THESIS

STREAMLINING DECENTRALIZED LEDGER: ENHANCING HISTORICAL DATA ACCESS IN HYPERLEDGER FABRIC

Submitted by

Andrei Bachinin

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2024

Master's Committee:

Advisor: Indrakshi Ray

Yashwant K. Malaiya
Indrajit Ray
Steven J. Simske

ABSTRACT

STREAMLINING DECENTRALIZED LEDGER: ENHANCING HISTORICAL DATA ACCESS IN HYPERLEDGER FABRIC

This thesis presents design and implementation of Indexing Solution (`IS`) that aims to enhance query retrieval process in Hyperledger Fabric (*HLF*). In order to address limitations of *HLF*, we introduce new indexing algorithms: the Version-Based Index (`VBI`) and the Block-Based Index (`BBI`). We also introduce several previously unsupported query APIs: GetHistoryForVersionRange (*GHFVR*), GetHistoryForBlockRange (*GHVBR*), GetHistoryForKeyRange (*GHFKR*). All the work we propose is designed to integrate and work seamlessly with *HLF*, ensuring full backward compatibility with existing architecture.

Our experiments demonstrates that proposed solution significantly outperforms original *HLF* regarding query execution time, reducing it from *6.621 seconds* to *0.019 seconds* for certain queries. While `VBI` and `BBI` introduce a space index overhead, it remains comparable to the space overhead shown in *HLF*. Furthermore, we adopt parallel data insertion that mitigates a slower data insertion observed in both `VBI` and `BBI`. Comparison with traditional database systems (*LevelDB* and *MySQL*) and other blockchain solution from literature (*Lineage Chain* and *vChain+*) highlights significant advantage of our `IS` in terms of querying capabilities, paving the way for broader application of *HLF*.

ACKNOWLEDGEMENTS

I am sincerely grateful to my advisor, Dr. Indrakshi Ray, for her continuous guidance and support. Without her supervision, this thesis would not have been completed. I also want to thank Dr. Yashwant Malaiya, Dr. Indrajit Ray, and Dr. Steven Simske for agreeing to serve on my thesis committee.

I would also like to extend my gratitude to the Computer Science Department at Colorado State University for providing me with the incredible opportunity to study and learn in such intellectually stimulating environment. The resources and support from the department have been invaluable to my academic and personal growth.

This research was supported in part by funding from the State of Colorado under SB 18-086, funds from NIST under Award Number 60NANB23D152, funds from NSF under Award Numbers DMS 2123761, CNS 2335687, CNS 1822118, NIST, ARL, Statnett, AMI, NewPush, and Cyber Risk Research.

I must express my deepest gratitude to my family and parents, who have been my constant support throughout this journey. Despite being thousands of miles away, their unwavering faith in me and endless encouragement have been my strength. Their sacrifices have not gone unnoticed, and I owe a significant part of my achievements to their love and support.

DEDICATION

Dedicated to my family, honoring both the paths they have forged and the journeys yet to come.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Distributed Ledger Technologies (*DLTs*) (for example blockchain) allow to conduct transactions in a secure and verifiable way without the necessity of a trusted intermediary agent. It is acknowledged that *DLT* is widely used in various industries, including finance, real estate, public administration, energy and transportation [1].

Blockchain introduces new features to the businesses as well as industrial sectors [2]. It is able to enhance, optimize, secure and streamline various existing processes. Furthermore, it promotes an emergence of new business models, which were previously impossible. Such models as well as rapid development of Internet have influenced numerous sectors, including healthcare, manufacturing and logistics. However, secure registration and validation of agreements among parties involved in business transaction is still a concern. Blockchain technology addresses this concern, allowing businesses and individuals to record and secure their agreements autonomously.

By incorporating various methods and techniques, *DLTs* maintain a shared ledger among various users, ensuring consensus on its content and security of transactions. Once transaction is submitted to the ledger, it cannot be changed. Blockchain enables entities to reach consensus on specific activities and register these agreements without any additional regulatory authority. Such activities might include payment transactions, voting processes, patients medical lab reports and many more.

DLT combines the benefits of both peer-to-peer networks and cryptographic algorithms to validate agreements. No participating entity can modify an already approved and registered activity without the consensus of all involved parties. This property is specifically important for various business agreements among entities from different locations. Furthermore, blockchain preserves chronological order of events, ensuring accuracy of transaction log over time. With all these properties, it becomes nearly impossible to falsify record or deny an existing agreement. As a result, variety of industries, businesses and research institutions are exploring *DLTs*.

1.1 Necessity and Challenges

In this thesis we focus on Hyperledger Fabric (*HLF*) [3], a popular blockchain platform maintained be the Linux Foundation [4]. *HLF* stands out as an open-source platform adjusted for enterprise-grade business applications. It has a modular architecture and strong performance capabilities of processing 3,500 transactions per second (TPS) [5]. This is a significantly better result compared to *Ethereum* [6], another renowned blockchain platform, which manages only about 25 TPS. *HLF* is used across multiple industries, including supply chain management by Walmart [7], procurement management by Hitachi [8], Internet of Things (IoT) implementations by Bosch [9], manufacturing processes by Honeywell [10], and land registry by Tech Mahindra [11].

Despite its wide adoption, *HLF* and *DLTs* in general encounter several challenges:

- ***Challenge 1 (C1).*** Current decentralized ledger infrastructures are burdened by a limited variety of supported queries. For example, *HLF*, which supports only a single query Application Programming Interface (API) focusing on transaction history with **GetHistoryForKey (GHFK)** [12]. To illustrate this limitation, consider a healthcare application where an end-user needs to access specific types of a patient's historical health records, such as all immunization records. However, due to the limited querying capabilities of *HLF*, an end-user would need to wait and retrieve all transactions related to the patient and then manually filter out the required records.

- ***Challenge 2 (C2).*** The process of retrieving a specific historical asset record requires traversing of the entire transaction history associated with the queried key. It leads to undesired query retrieval performance. For example, the time required to query one million Ethereum data records (see §6.1 for details on evaluation datasets) varies significantly across different platforms, including traditional databases and *DLTs*. It takes 6 seconds in *HLF*, 3.12 seconds in *Ethereum*, 0.003 seconds in *LevelDB*, and 0.195 seconds in *MySQL* (see §6.5 and §6.6 for details). This challenge can be exemplified in the financial sector where a bank may

need to quickly access transaction records for auditing purposes. The slow retrieval times can significantly slow down operational efficiency.

This thesis proposes *IS* - an Indexing Solution, which addresses both challenges by offering an efficient solution for improved query processing and data retrieval. *IS* is designed for seamless and easy integration with *HLF* platform.

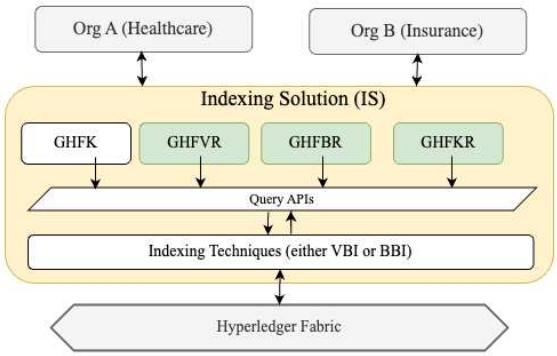

Figure 1.1: Architecture of Proposed Indexing Solution.

Consider a scenario involving two interrelated organizations (e.g., *Org A*, a healthcare provider and *Org B*, an insurance provider) utilizing the same *DLT*, as depicted in Figure 1.1. *Org A* employs *HLF* and maintains a set of ledgers for various data operations, such as diagnosis and treatment records. *Org B* uses the same instance of *HLF*, organizing its data into separate ledgers for registration, insurance claims, and more. Both organizations require a robust data retrieval mechanism for their business operations. However, the limited querying capabilities of *HLF* slow down their use as large-scale analytical systems.

This is where our proposed *IS* fits in as an efficient data querying solution that enhances *DLT* functionalities. Proposed solution offers several benefits. (i) enhance the speed of query retrieval in *HLF* by utilizing new efficient indexing methods — *Block-Based Index (BBI)* and *Version-Based Index (VBI)* (will be discussed in §4.3). (ii) expand the variety of supported queries: **GetHistoryForVersionRange (GHFVR)**, **GetHistoryForBlockRange (GHFBR)**, **GetHistoryForKeyRange (GHFKR)** (see §4.4 for details).

By implementing the proposed solution, both *Org A* (healthcare provider) and *Org B* (insurance provider) enable successful query execution even with millions of historical data records, which was not achievable with the original *HLF*. Furthermore, both indexing methods (`VBI` and `BBI`) enable new query types, which allow querying specific historical records or ranges of records for individual customers or patients. They also support queries for historical records with a minimum number of updates within a specified block range, where a block range might be considered as a time range due to the blockchain nature. Furthermore, each indexing technique optimizes different types of queries. `VBI` is more suitable for queries that target specific historical versions or ranges of versions. `BBI` excels in block-related queries. In addition, it should be noted that combining both `VBI` and `BBI` within the same ledger is not feasible. Given this distinctions, *Org A* and *Org B* should consider dividing the data across multiple ledgers, where each ledger is optimized with its own indexing technique.

1.2 Thesis Organization

Chapter 2 reviews multiple research studies within the relevant fields, establishing a connection to our work. Chapter 3 introduces essential concepts for our research. This chapter provides an in-depth introduction to the internal indexing mechanisms of *HLF* and serves as a basis for our proposed indexing methods. Chapter 4 provides details regarding our proposed `IS` and new querying APIs. Chapter 5 explores low-level design and implementation details. It offers a better understanding of how our proposal goes into practical application. Chapter 6 evaluates our proposed `IS` against other blockchain and non-blockchain systems through a series of experiments.

Chapter 2

Related Work

In this thesis, we introduce `IS`, a solution designed to address several challenges associated with historical data retrieval in the *DLT* field. These challenges are limited variety of supported queries and poor query retrieval performance. Previously in academia, the choice between *DLT* and Distributed Database Systems (*DDBS*) got considerable debate [13]. We conduct a theoretical exploration of the capabilities for both *DLTs* and *DDBS*. Furthermore, we review several well-known research studies that address similar challenges in *DLTs*.

2.1 Distributed Databases

Distributed databases are the systems where storage devices are not all attached to common processor. Instead, the database is managed by central Database Management System (*DBMS*), but stored across multiple interconnected sites. It allows the database to be accessed and modified from different geographical locations. Such architecture improves data availability, reliability, and scalability. *Apache Cassandra* [14], *MongoDB* [15], *Couchbase* [16] and *Google Spanner* [17] are examples of distributed databases. Every one of them is designed to meet specific needs in terms of scalability, consistency and availability, following the principles of the *CAP* theorem (Consistency, Availability, Partition tolerance) [18]. Let's describe their capabilities.

