to the constraint of the machines available to us, we performed the tests on two machines.

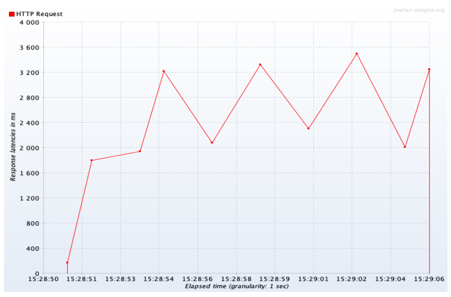
As part of our setup, only the servers participate in the consensus. Clients send their requests to the servers as an HTTP request. Clients do not store the blockchain's contents or the database locally.



(a) Number of threads = 1



(b) Number of threads = 50

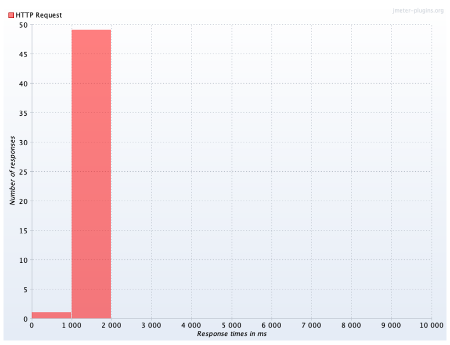


(c) Number of threads = 100

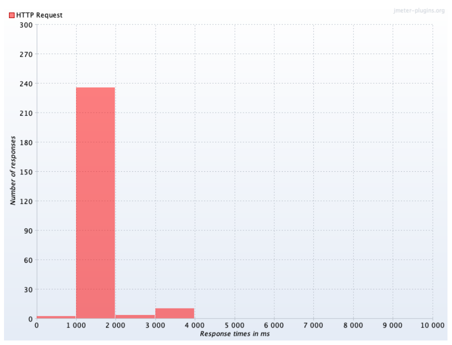
Figure 7.12: Latencies vs. Time

Figure 7.12 has plots of latencies as a function of time. We varied the number of threads and plotted the latencies with respect to time. The average latency increased as the number of threads increased. This is an expected behaviour as network congestion increases with more threads, which leads to more latency in fulfilling the requests.

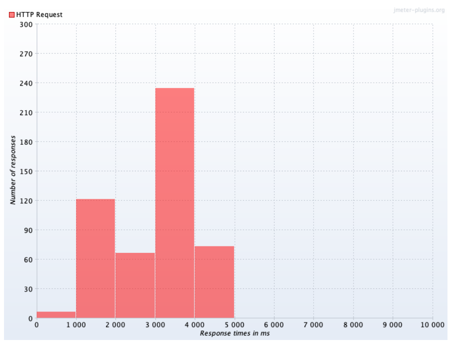
Also, when the number of threads is less there is lesser variability in latency. After a time, it almost remains constant. However, as the number of threads keeps on increasing, the variability in latency becomes more noticeable. When there are 100 concurrent threads requesting the server node for a query, the latency almost starts oscillating up and down.



(a) Number of threads = 1



(b) Number of threads = 50

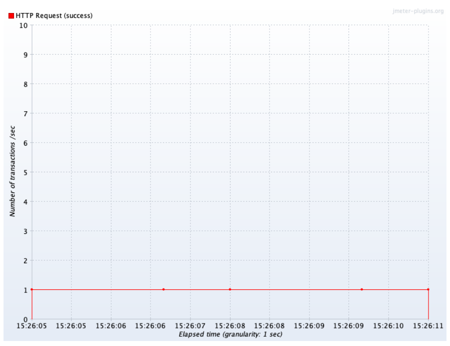


(c) Number of threads = 100

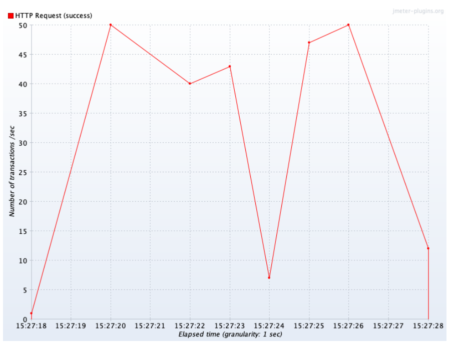
Figure 7.13: Response time distribution

The response times distribution are plotted in Figure 7.13. Again, we have experimented by varying the number of concurrent threads. Response time is slightly different from latency in the sense that response time is the total time it takes for a system or component to respond to a request. It includes the time required for processing the request, executing any necessary operations, and preparing the response whereas latency focuses specifically on the time it takes for a request to travel from the source to the destination and back. It represents the delay or time lag introduced by the network or communication channels during the transmission of the request and response.