Apache Cassandra is famous for its robust performance in big data environment, offering high availability, linear scalability and fault tolerance [19], [20], [21]. It uses a masterless ring architecture that allows seamless operation even in the case of data center outage. It also supports asynchronous replication across data centers [20], [22]. *Cassandra's* data model is flexible, allowing a schema-free approach with different rows having varying columns and columns data types can be altered after the table is created [23], [24]. Cassandra Query Language (*CQL*) provides tools to execute *CRUD* (perform create, read, update and delete) operations in various programming languages [21]. Research work has focused on developing methods and tools for efficient

data modeling, along with resource management systems to optimize performance and resource utilization in *Cassandra* deployments [19], [22], [25].

MongoDB is a flexible NoSQL database, which gained popularity for its document-oriented storage model and scalability. It is designed to handle large volumes of data, making it an appropriate choice for various applications. From social networking to data warehousing, *MongoDB* provides adjustable consistency levels, allowing developers to balance latency and consistency based on the application requirements [26]. The efforts have been made to test and verify causal consistency implementation, which is important for maintaining the order of operations in distributed system [27]. *MongoDB* highlights the simplicity and ease of use as a key benefits for developers [28]. *MongoDB's* query language, *MQL*, is formally analyzed for its expressivity and computational complexity, showing equivalence with nested relational algebra for well-typed fragments [29]. Log management is essential for security and performance, with profiling and *mongosniff* (*mongorepplay* for *MongoDB 3.4*) being useful techniques for this purpose [30]. MongoDB's performance effectiveness is associated with its in-memory computing and document-oriented storage, which uses binary JSON (*BSON*) for efficient data storage and transfer [31]. Data migration process from relational databases to *MongoDB* is well explored, highlighting system's flexibility and compatibility [32].

Couchbase represents a modern database management system, which has been designed to meet the requirements of web-based, document-oriented, and distributed applications. It represents a shift to more flexible, scalable, and schema-less data storage solutions. *Couchbase* is a document-oriented database. It supports efficient storage and retrieval of JSON documents. It supports complex queries such as ad hoc joins and aggregations and it is suitable for scalable, distributed environments [33], [34], [35]. *Couchbase* is also characterized by a key-value store with managed cache, indexing, and a powerful query engine, making it appropriate for large volumes of data and high-throughput applications [33]. Its distributed architecture is based on a shared-nothing model, which is essential for horizontal scalability and high availability. Such approach distinguish this database from traditional relational databases and some NoSQL alternatives [36]. *Couchbase*

demonstrates decent *CRUD* operation performance in distributed environments [37]. The database is recognized for its ease of use, making it a practical choice for developers and administrators.

As demonstrated in the above systems descriptions, each *DDBS* system is designed for specific operational task. Performance metrics for these distributed databases vary based on the configuration, use case, and hardware. For example, a benchmark conducted by *MongoDB* [38] evaluates performance of three leading solutions: *Cassandra*, *Couchbase*, and *MongoDB*. This study uses an industry benchmark created by *Yahoo!* called *YCSB* [39]. Several workloads were tested, including *Workload A* (a mix of 50/50 reads and writes) and *Workload B* (a 95/5 reads/writes mix). In the case of *Workload A*, *MongoDB* demonstrated 160,719 operations per second (ops/sec), *Cassandra* achieved 134,839 ops/sec, and *Couchbase* reached 106,638 ops/sec. In case of *Workload B*, *Couchbase* achieved 187,798 ops/sec, *Cassandra* showed 144,455 ops/sec, and *MongoDB* achieved 196,498 ops/sec. However, despite their robust performance, security, and flexibility, these databases lack the inherent security and trust model that blockchain technologies provide. This gap is addressed by our proposed `IS`, which aims to integrate high-performance query processing within the secure and trustworthy environment of *DLT* platforms.

2.2 DLT Frameworks

Decentralized ledger platforms can be used to maintain a transparent and traceable transaction history for a variety of use cases. Some examples of decentralized ledger platforms are *HLF*, *Ethereum*, *Quorum* [40], *Bitcoin* [41]. Let's consider *HLF* and *Ethereum* more closely.

To the best of our knowledge, only a few studies have experimentally evaluated *Ethereum's* performance. Specifically, the research conducted by *Aldweesh et al., 2018* [42] involved deploying private *Ethereum* networks utilizing both Parity [43] and Geth (Go-Ethereum) [44] clients with various consensus algorithms. The findings showed that Parity outperforms Geth in handling concurrent transactions. However, these results may not accurately represent the performance of *Ethereum's* main network. *Pongnumkul et al., 2017* [45] conducted a comparative evaluation of *HLF* and *Ethereum* by establishing private networks. The experimental outcomes suggested that

although *Ethereum* can manage a greater number of concurrent transactions, it struggles to simultaneously achieve high throughput and low latency. However, this study did not explore the reasons behind *Ethereum's* performance instability. Furthermore, test results from private networks may not accurately reflect the main network's performance. *Xu et al., 2017* [46] proposed a taxonomy method for comparing blockchains and blockchain-based systems, helping in the design and evaluation of their impact on software architectures. More recently, *Maple et al., 2019* [47] introduced a framework for describing a generic blockchain structure, including a range of aspects from permissions to consensus mechanisms. This framework serves as a reference for assessing architecture of the blockchain solutions, facilitating the design and implementation of business logic. In practice, *Ethereum's* block gas limit is typically set to 7,999,992 with each transaction consuming 21,000 gas in the absence of smart contracts. This configuration implies that Ethereum can process approximately 380 simple transactions per block. Given that blocks are generated roughly every 14.37 seconds, the estimated transaction throughput is approximately 25.346 TPS.

In terms of *HLF*, *Pongnumkul et al., 2017* [45] presents the performance analysis, comparing the Go implementation of the *Ethereum* client with *HLF*. Subsequently, *Dinh et al., 2018* [48] established a standardized performance analysis framework for decentralized ledger platforms by introducing *Blockbench* [49] benchmark framework. Further studies mostly utilized *HLF* v1.0. In comparison to the results of *Dinh et al., 2018* [48], which achieved approximately 1,000 transactions per second, *Androulaki et al., 2018* [50] showcased significantly higher performance metrics with the introduction of the v1.x architecture. They conducted an analysis of the preliminary v1.1 and demonstrated that *HLF* is capable of supporting over 100 peers. Furthermore, under optimal conditions, it can process more than 3,500 transactions per second. However, the performance outcomes reported by *Baliga et al., 2018* [51] were noticeably lower than those of *Androulaki et al., 2018* [50]. It demonstrated that *HLF's* potential performance is influenced by a variety of factors, including the benchmark framework, the version of the software, and the hardware used. As a result, later research expanded the scope of testing to include multiple parameters and more recent versions of *HLF*. Specifically, *Kuzlu et al., 2019* [52] and *Thakkar et al., 2021* [53] examined

v1.4 of *HLF*. Their findings further underscore the complexity of blockchain performance testing. Notably, *Kuzlu et al., 2019* [52] determined that performance is significantly affected not only by the specific infrastructure hosting the blockchain but also by the design of the transactions, including their type and number. More recent studies have focused on *HLF* v2.0, with *Dreyer et al., 2020* [54] conducting the initial performance measurements of *HLF* v2.0 and revealing substantial improvements over previous versions. Furthermore, *Toumia et al., 2021* [55] evaluated the performance of *HLF* v2.2. However, these studies do not provide comprehensive and comparable results for versions v1.x and v2.x due to variations in the testing conditions. Thus, while the following studies have introduced additional influencing factors, the findings of *Androulaki et al., 2018* [50] and *Thakkar et al., 2018* [56] remain the most exhaustive. *Gorenflo et al., 2019* [57] indicates that *HLF* is theoretically capable of achieving up to 20,000 TPS.

Our proposed `IS` does not specifically target enhancements in transaction throughput. It focuses on improving query performance and expanding the diversity of supported queries, unlike the original implementations of *HLF*, which naturally support only a retrieval of key history [12]. The following section explores related work that has similar objectives.

2.3 Previous Work

Lineage Chain [58] is implemented on top of *HLF* with *ForkBase* [59] as the storage. It enhances data provenance in blockchain systems with features such as provenance capture, Merkle tree storage, and a skip list index for efficient queries. *Singh et al., 2023* [60] introduces indexing models to further enhance blockchain data management by addressing multi-dimensional and historical data challenges. Specifically, the Two-tier Deterministic Appended Only Skip List (*TDASL*) method improves the *Lineage Chain* with an additional indexing layer for faster state version retrieval and multi-dimensional querying. Another indexing model, the Predefined Partitioned B-Plus Tree (*PPBPT*) optimizes historical data index generation by reducing reconstruction costs. In contrast to *Lineage Chain* and its variants, we do not replace the data storage layer with proprietary technology. We introduce a broader range of query types and support for future extensions.

Conventionally, light nodes store only the metadata, while the full nodes store the entire ledger. In *vChain* [61], an accumulator-based *ADS* is proposed to support authenticated boolean queries at the light node's end. Later, *vChain+* [62], an extension of *vChain*, further fine-tunes the indexing component by introducing sliding window accumulator to the index. Both *vChain* and *vChain+* use Merkle tree-based intra-block index and skip list-based inter-block index. Nevertheless, it still faces some challenges that limit its practicality. The first one is that *vChain+* directly supports only verifiable boolean range queries, where some other queries types can be derived. The second limitation is query efficiency. The sliding window accumulator index can speed up the proving for aggregated mismatching blocks, but it cannot directly pinpoint the desired block. Thus, the query can degenerate into a linear scan process of sliding window accumulator trie. The last issue is impractical public key management. The size of the public key increases quadratically with the number of transactions in a continuous set of *N* blocks, where *N* denotes the size of the sliding window. This results in a crucial increase of the public key size due to the large number of transactions.

Similarly, custom *ADS* methods were designed for hybrid-storage *DLTs*. By analyzing the performance of the existing techniques, *Zhang et al., 2019* [63] propose a novel *ADS*, called $GEM^2 - tree$. It is not only gas efficient but also effective in supporting authenticated queries. To further reduce the *ADS* maintenance cost without loosing query performance, they also propose $GEM^{2*} - tree$, an optimized two-level index structure.

Zhang et al., 2021 [64] utilize an authenticated query process where both smart contract and the off-chain storage maintain an *ADS* named Merkle inverted index. It is an inverted index in which each keyword corresponds to Merkle B-tree (*MB-tree*) that indexes the corresponding objects IDs. Smart contract maintains only the root digest of each *MB-tree* in the Merkle inverted index. This idea comes with the observation that for the on-chain *ADS*, only the root digests are used during the authenticated keyword search, which reduces the storage cost.

EtherQL [65] is a query layer on top of Ethereum. It synchronizes the Ethereum blockchain with a MongoDB database, and makes possible to query data in MongoDB. Queries can be performed on block, account, and transaction data.

Muzammal et al., 2019 presents *ChainSQL* [66], a blockchain database application platform developed by integrating the blockchain with the database. It represents a system that has main features of the blockchain as well as quick query processing and well-designed data structure of the distributed databases. This system features a tamper-resistant, consistent and cost-effective multi-active database. It also features an effective and reliable data-level disaster recovery backup. From implementation point of view, blockchain (Ripple [67]) stores transactions, while the database stores current data. *ChainSQL* considers its domain to be financial institutions distributed geographically, with a large volume of business transactions.

Our proposed `IS` introduces a broader range of query types and support for future extensions. Table 2.1 summarizes related work done in decentralized ledger data management. Although several research studies have recently been reported in the same field (i.e., [68] [69]), our work stands out because of streamlined query performance and diverse query support at the cost of a small index overhead.

Table 2.1: Blockchain Data Management.

Papers/Criteria	**Authenticated Data Structures**	**Rich Querying Support**	**Index Overhead**	**Point Query(S)**
Ethereum [6]	No	No	Small	3.12
HLF [3]	No	State DB	Small	6.621
TDASL [60]	Skip List, B+ tree	State DB	Large	0.0475
Merkle Inverted Index [64]	Merkle B-tree	Keyword Search	Small	N/A
EtherQL [65]	No	Yes	Small	0.039
ChainSQL [66]	No	N/A	N/A	N/A
Lineage Chain [58]	Skip List, Merkle Tree	State DB	Large	0.024
vChain+ [62]	Sliding Window Accumulator	Yes	Large	7.756
GEM^2tree [63]	Gas-Efficient Merkle Merge Tree	Yes	Small	9.7
Proposed Indexing	Key/Value Pairs	Yes	Small	0.019

Chapter 3

Background

Before going into the details of our proposed `IS`, it is essential to consider key concepts important to our work. This chapter describes fundamentals of *DLT* platforms, focusing on blockchains. It covers their design features, characteristics and advantages. Furthermore, this chapter provides an overview of main consensus protocols and blockchain systems classification. Some concepts discussed serve as inspiration for our work, while others represents already established facts in the field. Also an analysis of *HLF's* internal mechanisms from an indexing perspective will be presented since it forms the basis for our proposed `IS`.

3.1 Blockchain Overview

3.1.1 Blockchain Technology

In the pioneering white paper *Bitcoin: A Peer-to-Peer Electronic Cash System* [41], Satoshi Nakamoto introduced several groundbreaking ideas. The first was the concept of *Bitcoin*, a digital currency, which can be exchanged without a need for central financial authority. The second was a concept of blockchain, a special type of *DLT*. Blockchain is a sequence of blocks linked together by cryptographic algorithms. The core feature of this technology is storing digital assets in blocks. Such blocks are connected through a digital fingerprint or hash and they are distributed across decentralized database.

Blockchain represents a secure ledger or registry for data records. It maintains a storage of transaction histories, which are validated by participants of blockchain network. The fundamental value lies in its property to foster trust among business participants since blockchain accurately reflects the state at any given moment. Thus, blockchain functions as a state machine. It records and updates states of events as they happen, while maintaining a record of previous states. These historical states are nearly immutable.

Important and recognizable feature of blockchain technology is its use of hashing. Every block contains data to be stored, where every new block added to the chain is encoded with hash. This hash is arithmetically derived from the data stored in the block. Furthermore, each new block utilizes the hash of previous block in its own hash. Thus, block tampering becomes exceedingly challenging. Modifying a single block would require a change of the entire blockchain.

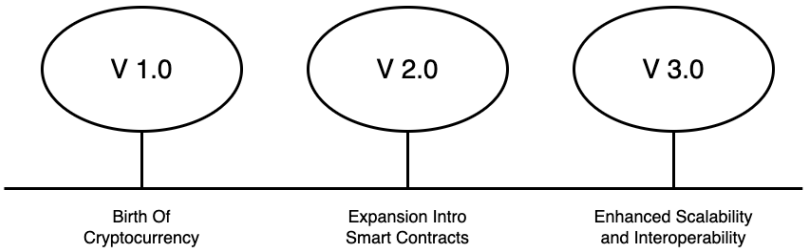

Figure 3.1: Development of Blockchain Technology.

As depicted in Figure 3.1, blockchain technology was originally developed to operate with cryptocurrency. This initial blockchain application, *Blockchain 1.0*, promoted a wide distribution of digital currencies. Further development of technology has expanded its capabilities. These advancements resulted in the creation of smart contracts, marking the evolution to *Blockchain 2.0*. Currently, *Ethereum* is one of the most prominent blockchain platforms that support and widely use smart contracts.

3.1.2 Blockchain-Based Design Features

Blockchain technology's core features are critical to its uniqueness. These features are particularly valued in environments where transaction verification, settlement and dispute resolution consume large resources. The following are the key features of blockchain-based design:

- **Transaction Confirmation:** Blockchain protocols require transaction confirmation by users before posting, ensuring transaction validity.

- **Settlement Verification:** Blockchain verifies the ownership of assets being exchanged, while counter-parties confirm transaction details. Such approach facilitates immediate settlement upon transaction completion.

- **Permanent Timestamp:** With new blocks being created and ordered, an immutability and timing of the chain is maintained.

- **Smart Contract Automation:** Blockchain ledgers support smart contracts, which can be automatically executed under specific predefined conditions.

3.1.3 Characteristics of Blockchain Technology

The characteristics of blockchain technology are extensively discussed in academic literature, with many studies highlighting similar attributes. However, some characteristics overlap and contribute to others, allowing for a more concise summary as follows:

- **Distributed:** Blockchain allows all users to access the entire system, view all transactions in the distributed ledger. It also allow to verify transaction partners' records without a central authority or trusted third party. Data validity is achieved through consensus among participants. Peers communicate directly and nodes distribute information throughout the network.

- **Standardized Rules:** Blockchain operations adhere to a consistent set of rules, which makes data falsification within a block extremely difficult. It is happening due to the properties of hashing functions and unique hash identifiers. However, blockchain can accommodate changes through the addition of new blocks to modify or delete erroneous data.

- **Privacy:** Blockchain ensures user privacy by allowing participation under generated addresses, which do not reveal real identities. However, absolute privacy cannot be guaranteed due to inherent limitations.

- **Auditability:** The blockchain's validation and transaction time-stamping enhance traceability and transparency. Computational algorithms ensure the permanence and chronological order of ledger records.

- **Security:** Blockchain's robust cryptography guarantees users a control over their addresses and associated assets. It is achieved via a combination of public and private keys, which don't allow to establish direct link to their true identities. Furthermore, the interconnected nature of blocks means that altering data within any block would necessitate changes in subsequent blocks.

3.1.4 Classification of Blockchain Systems

Since the introduction of *Bitcoin*, the landscape of blockchain systems has expanded significantly. Currently a variety of options are available for both individuals and enterprises. These blockchain systems can be primarily categorized based on two criteria: the method of access to the blockchain (permissionless and permissioned) and the accessibility of blockchain data (public and private).

- **Permissionless Blockchains:** These blockchains allow any participant to engage in the transaction verification process without needing restrictions or prior authorization. Users are free to create blocks autonomously.

- **Permissioned Blockchains:** In contrast, permissioned blockchains restrict the creation and verification of transaction blocks to a select group of preapproved users.

- **Public Blockchains:** These blockchains are open for anyone to join the network, view data, and submit transactions.

- **Private Blockchains:** Private blockchains limit the access to data and transaction submission to a specific set of users, usually within a single organization.

Peters et al., 2015 [70] observed that permissionless blockchain systems commonly provide public access, which is seen in real-world implementations. On the other hand, permissioned blockchains are designed to limit data access to the operating organization. Thus, it is noteworthy that the terms "permissionless" and "public" are often used interchangeably, as are "permissioned"

and "private". Therefore, blockchain systems can be simply classified into two main types: public/permissionless and private/permissioned.

3.1.5 Consensus Protocols

A well-known paper from 1982 [71] introduced a problem of achieving consensus among malicious entities. This paper used the byzantine generals problem to illustrate the challenge. It showed that in a scenario with three generals, if at least one general acts maliciously, the remaining two generals cannot achieve a consensus or agreement. For more general case, in order to ensure consensus among generals in the presence of up to N traitors, at least $3N + 1$ generals are required. Such number ensures that a majority of $2N + 1$ non-malicious generals can agree on the decision.

In cases of non-malicious failures, the task is to achieve an agreement or consensus on specific value or actions. Such failures are known and recognized by all participants. In contrast, in malicious scenarios, failing node might send different information to different nodes. It leads to discrepancy in observations, some nodes might not recognize a failure. Number of consensus mechanisms is used in existing blockchain systems. Although a detailed description of consensus protocols is beyond the scope of this work, we are providing a highlevel overview of *PoW* for illustration.

Proof of Work (*PoW*) is the mechanism implemented by the *Bitcoin* [41]. In *PoW*, nodes (also called as miners) validate transactions and prevent double spending. Miners aggregate validated transactions into a block. Then they compete in solving computationally intensive puzzle. The first node that solves the puzzle is able to add its block to the blockchain and receive a reward in cryptocurrency. The nature of *PoW* is in the difficulty of the puzzle. It should be challenging enough to discourage malicious activity due to high computational costs. In the same time, validating puzzle's solution should be straightforward so it can be easily done across the network. The *Bitcoin's* puzzle solution relies on brute force, making it unpredictable which node will mine the next block. Thus, several factors ensure integrity of transactions: unpredictability of puzzle's

solution and regular adjustments to the puzzle's difficulty. Outside of *Bitcoin*, *PoW* has also proven its efficiency in *Ethereum*.

3.2 Hyperledger Fabric

This section describes fundamental concepts of *HLF*, which will be used in the following sections.

3.2.1 Key Concepts Explained

We introduce Table 3.1, which represents a few rows and columns of a more complex and bigger MySQL customer database of a bicycle store. This database has several records attributed to the same customer, indicating that the same customer made multiple purchases at different times. In the context of a traditional database (like MySQL), every record in that table is a transaction. We use a simplified description of the transaction flow to explain some of the most essential terms that we will use throughout the thesis. We provide a detailed transaction flow explanation in §3.2.2.

Table 3.1: Selected Customer Data.