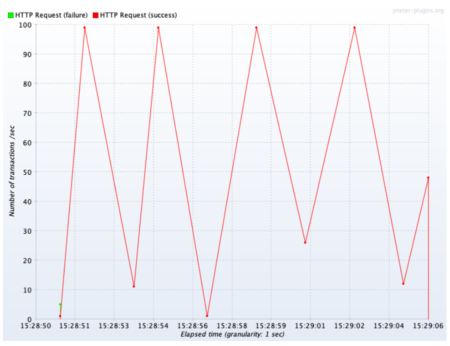
Similar to latency, response time also increased with increasing number of threads. As can be seen from the above plots, almost all of the requests took between 1 and 2 seconds to complete in the case of a single user making the requests, whereas as the number of threads got increased to 100, response time for the majority of requests shot up to 3-4 seconds.



(a) Number of threads = 1



(b) Number of threads = 50



(c) Number of threads = 100

Figure 7.14: Transactions per second

Plots in Figure 7.14 depict the throughput on the vertical axis and time on the horizontal axis. In the case of a single thread making requests to the server, we observe a constant throughput of 1 tps (transactions per second). As we increase the number of threads, the throughput also increases. In fact, the throughput can reach as high as the number of concurrent threads.

But the variability in throughput increases at a higher number of threads, so a relatively lower number of threads is preferred for applications that demand a comparatively constant throughput.

CHAPTER 8

Conclusion

In this thesis, we partially solved an industry problem of building and maintaining a collaborative database for a consortium of Fintech companies. We explored various design options available, including both the Centralised Ledger Databases like LedgerDB as well as the Permissioned Blockchain Based Databases like Blockchain Relational Database, BigchainDB, FalconDB, ChainifyDB, etc. We explained the designs of each of the databases in depth.

We then compared all of the above designs and finally chose and adopted FalconDB for our problem. We provided sufficient reason and argumentation for choosing FalconDB over others.

Then, we went ahead and explained about IPFS, another component used in our design. We explained the design of our system in detail, along with the role that IPFS plays in our design. We showed an example run of the ADD query along with bash commands and appropriate screenshots.

We strongly believe that this project has the capability to revolutionize the Fintech industry in the sense that more companies would want to come together and form a consortium to store their data in a collaborative manner. There is an incentive for both large-scale companies as well as small to medium-scale companies to form a consortium.

CHAPTER 9

Future Work

As we look towards the future, one of the immediate directions to pursue would be the implementation of the Authenticated Data Structures (ADS) layer. One promising open-source ADS that can be used is IntegriDB [51]. It is an open-source ADS that has been recommended in the original paper of FalconDB. The proof generation, communication and verification time do not depend on the database size in IntegriDB. Also, the digest could be implemented in $\mathcal{O}(1)$ space using IntegriDB.

Another area for future work is addressing the privacy issue posed by FalconDB. As a public transparent database, FalconDB makes the data accessible to all participants, including untrusted servers. However, exploring ways to query and update encrypted databases in a secure manner remains an open problem in academia. These areas are ripe for further research and development to enhance the security and privacy of our collaborative database system.

In the future, it is also worth exploring the utilization of alternative distributed ledger technologies (DLTs), such as Hedera [52], that offer enhanced transaction speeds, reduced latency, and efficient scalability. These DLTs have demonstrated notable improvements in these areas, which could potentially contribute to the advancement of various applications and systems. By considering the adoption and integration of such DLTs, organizations can aim to achieve higher transaction throughput, faster response times, and improved overall performance. Further research and experimentation are required to assess the feasibility and suitability of these DLTs in specific use cases, with the potential to unlock new possibilities and opportunities for innovation in the field. Expanding the investigation into these alternative DLTs can pave the way for future advancements in distributed systems and contribute to the ongoing evolution of decentralized technologies.

REFERENCES

[1] Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008.

[2] Ahram, T.; Sargolzaei, A.; Sargolzaei, S.; Daniels, J.; Amaba, B. Blockchain technology innovations. In Proceedings of the 2017 IEEE Technology & Engineering Management Conference (TEMSCON), Santa Clara, CA, USA, 8–10 June 2017.

[3] Risius, M.; Spohrer, K. A blockchain research framework. Bus. Inf. Syst. Eng. 2017, 59, 385–409.

[4] Grech, A.; Camilleri, A.F. Blockchain in Education; Publications Office of the European Union: Luxembourg, 2017.

[5] Agbo, C.C.; Mahmoud, Q.H.; Eklund, J.M. Blockchain technology in healthcare: A systematic review. In Healthcare; Multidisciplinary Digital Publishing Institute: Basel, Switzerland, 2019; Volume 7, p. 56.