Transaction ID	Customer Account Key	Amount Spent (USD)	Timestamp
10874101	1999011	100	2024-01-01 10:00
10874102	1999011	200	2024-01-02 11:00
10874103	1999012	300	2024-01-01 11:30
10874104	1999011	250	2024-01-03 12:00
10874105	1999013	150	2024-01-02 12:30
10874106	1999012	350	2024-01-02 13:00

We also have a running instance of *HLF* that consists of multiple peers or nodes. A *peer* is an entity that maintains a ledger and runs *smart contracts* (code that manages access and modifications to *HLF* data). In *HLF*, smart contracts are packaged as *chaincode*.

To show how the table data can be inserted into *HLF*, we invoke a *chaincode*. This process involves taking each row of the table and inserting it into *HLF*. During this insertion process, we need to adhere to the blockchain data format. Everything we insert should have a "<*key:value*>" format. Thus, we take the *Customer Account Key* as the key and combine the rest of the data into the value.

In the context of *HLF*, *transactions* are created when a *chaincode* is invoked to read or write data to the ledger. Once enough transactions are processed, peers will group transactions into blocks. A *block* contains an ordered set of transactions. Block is cryptographically linked to the preceding block and linked to subsequent block. Since our initial Table 3.1 has several records for the *Customer Account Key 1999011*, it will result in several transactions that modify the same key in *HLF*. In this particular case, we have three updates of the key *1999011* in *HLF*. This means that this key has three historical *versions*.

To summarize, the following is a set of terms that we will use in the subsequent sections. More terms and details will be provided later, but this section describes the most fundamental concepts.

3.2.2 Transaction Flow

HLF chronologically records all transactions (both successful and failed) and organize them into blocks. These blocks are replicated across all nodes or *peers* registered in the network. The *ordering service* ensures that all *peers* receive and proceed transactions in the same order. In order to further enhance fault tolerance, an *ordering service* can be distributed with its own consensus protocol, such a Raft [72], applied. *HLF* supports channels for private communication among its different components. A subset of peers called *endorsers* has a task to simulate transaction execution and generate *read set* (the set of data items read during the transaction execution) and *write set* (the set of data items that the transaction intends to modify or create). The *endorsement policy* specifies required number of endorsements for a transaction to be considered valid.

Let's consider the transaction flow that happens during a standard asset exchange. There are two clients, A and B, who are buying and selling bicycles. Each client has a *peer* through which

they send their transactions and interact with the ledger. Figure 3.2 provides an overview of the transaction flow. We assume that the *HLF* network is successfully set up and operational.

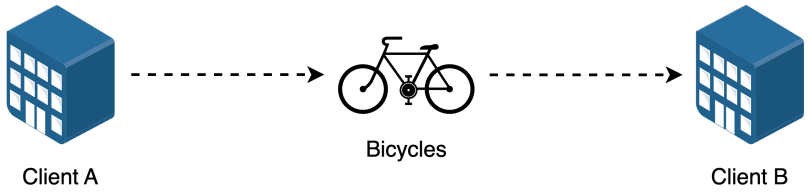

Figure 3.2: Transaction Flow Overview.

1. Client A Initiates a Transaction.

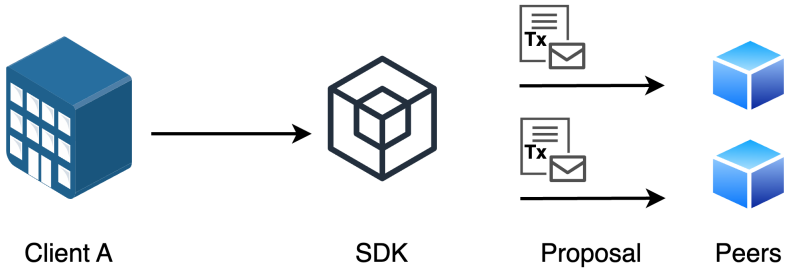

Figure 3.3: Client A Initiates a Transaction.

Client A sends a request to purchase bicycles (refer to Figure 3.3). This request targets both *peerA* and *peerB* since *endorsement policy* requires both peers to endorse transaction. Next, the *transaction proposal* is created using *Software Development Kit (SDK)* using one of the supported programming languages. The *SDK* packages the *transaction proposal* into the proper format and sign it with user specific cryptographic credentials. Then it is submitted to a target peer, which first forwards the *transaction proposal* to other peers for execution, as required by endorsement policy.

2. Endorsing Peers Verify signature and Execute the Transaction.

The *endorsing peers* verify:

1. That the *transaction proposal* is well-formed.

2. It has not been submitted already in the past.

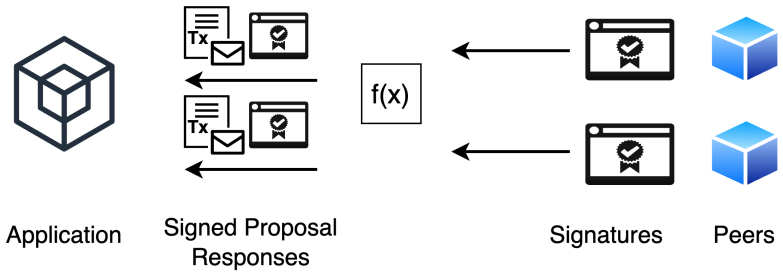

Figure 3.4: Endorsing Peers Verify and Execute Transaction.

3. The signature is valid.

4. That the submitter (Client A, in the example) is properly authorized to perform the proposed operation on that channel.

The endorsing peers take the *transaction proposal* inputs as arguments to the invoked *chaincode*'s function. The *chaincode* is then executed against the current state database. This database reflects the most recent state of each asset or data item recorded in the blockchain. The *chaincode* execution against this database produces transaction results that include response value, *read set* and *write set*. Then these values, together with the endorsing peer's signature, are sent back as a "proposal response" to the target peer (refer to Figure 3.4).

3. Proposal Responses are Inspected.

The target peer verifies that proposal responses are the same before proceeding with the transaction submission. Even if a transaction is submitted without this check, the *endorsement policy* will still be checked and enforced when each peer validates transactions prior to committing them.

4. Target Peer Assembles Endorsements Into a Transaction.

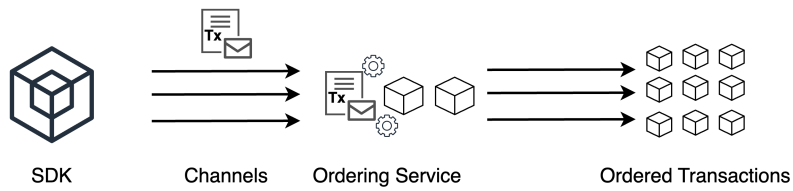

Figure 3.5: Peer Assembles Endorsements.

The target peer broadcasts both transaction proposal and response using "transaction message" to the ordering service. This message includes channel id, read-write sets and signature from every endorsing peer. The ordering service does not need to inspect the entire content of transaction to perform its operation (see Figure 3.5).

5. Transaction is Validated and Committed.

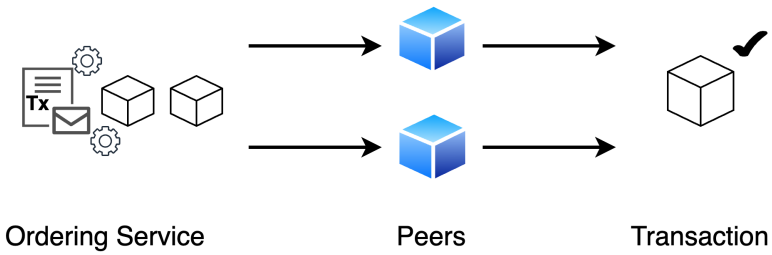

Figure 3.6: Transaction is Validated and Committed.

Transaction blocks are delivered to all peers on the channel (see Figure 3.6). Transactions in the block are tagged as being valid or invalid. Furthermore, all transactions are validated to ensure that the *endorsement policy* is fulfilled and there is no changes to the ledger state for *read set*.

6. Ledger is Updated.

Every peer appends the block to the channel's chain. *Write sets* are committed to the current state database for every valid transaction. Then each peer notifies the client that the transaction was immutably appended to the chain.

3.2.3 Data Management

HLF has two integral components: the *world state* and the *transaction log*. Both of them record facts about a set of business objects. The *world state* is a database that stores current values of ledger states. Such ledger states follow "<*key:value*>" pairs format. *World state* is mutable, so the states can be created, updated and deleted if necessary. *Transaction log* (or blockchain) stores all alterations that lead to the present *world state*.

HLF offers two options for state database: *LevelDB* and *CouchDB*. *LevelDB* is the default option, it stores *world state* in key-value format and supports limited variety of queries. *CouchDB* is an alternative solution that supports rich queries against the stored data. Furthermore, *CouchDB* supports indexes within the *chaincode* that further improve querying capabilities and efficiency.

Our research focuses on retrieving historical data from the *transaction log*, which is implemented in peer node's file system. Such architecture permits only append operations ensuring data integrity. Historical data is stored in *block files* in the file system. Each *block file* contains multiple transactions, serialized and securely stored in encoded byte format. Furthermore, specific *index files* are also stored in the file system. They allow to obtain a location pointer, which is required to retrieve specific transaction from *block files* by calculating byte offset. Therefore, such retrieval process can become notably slow. Thus, achieving fast and timely access to historical data presents a significant challenge this thesis aims to address.

For example, consider a case when auditor wishes to verify the history of ownership changes for particular asset, identified by the key *Asset138*. In order to retrieve historical data from *transaction log*, the auditor will utilize the only available API in *HLF*. This API, **GetHistoryForKey**, allows users to access a list of all transactions that have modified the state of a specified key.

This process involves several steps:

1. The auditor invokes the **GetHistoryForKey** function with *Asset138* as the argument.

2. The function queries the *transaction log* to fetch all blocks containing transactions related to *Asset138*.

3. Each transaction related to *Asset138* is examined to extract the historical values and the timestamps of the changes.

4. The results are then returned to the auditor, providing a chronological record of all values that *Asset138* has had, as well as timestamps and transaction IDs.

3.2.4 Transaction History Index

Transaction log records transaction history of decentralized ledger in *HLF*. However, since it is implemented in peer's file system, searching for specific transaction is complicated. Thus, there is a need for a way to access that data. This functionality is covered by Transaction History Index (*THI*) stored in *LevelDB* instance. When a new transaction is passed to the *HLF*, a corresponding "<*key:value*>" *THI* entry is also created. This index is complex. Each asset, identified by unique key "k" is updated with transaction data "tx" recorded in a specific block "b". This process is illustrated with the help of Example 1. It demonstrates how a sequence of transactions that update keys results in unique entries in the *THI*. Table 3.2 displays the resulting state of the *THI*.

Example 1. *Visualize the state of a history database index following a sequence of nine transactions, each impacting three keys: k1, k10, and k12. These transactions are represented as follows: PUT(k1, b8, tx1), PUT(k1, b8, tx2), PUT(k10, b8, tx3), PUT(k10, b8, tx4) for keys k1, k10 in* ***block 8****; PUT(k12, b9, tx1) for key k12 in* ***block 9****; PUT(k1, b10, tx3) for key k1 in* ***block 10****; PUT(k10, b11, tx1), PUT(k10, b11, tx2), PUT(k10, b11, tx3) for key k10 in* ***block 11****.*

Table 3.2: Transaction History Index.

Order	Key	Value
#1	ns-len(k1)-k1-b8-tx1	null
#2	ns-len(k1)-k1-b8-tx2	null
#3	ns-len(k1)-k1-b10-tx3	null
#4	ns-len(k10)-k10-b8-tx3	null
#5	ns-len(k10)-k10-b8-tx4	null
#6	ns-len(k10)-k10-b11-tx1	null
#7	ns-len(k10)-k10-b11-tx2	null
#8	ns-len(k10)-k10-b11-tx3	null
#9	ns-len(k12)-k12-b9-tx1	null

The key in the *THI* is a composite value that consists of namespace (*ns*), *key*, block number (*blockNum*) and transaction number (*tranNum*). The *ns* allows different users to operate seamlessly

in their own system fragments. Table 3.2 shows that the *THI* is sorted on key despite chronological order. It happens due to *LevelDB's* lexicographical ordering. The value component of *THI* is designed to be *null*. The key component is encoded into a byte array with order-preserving encoding.

The Algorithm 1 describes the process of creating this index. During the block commit phase, every transaction inside block is examined. For every valid endorser transaction, the algorithm iterates through the read-write set. It extracts the key for every write operation and constructs a *THI* key combining key, block number, and transaction number *(line 8)*. Next, this key is added to update batch with an empty value, indicating transaction historical record *(line 9)*. Once all updates have been added to the batch, it is written to the *LevelDB (line 13)*. The main advantage of such approach is its simplicity. However, it creates separate *THI* records when multiple transactions are in the same block, resulting in increased database size. Code for *THI* construction (*Commit* function) is included in Appendix A.1.

Algorithm 1 Hyperledger Fabric THI Construction.

Input: Block to be Committed
Output: Data and Index Commits

```
1:  procedure COMMIT(block)
2:      bNum ← block.Header.Number
3:      txNum ← 0
4:      batch ← NewUpdateBatch()
5:      for bytes in block.Data do
6:          RWSet ← GetActionFromBytes(bytes)
7:          for each kvWrite in RWSet.kvWrites do
8:              dataKey ← (kvWrite.Key, bNum, txNum)
9:              batch.Put(dataKey, emptyValue)
10:         end for
11:         txNum ← txNum + 1
12:     end for
13:     db.WriteBatch(batch)
14: end procedure
```

3.2.5 Querying Transaction History

HLF provides the **GetHistoryForKey (GHFK)** API to retrieve all historical versions of a given key (see Algorithm 2). For simplicity, from this point onward, we will work within a single

namespace, omitting most of its explicit mentions. *GHFK* commences with a construction of *rangeScan* (*rs*) tailored to the specified key *(line 2)*. This *rangeScan* defines the bounds within which the history of the key will be explored. During its creation, *rangeScan* object setups *start* and *end*. Their purpose is to outline entries relevant to specified queried key. Following this, a *LevelDB* iterator is obtained *(line 3)*. The iterator is positioned at the end, enabling reverse reverse traversal from the most recent to the oldest entry *(line 4)*. If there is no error during obtaining *LevelDB* iterator, a wrapper *historyScanner* object is created over an iterator *(line 5)*. It plays a key role in iterating the ledger's historical data. It implements a *Next* method that iterates from the newest to the oldest entries *(line 6)*. Then *bNum* and *txNum* are decoded from each historical entry associated with a key *(line 8-9)* during iterations. The algorithm fetches an actual transaction data associated with *THI* entry from file system block storage using the *bNum* and *txNum (line 10)*. Then, the *historyScanner* is repositioned to the next *THI* record. This traversing process repeats until all historical versions are found or there are no more records to search. A code for **GetHistoryForKey** is included in Appendix A.2.

Algorithm 2 GetHistoryForKey or GHFK

Input: Key For Which History is Being Retrieved
Output: Query Result Iterator

1: **procedure** GHFK(key)
2: $rs \leftarrow$ constructRangeScan(key)
3: $itr \leftarrow levelDB.Itr(rs.start, rs.end)$
4: $itr.Last()$
5: $historyScanner \leftarrow (rs, key, dbItr)$
6: $found \leftarrow historyScanner.Next()$
7: **repeat**
8: $hk \leftarrow historyScanner.itr.Key()$
9: $bNum, txNum \leftarrow decode(hk)$
10: **return** $fetchTransactionData(bNum, txNum)$
11: $found \leftarrow historyScanner.Next()$
12: **until not** $found$
13: **end procedure**

Chapter 4

Foundations of Indexing Solution

This section describes *IS*. First, it presents an overview of proposed solution (see §4.1). Next, we discuss the architecture (see §4.2). Furthermore, we describe new indexing methods (*VBI* and *BBI*), including maintaining and querying (see §4.3). We also introduce new query APIs that utilize proposed indexing techniques for enhanced performance (see §4.4).

4.1 Overview of IS

Initially, we examine various internal operational mechanisms in *HLF* with focus on data processing, index creation and query handling. Next, building on these insights, we integrate *IS* into *HLF* infrastructure and introduce new query APIs. Existing querying mechanisms in *HLF* are heavily relied on the *THI*. Therefore, in response, we entirely overhaul it to maintain both *VBI* and *BBI*.

4.2 IS Architecture

Reflecting on the querying time for one million Ethereum data records (see §1.1), it becomes clear that slow data retrieval is a significant challenge, which inhibits spread of blockchain technology. Furthermore, as shown with *HLF* example in §3.2.5, the limited variety of supported queries strengthens this issue. Thus, there is a need to develop indexing mechanism that can handle complex queries, for example, those involving ranges of blocks or keys.

In order to address these challenges, we propose the *IS* solution. The following Figure 4.1 illustrates the architecture, highlighting its role as intermediary between the user and decentralized system such as *HLF*. Such design enables efficient data-driven business operations without compromising the integrity and functionality of the underlying blockchain system. The *IS* reuses various parameters previously used in *THI*, such as keys, block numbers, transaction numbers and version counters.

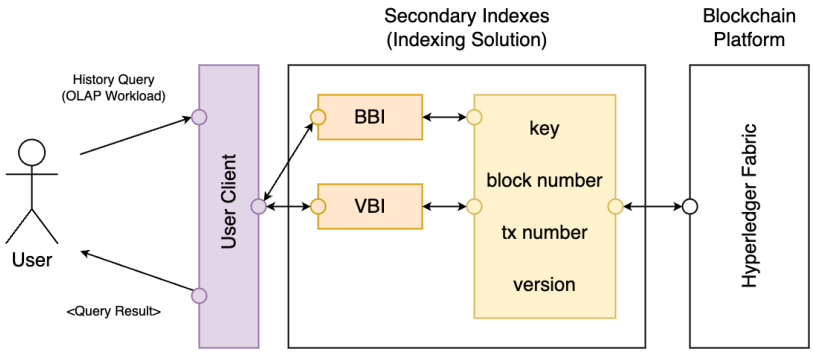

Figure 4.1: Architecture of IS.

4.3 Designing Indexing Techniques

We propose two indexing techniques: the Version-Based Index (`VBI`) and the Block-Based Index (`BBI`). Both of them maintain two indexing entities: Global Index (*GI*) and Local Index (*LI*). *LI* and *GI* records follow "<*key:value*>" structure, being created or modified with every new transaction in the system. The *LI* records are complex. They are identified by key "k", updated with transaction data "tx" in the block "b". Furthermore, *LI* records store additional information specific to `VBI` or `BBI`. In order to illustrate `VBI` and `BBI`, we use the same Example 1 previously used for *THI* illustration (see §3.2.4).

4.3.1 Version-Based Index

The purpose of `VBI` is to enhance efficiency of queries that necessitate access to one or more historical versions of a key. Thus, the *GI* is optimized to locate an information about the most recent version of an asset. The *LI* maintains direct access to specific historical versions, eliminating a need for linear search.

Both `VBI` and `BBI` support new types of queries (see §4.4): (1) querying a specific historical version of a key; (2) querying a consecutive range of historical versions of a key; (3) querying historical versions of keys that have undergone a minimum number of updates within a specified block range; (4) querying all historical versions for a range of contiguous keys. Among these new queries, `VBI` significantly improves the efficiency of version-centric queries (query (1) and

(2)). Such improvement is advantageous when a rapid access to specific historical data versions is required.

Table 4.1: Version-Based Index.

LI/GI	Key	Value
LI	ns-len(k1)-k1-minVer1	b8-{tx1, tx2}
	ns-len(k1)-k1-minVer3	b10-{tx3}
	ns-len(k10)-k10-minVer1	b8-{tx3, tx4}
	ns-len(k10)-k10-minVer3	b11-{tx1, tx2, tx3}
	ns-len(k12)-k12-minVer1	b9-{tx1}
GI	ns-k1	ver3
	ns-k10	ver5
	ns-k12	ver1

We describe the process of maintaining the `VBI`, as illustrated in Example 1, by demonstrating the creation of distinct *LI* and *GI* entries through a sequence of transactions involving three keys. Table 4.1 demonstrates the state of the `VBI` after operations described in Example 1. Each transaction writes an update to the *GI* to reflect the most recent version number. For example, after three updates, key *k1* has version 3 in the *GI*. Similarly, key *k10* has version 5 after five updates. The *LI* value combines transactions related to the same key within the same block into a singular record. Thus, *LI* value includes block number and the list of relevant transactions. *LI* key consists of updated key (like customer id) and the minimum version number included in that *LI* entry. For example, the second *LI* entry for key *k10* has minimum version 3. It means that this *LI* entry has at least one transaction, which is the third overall for *k10*. It is worth noting that *LI* entries are sorted by key.

Algorithm 3 illustrates a `VBI` traversal. It starts when the user specifies a key and a desired version of that key as input. First, it accesses a *GI* record corresponding to the specified key *(line 2)*. Then it retrieves the latest version number stored in the *GI* for that key. If there is no record for a given key, then there is no recorded transactions for that key. If the version number retrieved

from the *GI* is less than the desired version specified by user, then requested version does not exist yet. Upon validating that the desired version is within the range of existing versions, the algorithm prepares to scan the *LI*. It constructs start and end limits for the scan *(line 8-9)*. The end limit is specifically crafted by adding one to the desired version to ensure that the version required by user is in scan range. Using start and end limits, the algorithm initiates a scan over the *LI* by creating a database iterator *(line 9)*. It positions the iterator at the last record within the specified range *(line 10)*. This step is crucial since it makes sure that the desired historical version is in the last record within the iterator. It happens due to the *LI* design features. After locating the appropriate *LI* record, the algorithm extracts the minimum version number *(line 13)*, decodes the block number and the list of transactions from the record's value *(line 14)*. Then it calculates the index of the desired transaction within the list by subtracting the minimum version number in the record from the desired version number. With the transaction index found, the algorithm fetches the specific transaction from the list. It then retrieves the actual transaction data from the block storage, utilizing the block number and the transaction identifier *(line 17)*. This final step returns the user historical data requested.