[6] Tanwar, S.; Parekh, K.; Evans, R. Blockchain-based electronic healthcare record system for healthcare 4.0 applications. J. Inf. Secur. Appl. 2020, 50, 102407.

[7] Hyla, T.; Pejas, J. eHealth integrity model based on permissioned blockchain. Future Internet 2019, 11, 76.

[8] Hölbl, M.; Kompara, M.; Kamišalic , A.; Nemec Zlatolas, L. A systematic review of the use of blockchain in healthcare. Symmetry 2018, 10, 470.

[9] Bigini, G.; Freschi, V.; Lattanzi, E. A review on blockchain for the internet of medical things: Definitions, challenges, applications, and vision. Future Internet 2020, 12, 208.

[10] Tapscott, A.; Tapscott, D. How blockchain is changing finance. Harv. Bus. Rev. 2017, 1, 2–5.

[11] Treleaven, P.; Brown, R.G.; Yang, D. Blockchain technology in finance. Computer 2017, 50, 14–17.

[12] Shekhtman, L.; Waisbard, E. EngraveChain: A Blockchain-Based Tamper-Proof Distributed Log System. Future Internet 2021, 13, 143.

[13] Ibba, S.; Pinna, A.; Lunesu, M.I.; Marchesi, M.; Tonelli, R. Initial coin offerings and agile practices. Future Internet 2018, 10, 103.

[14] Cocco, L.; Pinna, A.; Marchesi, M. Banking on blockchain: Costs savings thanks to the blockchain technology. Future Internet 2017, 9, 25.

[15] Reyna, A.; Martín, C.; Chen, J.; Soler, E.; Díaz, M. On blockchain and its integration with IoT. Challenges and opportunities. Future Gener. Comput. Syst. 2018, 88, 173–190.

[16] Bellini, A.; Bellini, E.; Gherardelli, M.; Pirri, F. Enhancing IoT data dependability through a blockchain mirror model. Future Internet 2019, 11, 117.

[17] Tseng, L.; Yao, X.; Otoum, S.; Aloqaily, M.; Jararweh, Y. Blockchain-based database in an IoT environment: Challenges, opportunities, and analysis. Clust. Comput. 2020, 23, 2151–2165.

[18] Du, Y.; Wang, Z.; Leung, V. Blockchain-Enabled Edge Intelligence for IoT: Background, Emerging Trends and Open Issues. Future Internet 2021, 13, 48.

[19] Bouras, M.A.; Lu, Q.; Dhelim, S.; Ning, H. A Lightweight Blockchain-Based IoT Identity Management Approach. Future Internet 2021, 13, 24.

[20] Li, Y. An integrated platform for the internet of things based on an open source ecosystem. Future Internet 2018, 10, 105.

[21] Yang, X.; Zhang, Y.; Wang, S.; Yu, B.; Li, F.; Li, Y.; Yan, W. LedgerDB: A centralized ledger database for universal audit and verification. Proc. VLDB Endow. 2020, 13, 3138–3151.

[22] Mohan, C. State of public and private blockchains: Myths and reality. In Proceedings of the 2019 International Conference on Management of Data, Amsterdam, The Netherlands, 30 June–5 July 2019; pp. 404–411.

[23] Helliar, C.V.; Crawford, L.; Rocca, L.; Teodori, C.; Veneziani, M. Permissionless and permissioned blockchain diffusion. Int. J. Inf. Manag. 2020, 54, 102136.

[24] Zheng, W.; Zheng, Z.; Chen, X.; Dai, K.; Li, P.; Chen, R. Nutbaas: A blockchain-as-a-service platform. IEEE Access 2019, 7, 134422–134433.

[25] Singh, J.; Michels, J.D. Blockchain as a service (BaaS): Providers and trust. In Proceedings of the 2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), London, UK, 23–27 April 2018.

[26] Ruan, P.; Dinh, T.T.A.; Loghin, D.; Zhang, M.; Chen, G.; Lin, Q.; Ooi, B.C. Blockchains vs. Distributed Databases: Dichotomy and Fusion. In Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data, Xi'an, China, 20– 25 June 2021; pp. 543–557.

[27] Muzammal, M.; Qu, Q.; Nasrulin, B. Renovating blockchain with distributed databases: An open source system. Future Gener. Comput. Syst. 2019, 90, 105–117.

[28] Bergman, S.; Asplund, M.; Nadjm-Tehrani, S. Permissioned blockchains and distributed databases: A performance study. Concurr. Comput. Pract. Exp. 2020, 32, e5227.