The following *Example 2* demonstrates a query execution based on the Version-Based Index state from *Table 4.1*.

Example 2. ***Executing a Query Using Version-Based Index***

User invokes the GetSpecificVersion procedure for version 4 of key 10. The number of versions for that key is read from the GI (lines 2-7). There are 5 versions for key 10. Next, a database iterator is created spanning from the first entry for key 10 up until the entry containing the desired version (lines 8-9).

$$k10 - minVer1 : b8 - tx3, tx4$$

$$k10 - minVer3 : b11 - tx1, tx2, tx3$$

The record containing the desired version is read from the history database (lines 10-12).

Algorithm 3 VBI Traversal for Accessing a Specific Key Version.

Input: Key, Desired Version
Output: Query Result Iterator

```
1: procedure GETSPECIFICVERSION(key, desiredVer)
2:     GIk ← key
3:     bytes ← Get(GIk)
4:     if bytes = nil then
5:         return error("Key not found in Global Index.")
6:     end if
7:     latest ← Decode(bytes)
8:     rs ← constructRangeScan(key)
9:     itr ← GetIterator(rs.start, rs.end)
10:    itr.Last()
11:    k ← itr.Key()
12:    v ← itr.Value()
13:    min ← extract(k)
14:    bNum, txList ← extract(v)
15:    i ← desiredVer − min
16:    desiredTx ← txList[i]
17:    return fetchTxData(bNum, desiredTx)
18: end procedure
```

$$k10 - minVer3 : b11 - tx1, tx2, tx3$$

Next, the first version whose index data is contained in that record is determined (line 13) and the block number as well as list of transactions are decoded from the record (line 14).

$$minVersionInRecord == 3$$

$$blockNum == 11$$

$$txList == [1, 2, 3]$$

An index, determining which transaction will be retrieved from the block storage is calculated (line 15).

$$txIndex == 1$$

The transaction number is read from the list using the index (line 16).

$$desiredTx == 2$$

The block and transaction number (block 11, transaction 2) are then used to retrieve version 4 for key 10 from the file system (line 17).

4.3.2 Block-Based Index

The `VBI` is ordered based on keys and it helps for queries involving key range (k_i, k_j), where $i \geq j$. However, it is inefficient for queries involving block ranges since we cannot assume what keys were updated in the specified block range, i.e., (b_i, b_j), where $i \geq j$. Thus, we have designed the `BBI` mechanism to address this issue.

Table 4.2: Block-Based Index.

LI/GI	Key	Value
LI	ns-b8-len(k1)-k1	b8-numVer2-{tx1, tx2}
	ns-b8-len(k10)-k10	b8-numVer2-{tx3, tx4}
	ns-b9-len(k12)-k12	b9-numVer1-{tx1}
	ns-b10-len(k1)-k1	b8-numVer3-{tx3}
	ns-b11-len(k10)-k10	b8-numVer5-{tx1, tx2, tx3}
GI	ns-k1	b10
	ns-k10	b11
	ns-k12	b9

The *GI* directs user to the most recent block that has updates for the target key. The *LI* organizes records in block first order, which enhances data retrieval efficiency for block range queries. We consider the same Example 1, previously on the `VBI`, to show both *LI* and *GI* states after a series of transactions. Each transaction updates the *GI* so its every entry points to the most current block number associated with it. For example, key *k1*, updated three times, points to block 10 and key *k10*, updated five times, points to the block 11. The *LI* aggregates transactions per key within the block, similar to the `VBI`. It results in a single record per block per key in the *LI*. The *LI* key

consists of the block where the key (like customer id) was updated as well the key itself. The *LI* value includes previous block where the given key was updated, total number of historical versions for given key and a list of transactions. For example, if key *k1* is updated in block 10, the *LI* value part points to block 8 as the previous update block, specifies one transaction (tx3) updating *k1* in block 10 and indicates a total of three historical versions for *k1*. Table 4.2 demonstrates the `BBI` state after described operations.

Algorithm 4 BBI Traversal for Accessing a Specific Key Version.

Input: Key, Desired Version
Output: Query Result Iterator

```
1: procedure GETSPECIFICVERSION(key, desiredVer)
2:     GIk ← key
3:     bytes ← Get(GIk)
4:     latest ← Decode(bytes)
5:     dataKey ← constructDataKey(key, latestBlockNum)
6:     itr ← GetIterator(nil, nil)
7:     found ← itr.Seek(dataKey)
8:     repeat
9:         k ← itr.Key()
10:        v ← itr.Value()
11:        prev, maxVer, txList ← decode(v)
12:        first ← calculateFirstVersion(maxVer, txList)
13:        if first ≤ desiredVer then
14:            i ← desiredVer − first
15:            desiredTx ← txList[i]
16:            return fetchTransactionData(latest, desiredTx)
17:        end if
18:        latest ← prev
19:        dataKey ← constructDataKey(k, latest)
20:        found ← itr.Seek(dataKey)
21:    until not found
22:    return error("Desired version not found.")
23: end procedure
```

The Algorithm 4 illustrates a `VBI` traversal. It starts when user specifies a key and a desired version of that key as input. Unique identifier for the specified key is constructed to access its record in the *GI*. That step determines latest block containing updates for the key, which serves as the starting point for the search within the *LI*. Using the block number obtained from the the *GI (line 4)*, an algorithm constructs a *dataKey*. It is designed to seek corresponding record in the *LI (line 5)*. Once the relevant block record is located in the *LI*, the algorithm decodes it to extract previous block number, total number of transactions and list of transactions *(line 11)*. Using that

information, the algorithm calculates the first version number within the block. It then assesses if the desired version is in that block or not *(line 13)*. If requested version is within the block, the algorithm proceeds to retrieve the actual transaction data. It fetches transaction data from the block storage on the disk using key, block number and transaction number *(line 16)*. This step culminates a successful retrieval of requested historical data. If the desired version is not within the block, the algorithm uses previous block number and repositions the iterator to the next *LI* record *(line 19)*. Such process repeats, navigating backward through the blocks and transactions, until the specific historical version is found or there is no more records to search.

The following Example 3 demonstrates a query execution based on *BBI* state from Table 4.2.

Example 3. ***Executing a Query Using Block-Based Index***

User invokes the GetSpecificVersion procedure for version 4 of key 10. The last block where the key 10 appeared is read from the GI (lines 2-4).

$$latestBlockNum == 11$$

A database iterator is created, the latestBlockNum is used to construct a history database key, and the iterator is moved to that record (lines 5-7). Then, records are read in reverse chronological order, repeating until the index record containing the desired version is found.

Previous block, total number of versions and the list of transactions are decoded from each record (lines 9-11).

$$prevBlock == 8$$

$$maxVersion == 5$$

$$txList == [1, 2, 3]$$

The first version stored in the current block is calculated by subtracting the length of the transaction list from the max version and adding 1 (line 12).

$$firstVersionInBlock == 3$$

Next, it checks if this record contains the desired version. It is done by checking if the first version in the block is less than or equal to the desired version (line 13). For our example this condition is true during the first iteration. The index determining which transaction will be retrieved from the block storage is calculated (line 14). It is done by subtracting the first version number from the desired version number.

$$txIndex == 1$$

The transaction number is read from the list using the index (line 15).

$$desiredTx == 2$$

The block and transaction number (block 11, transaction 2) are then used to retrieve version 4 for key 10 from the file system's block storage (line 16).

4.4 Designing Rich Query APIs

In this work, we also introduce several specialized query APIs: **GetHistoryForVersionRange (GHFVR)**, **GetHistoryForBlockRange (GHFBR)** and **GetHistoryForKeyRange (GHFKR)**. These APIs enhance querying capabilities beyond *HLF's* **GetHistoryForKey (GHFK)** API. **GHFVR** focuses on retrieving historical versions for a given key, optimizing version-based queries. **GHFBR** improves the efficiency of traversing block ranges, suitable for block range queries. **GHFKR** caters to key range queries by fetching data for a range of contiguous keys. Figure 4.2 illustrates the integration of these APIs within our system architecture. Our proposed `IS` not only supports these new APIs but also optimizes queries based on the schema used. Specifically, **GHFVR** is more efficient within the `VBI` schema, while **GHFBR** performs better using the `BBI` schema. Such task optimized efficiency ensures that our system can handle diverse and com-

plex query requirements effectively. Specific details on implementation strategies for **GHFVR**, **GHFBR** and **GHFKR** are in sections §5.6, §5.7 and §5.8, respectively.

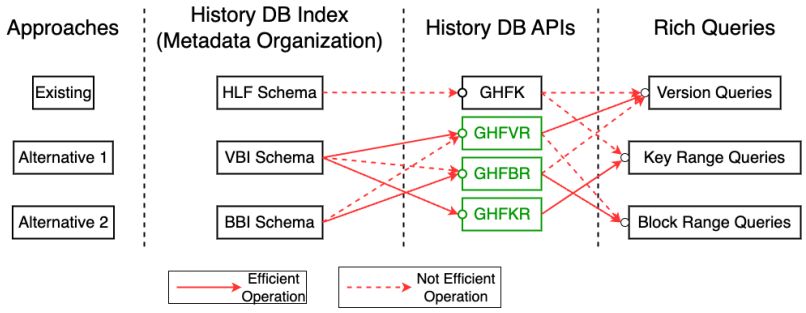

Figure 4.2: Proposed Querying APIs.

Chapter 5

Indexing Solution Implementation

In this section we describe the implementation of the proposed `IS`. It builds on foundational overview of *HLF* (see §3.2) and `IS` design specifics (see §4.3, §4.4). Our task here is to elaborate on `VBI` and `BBI` implementation details, which lies in the replacement of the LevelDB *THI* database with *GI* and *LI*. We also describe implementation details behind new query APIs: **GHFVR**, **GHFBR** and **GHFKR**. However, it's important to note that our enhancements operate in peace with the existing infrastructure. The original *HLF* remain untouched and fully functional.

5.1 Global Index

One of the cornerstones of our work is the introduction of *GI*. By our design `IS` should require minimum changes to the *HLF*, so we position *GI* within the *LevelDB* instance, originally used as a *THI* database. Given the *LevelDB* nature and lexicographical sorting, each *GI* record follows "<*key:value*>" format. For both `VBI` and `BBI`, the key is the business asset key (such as customer id). In terms of the value part of the record, for `VBI`, the value represents the latest version of an asset. As for `BBI`, the value contains the latest block where the corresponding key was updated.

5.2 Local Index

Another cornerstone is the introduction of *LI*. Following the same architectural reasoning as for the *GI*, we positioned the *LI* within the same *LevelDB* instance as *GI*. Thus, we completely replace the original *THI*. Adhering to the *LevelDB* key-value record format, the *LI* introduces a new "<*key:value*>" structure for `VBI` and `BBI`.

`VBI` key part of *LI* includes namespace, length of the key, key, and the minimal version number updated in that index record (*minVer*). Correspondingly, the value part includes the block number where the key is currently updated and a list of transactions that updated a key in that block.

`BBI` key part of *LI* includes namespace, block number, length of the key, and key. The value part includes the block number where the key was updated previously, the total number of versions this key has in the whole ledger and the list of transactions that updated a key in that block. When the first *LI* record is created for a key, its previous block number where the key was updated points to itself. Our experiments show that writing this self-pointer introduces a negligible space overhead of only 0.4 MB for 1 million Ethereum dataset and 4 MB for 10 million Ethereum dataset. On the other hand, eliminating this self-referential pointer introduces additional complications, affecting performance.

5.3 General Steps in Index Construction Workflow

First, we outline general index construction flow that is common for both `VBI` and `BBI`. During the initial step of Algorithm 5, it extracts the block number from the provided block and initializing the transaction number to 0. It also prepares a database update batch and creates empty map to hold data keys. That map is used to track transactions associated with specific keys. The algorithm iterates over each transaction within the block *(line 6)*. It uses the validation flags from the block's metadata to determine the validity of each transaction *(line 7)*. Invalid transactions are skipped. For each valid transaction, algorithm extracts the transaction envelope, payload, and channel header. Then transactions are processed depending on the indexing technique (see §5.4 and §5.5) *(line 12)*. Later, after processing all transactions, the algorithm adds a savepoint to the database update batch *(line 17)*. It serves as a recovery point and marks the last successfully processed block and transaction. The update batch is then committed to the database, completing the commit operation *(line 18)*.

5.4 Version Based Index Construction

The `VBI` specific construction focuses on processing read-write sets to update transaction log, *LI* and *GI*. The Algorithm 6 examines every read-write set within the transaction *(line 2)*. For every write operation found, it constructs *GI* key and retrieves its current value (if any) *(lines 3-6)*.

Algorithm 5 General Index Construction Procedure.

Input: Block to be Committed
Output: Data and Index Commits

```
1: procedure COMMIT(block)
2:     bNum ← block.Header.Number
3:     txNum ← 0
4:     batch ← NewUpdateBatch()
5:     dataKeys ← NewMap()
6:     for all tx in block.Data.Data do
7:         if tx is valid then
8:             txEnvelope ← ExtractEnvelope(tx)
9:             txPayload ← ExtractPayload(txEnvelope)
10:            channelHeader ← ExtractChannelHeader(txPayload)
11:            if channelHeader.Type is endorser transaction then
12:                PROCESSRWSET(bNum, txNum, batch, dataKeys)
13:            end if
14:        end if
15:        txNum ← txNum + 1
16:    end for
17:    batch.Put(savePointKey, height.ToBytes())
18:    db.WriteBatch(batch)
19: end procedure
```

For each key in the write set, the version is incremented to reflect the new write *(line 11)*. Current transaction is appended to the list of transactions associated with the key. The *GI* then is updated for the key with new maximum version *(line 15)*. A *LI* entry is created and added to the database update batch *(lines 17-19)*.

Table 4.1 (see §4.3.1) presents the state of the `VBI` after processing a series of nine transactions. A code-snippet for Version-Based Index Construction is included in Appendix B.1.

5.5 Block-Based Index Construction

The core of `BBI` construction (Algorithm 7) involves iterating through transactions within the read-write set *(line 2)*. The algorithm initializes or updates a *GI* for every key (such as customer id) involved in write operation. The *GI* maintains a record of the latest block number where the key was updated. The *LI* is updated to reflect every update to a key within the block. A *dataKeys* map of *GI* values is maintained to reflect the new write operation for each key. First, the map is checked to find out if the key was previously updated in the current block. If yes, algorithm reads the records from the map *(lines 4-6)*. Otherwise, algorithm makes an attempt to retrieve the

Algorithm 6 Version Based Index Construction.

Input: Block Number, Transaction Number, Update Batch, DataKeys Map
Output: Void

```
1: procedure PROCESSRWSET-VBI(bNum, txNum, batch, dataKeys)
2:     for each kvWrite in RWSet.KvRwSet.Writes do
3:         GIk ← kvWrite.Key
4:         bytes ← database.get(GIk)
5:         if bytes is not null then
6:             v, txList ← decode(bytes)
7:         else
8:             v ← 0
9:             txList ← emptylist
10:        end if
11:        v ← v + 1
12:        txList.append(txNum)
13:        dataKeys[kvWrite.Key] ← txList
14:        bytes ← encode(v)
15:        database.put(GIk, bytes)
16:        minVer ← v − length(txList) + 1
17:        dataKey ← buildKey(kvWrite.Key, minVer)
18:        val ← buildVal(bNum, txList)
19:        batch.put(dataKey, val)
20:    end for
21: end procedure
```

previous block number where the key was updated *(lines 8-10)*. If it return *null*, the key appears in the ledger for the first time. New record is initialized for the key *(lines 11-14)*.

Next, transaction number is appended to the list of transactions in the current block *(line 17)*. The version count is incremented to reflect the new write operation *(line 18)*. Then the information required for *LI* value part (previous block, version counter, transaction list) is encoded *(line 19)*. Both *dataKey* map and *GI* are updated *(lines 20-23)*. Complete *LI* entry is created and added to the database update batch *(line 24)*.

Table 4.2 (see §4.3.2) presents the state of the `BBI` after processing a series of nine transactions. A code-snippet for Block-Based Index Construction is included in Appendix C.1.

5.6 Implementing GetHistoryForVersionRange

We introduce **GHFVR** that makes possible to efficiently query one or multiple consecutive historical versions. Due to the difference in indexing techniques, we implemented that API for both `VBI` and `BBI`. Both implementations are based on index traversing algorithms described in §4.3.1 and §4.3.2.

Algorithm 7 Block-Based Index Construction.

Input: Block Number, Transaction Number, Update Batch, DataKeys Map
Output: Void

procedure PROCESSRWSET-BBI($bNum, txNum, batch, dataKeys$)
  for each kvWrite in RWSet.KvRwSet.Writes **do**
    $GIk \leftarrow kvWrite.Key$
    $val, present \leftarrow dataKeys[kvWrite.Key]$
    if present **then**
      $prev, maxVer, txList \leftarrow$ decode(val)
    else
      $bytes \leftarrow$ db.get(GIk)
      if $bytes$ is not null **then**
        $prev, maxVer, txList \leftarrow$ decode($bytes$)
      else
        $prev \leftarrow bNum$
        $maxVer \leftarrow 0$
        $txList \leftarrow$ empty list
      end if
    end if
    $txList \leftarrow append(txList, txNum)$
    $maxVer \leftarrow maxVer + 1$
    $val \leftarrow$ buildVal($prev, maxVer, txList$)
    $dataKeys[kvWrite.Key] \leftarrow val$
    $bytes \leftarrow$ buildGI($bNum, maxVer$)
    $db.put(GIk, bytes)$
    $dataKey \leftarrow$ buildKey($bNum, kvWrite.Key$)
    $batch.put(dataKey, val)$
  end for
end procedure

Algorithm 8 GetHistoryForVersionRange for VBI

Input: Key, Earliest Desired Version, Latest Desired Version
Output: Query Result Iterator (Version Scanner)

procedure GETHISTORYFORVERSIONRANGE($key, startV, endV$)
  $GIk \leftarrow key$
  $bytes \leftarrow db.get(GIk)$
  if $bytes \neq nil$ **then**
    $max \leftarrow$ `decode`($bytes$)
  end if
  $rs \leftarrow$ `constructRangeScan`(key)
  $end \leftarrow$ concat($rs.start$, `encode`($endV + 1$))
  $itr \leftarrow$ `levelDB.GetIterator`($rs.start, end$)
  $bNum, tx, i \leftarrow Initialize$
  if $itr.Last()$ **then**
    $histKey \leftarrow itr.Key()$
    $first \leftarrow$ `decodeMinVer`($histKey$)
    $bNum, tx \leftarrow$ `decodeNewIndex`($itr.Value()$)
    $last \leftarrow first + |transactions| - 1$
    if $endV > last$ **then**
      $i \leftarrow |transactions| - 1$
    else
      $i \leftarrow endV - first$
    end if
  else
    $i \leftarrow -1$
  end if
  return `versionScanner`(rs, key, itr, bNum, tx, i, startV, endV)
end procedure

The `VBI` realization, depicted in Algorithm 8, starts with retrieving latest version number from *GI (lines 2-5)*. The algorithm then constructs a *rangeScan* (*rs*) object *(line 7)*. This object includes *LevelDB* iterator that traverses *LI* entries within the specified version range *(line 9)*. The iterator is positioned at the last record within the specified range since *LI* makes sure that the last desired version is within this last record. The algorithm decodes the block number and the list of transactions from the *LI* value *(lines 11-15)*. It calculates the index of the desired transaction *(lines 16-22)*. Finally, it prepares and returns versionScanner object *(line 24)*. This object allows to iteratively access and retrieve desired historical versions within specified range. A complete code-snippet for `VBI` **GetHistoryForVersionRange** API is included in Appendix B.2.

Algorithm 9 GetHistoryForVersionRange for BBI.