[29] Raikwar, M.; Gligoroski, D.; Velinov, G. Trends in Development of Databases and Blockchain. In Proceedings of the 2020 Seventh International Conference on Software Defined Systems (SDS), Paris, France, 30 June–3 July 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 177–182.

[30] Gilbert, S.; Lynch, N. Perspectives on the CAP Theorem. Computer 2012, 45, 30–36.

[31] Zhang, K.; Jacobsen, H.A. Towards Dependable, Scalable, and Pervasive Distributed Ledgers with Blockchains. In Proceedings of the 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria, 2–5 July 2018.

[32] Nathan, S.; Govindarajan, C.; Saraf, A.; Sethi, M.; Jayachandran, P. Blockchain meets database: Design and implementation of a blockchain relational database. arXiv 2019, arXiv:1903.01919.

[33] Androulaki, E.; Barger, A.; Bortnikov, V.; Cachin, C.; Christidis, K.; De Caro, A.; Enyeart, D.; Ferris, C.; Laventman, G.; Manevich, Y.; et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. In Proceedings of the Thirteenth EuroSys Conference, Porto, Portugal, 23–26 April 2018; pp. 1–15.

[34] Berenson, H.; Bernstein, P.; Gray, J.; Melton, J.; O'Neil, E.; O'Neil, P. A critique of ANSI SQL isolation levels. arXiv 2007, arXiv:cs/0701157.

[35] Adya, A.; Liskov, B.; O'Neil, P. Generalized isolation level definitions. In Proceedings of the 16th International Conference on Data Engineering (Cat. No. 00CB37073), San Diego, CA, USA, 28 February–3 March 2000; IEEE: Piscataway, NJ, USA, 2000; pp. 67–78.

[36] McConaghy, T.; Marques, R.; Müller, A.; De Jonghe, D.; McConaghy, T.; McMullen, G.; Henderson, R.; Bellemare, S.; Granzotto, A. Bigchaindb: A Scalable Blockchain Database; White Paper, BigChainDB. 2016.

[37] Yiu, N.C. Decentralizing Supply Chain Anti-Counterfeiting Systems Using Blockchain Technology. arXiv 2021, arXiv:2102.01456.

[38] Rejeb, A.; Keogh, J.G.; Treiblmaier, H. Leveraging the internet of things and blockchain technology in supply chain management. Future Internet 2019, 11, 161.

[39] Buchman, E. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains. Ph.D. Thesis, University of Guelph, Guelph, Canada, June 2016.

[40] Lamport, L.; Shostak, R.; Pease, M. The Byzantine generals problem. In Concurrency: The Works of Leslie Lamport; ACM: New York, NY, USA, 2019; pp. 203–226.

[41] Peng, Y.; Du, M.; Li, F.; Cheng, R.; Song, D. FalconDB: Blockchain-based collaborative database. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, Portland, OR, USA, 14–19 June 2020; pp. 637–652.

[42] Martel, C.; Nuckolls, G.; Devanbu, P.; Gertz, M.; Kwong, A.; Stubblebine, S.G. A general model for authenticated data structures. Algorithmica 2004, 39, 21–41.

[43] Schuhknecht, F.M.; Sharma, A.; Dittrich, J.; Agrawal, D. Chainifydb: How to blockchainify any data management system. arXiv 2019, arXiv:1912.04820.

[44] El-Hindi, M.; Binnig, C.; Arasu, A.; Kossmann, D.; Ramamurthy, R. BlockchainDB: A shared database on blockchains. Proc. VLDB Endow. 2019, 12, 1597–1609.

[45] Ethereum 2.0. 2021.

[46] Zamani, M.; Movahedi, M.; Raykova, M. Rapidchain: Scaling blockchain via full sharding. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 931–948.

[47] Tanenbaum, A.S. Distributed operating systems anno 1992. What have we learned so far? Distrib. Syst. Eng. 1993, 1, 3.

[48] Lin, J.J.; Lee, Y.T.; Wu, J.L. The Effect of Thickness-Based Dynamic Matching Mechanism on a Hyperledger Fabric-Based TimeBank System. Future Internet 2021, 13, 65.

[49] Fekete, D.L.; Kiss, A. A Survey of Ledger Technology-Based Databases. Future Internet 2021, 13, 197.

[50] Juan Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. Technical Report Draft 3. IPFS.

[51] Y. Zhang, J. Katz, and C. Papamanthou. IntegriDB: Verifiable SQL for Outsourced Databases. In ACM Conference on Computer and Communications Security, pages 1480–1491. ACM, 2015.

[52] Baird, Leemon. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep 34 (2016): 9-11.