Input: Key, Earliest Desired Version, Latest Desired Version
Output: Query Result Iterator (Version Scanner)

1: **procedure** GETHISTORYFORVERSIONRANGE($key, startV, endV$)
2: $itr \leftarrow$ `levelDB.GetIterator`(nil, nil)
3: $GIk \leftarrow key$
4: $bytes \leftarrow db.get(GIk)$
5: **if** $bytes = nil$ **then**
6: **return** `versionScanner`(nil)
7: **end if**
8: $bNum \leftarrow$ `decode`$(bytes)$
9: $dataKey \leftarrow$ `buildKey`$(bNum, key)$
10: $found \leftarrow dbItr.Seek(dataKey)$
11: **if not** $found$ **then**
12: **return** error("Key not found in block")
13: **end if**
14: $val \leftarrow dbItr.Value()$
15: $prev, txList \leftarrow$ `decode`(val)
16: $i \leftarrow |txList| - 1$
17: $scanner \leftarrow$ `versionScanner`(key, itr, bNum, val, i, startV, endV)
18: **while** true **do**
19: $maxVer, txList \leftarrow$ `decode`(val)
20: $first \leftarrow maxVer - |txList| + 1$
21: **if** $first \leq endV$ **then**
22: **if** $maxVer \geq endV$ **then**
23: $scanner.i \leftarrow endV - first$
24: **else**
25: $scanner.i \leftarrow |txList| - 1$
26: **end if**
27: **return** $scanner$
28: **end if**
29: $old \leftarrow scanner.current$
30: $scanner.updateBlock()$
31: **if** $scanner.current = old$ **then**
32: $scanner.i \leftarrow -1$
33: **return** $scanner$
34: **end if**
35: **end while**
36: **end procedure**

Algorithm 9 details the execution of **GHFVR** for `BBI`. In contrast to `VBI`, the `BBI` implementation initializes a database iterator to scan the entire database without a predefined range *(line 2)*. This approach addressed the necessity to traverse blocks to locate desired historical versions. The algorithm accesses the *GI* to determine the latest block number containing updates for given key *(line 3)*. Next, it constructs *dataKey* in order to locate corresponding *LI* record *(line 9)*. Upon finding that *LI* record, the algorithm decodes it to extract previous block number and transaction list *(line 15)*. A *versionScanner* (*scanner*) is initialized *(line 17)*. It iterates the blocks until the *LI* record, containing the desired version is found. For every iteration, the first historical version in the block is determined *(line 20)*. If this first version is less than or equal to the desired version, it means that the current *LI* record contains the desired version *(line 21)*. Next, the index for required transaction inside of the transaction list is calculated. If the desired end version exists, the iterator is set to the corresponding location *(lines 22-23)*. Otherwise, iterator is positioned at the end of the list and the iterator begin to return results from the end *(lines 24-25)*. Algorithm then returns a *versionScanner* object for iterative access to historical versions. If *LI* record does not contain specified end version, scanner retrieves the next *LI* record based on previous block information *(lines 29-30)*. If the previous block value is the same as the current block, it indicates that we have reached the first *LI* record for that key *(line 32)*. A complete code-snippet for `BBI` **GetHistoryForVersionRange** API is included in Appendix C.2.

5.7 Implementing GetHistoryForBlockRange

Another API that we introduce is *GHFBR*. It is suitable for cases when user is searching for historical versions of assets that have undergone at least some minimum number of updates within a certain block range. We implement this API for both `VBI` and `BBI`.

Algorithm 10, designed for the Version Based Index, starts by initializing a *blockRangeScanner* object *scanner*. It uses key, start and end block range limits and the minimum number of updates as parameters for initialization *(line 2)*. Next, the *countKeyUpdates* method of the scanner object is called. This method iterates through the blocks within the specified range and counts how many

Algorithm 10 GetHistoryForBlockRange for VBI.

Input: Key, Earliest Desired Block, Latest Desired Block, Minimum Number of Updates
Output: Query result iterator (Block Range Scanner)

```
1: procedure GETHISTORYFORBLOCKRANGE(key, startB, endB, minUpd)
2:     scanner ← blockRangeScanner(key, startB, endB, minUpd)
3:     scanner.countKeyUpdates(minUpd)
4:     hasNext ← scanner.nextKey()
5:     if not hasNext then
6:         return error("No keys found meeting the update threshold")
7:     end if
8:     return scanner
9: end procedure
```

times each key is updated. Keys that meet or exceed the minimum number of updates are marked for further processing. Once the algorithm identifies keys meeting the update criteria, it attempts to position an iterator, which contains all the relevant keys and their corresponding versions at the first key for further processing *(line 4)*. If there is no key satisfying the update threshold, algorithm returns false *(line 5)*. Otherwise, if such keys are found, algorithm returns an initialized *scanner* object. This object allows user to iterate over relevant updates and retrieve desired data *(line 8)*. A complete code-snippet for `VBI` **GetHistoryForBlockRange** API is included in Appendix B.3.

Algorithm 11 GetHistoryForBlockRange for BBI

Input: Key, Earliest Desired Block, Latest Desired Block, Minimum Number of Updates
Output: Query Result Iterator (Block Range Scanner)

```
1: procedure GETHISTORYFORBLOCKRANGE(key, startB, endB, minUpd)
2:     start ← concat(encode(startB))
3:     end ← concat(encode(endB + 1))
4:     itr ← levelDB.GetIterator(start, end)
5:     scanner ← blockRangeScanner(itr, startB, endB)
6:     scanner.countKeyUpdates(minUpd)
7:     return scanner
8: end procedure
```

`BBI` specific Algorithm 11 starts with construction of start and end range limits for the database iterator *(lines 2-3)*. With start and end limits established, the algorithm initializes *LevelDB* iterator to traverse the entries within specified block range. A *blockRangeScanner* object *scanner* is then initialized. It takes *LevelDB* iterator as well as block range as parameters *(line 4)*. *Scanner* has *countKeyUpdates* method *(line 6)*. This method traverses the *LevelDB* iterator and counts the

number of updates per key within the specified block range. Keys that meet or exceed an updates threshold are marked for further processing. Finally, the *scanner* object is returned *(line 7)*. User iterates through this object and retrieves required data. A complete code-snippet for `BBI` **GetHistoryForVersionRange** API is included in Appendix C.3.

5.8 Implementing GetHistoryForKeyRange

We also introduce *GHFKR*, which makes possible to efficiently query all historical versions for a range of contiguous keys. This API is implemented for both `VBI` and `BBI`.

Algorithm 12, designed for `VBI` begins by initializing an empty map to store history scanners for each key *(line 2)*. For each key provided in the input list, the algorithm invokes the **GHFK** API adjusted to each respective indexing strategy *(line 4)*. Upon retrieval, each key's history scanner is stored in the map *(line 5)*. Then a *multipleHistoryScanner* object is created *(line 7)*. This object encapsulates the list of keys and the map of individual history scanners. Thus, it enables unified interface to iterate over the historical data of multiple keys. The algorithm then returns the *multipleHistoryScanner* object, indicating a successful initialization *(line 8)*. Next, user can use this scanner to iteratively access and return the historical versions of each key in required range. A complete code-snippet for `VBI` **GetHistoryForKeyRange** API is included in Appendix B.4.

Algorithm 12 GetHistoryForKeyRange for VBI

Input: List of Keys
Output: Query Result Iterator (Multiple History Scanner)

1: **procedure** GETHISTORYFORKEYRANGEVBI($keys$)
2: $scanners \leftarrow$ map of empty historyScanners
3: **for** each k in $keys$ **do**
4: $scanner \leftarrow$ GetHistoryForKey(k)
5: $scanners[k] \leftarrow scanner$
6: **end for**
7: $multiScanner \leftarrow$ new multipleHistoryScanner($keys, scanners$)
8: **return** $multiScanner$
9: **end procedure**

The Algorithm 13 for `BBI` starts by creating an empty map to hold *keyData* structures and an empty list to track valid keys *(lines 2-3)*. Every key in the input list is processed to determine its

validity. For each key, the algorithm initializes *LevelDB* iterator without specifying a range since it needs to traverse database sorted by blocks rather than versions *(line 5)*. Algorithm checks the *GI* to access the latest block number *(lines 6-7)*. If the key exists and is associated with a block, it is considered valid. Algorithm constructs a *datakey* to locate the corresponding record in the *LI (line 9)*. This record contains current block number, previous block number and the transaction list. Upon a successful retrieval of this information, specific structure is initialized for the key. It contains an iterator, block number, transaction list and an indicator of whether the key's history was fully traversed *(line 16)*. Once all keys are processed, a *parallelHistoryScanner* object is created *(line 20)*. This object encapsulates all the necessary information to iterate over the historical data of multiple keys. The algorithm concludes by returning this *parallelHistoryScanner* object *(line 21)*. A complete code-snippet for `BBI` **GetHistoryForKeyRange** API is included in Appendix C.4.

Algorithm 13 GetHistoryForKeyRange for BBI

Input: List of Keys
Output: Query Result Iterator (Multiple History Scanner)

1: **procedure** GETHISTORYFORKEYRANGEBBI($ns, keys$)
2: $keyMap \leftarrow$ empty map of keyData
3: $validKeys \leftarrow$ empty list of strings
4: **for** each key in $keys$ **do**
5: $itr \leftarrow$ levelDB.GetIterator(nil, nil)
6: $GIk \leftarrow key$
7: $GIBytes \leftarrow$ db.get(GIk)
8: $bNum \leftarrow$ decodeGI($GIBytes$)
9: $dataKey \leftarrow$ constructDataKey($ns, bNum, key$)
10: $found \leftarrow itr.Seek(dataKey)$
11: **if not** $found$ **then**
12: **return** error
13: **end if**
14: $prev, txs \leftarrow$ decodeNewIndex($dbItr.Value()$)
15: $txIndex \leftarrow length(txs) - 1$
16: $keyMap[key] \leftarrow$ new keyData with fields:
17: $(itr, bNum, prev, txs, txIndex)$
18: $validKeys$.append(key)
19: **end for**
20: $scanner \leftarrow$ new parallelHistoryScanner with $(ns, validKeys, keyMap)$
21: **return** $scanner$
22: **end procedure**

Chapter 6

Experimental Evaluation

This section evaluates the performance of `IS` on the datasets of different size and compares with different systems, including *HLF* and its optimized version (*HLF+*). *HLF+* represents a specifically tailored version of *HLF* that supports new APIs. This adaptation is achieved without proposed indexing techniques. Instead, *HLF+* employs a counter-based method to execute the query process. Furthermore, `IS` is compared against recent solutions from academia: *vChain+* and *Lineage Chain*. We have also considered traditional solutions such as *MySQL* and *LevelDB* in our comparison. We used a server configuration comprising of Intel(R) Xeon(R) eight-core CPU E5-2650 v2, 32 GB of RAM. The server operates under Ubuntu 22.04.3 LTS. Our solution uses *HLF* v2.4. All following results are the average of five observations.

6.1 Datasets Description

We evaluate the `IS` on two specifically selected datasets designed to simulate traditional database environments and blockchain-specific data structures.

TPC-H Dataset. It is derived from the TPC-H benchmark that offers a robust framework for database performance assessment. We specifically focus on the *LINEITEM* table, the largest of TPC-H's eight tables, featuring 16 fields. This table represents complexities of an e-commerce database with each *ORDERKEY* potentially having up to seven historical versions. We used the TPC-H data generation tool to generate datasets containing *1 million* and *10 million* records.

Ethereum Dataset. To complement the TPC-H benchmark, we also use the data intrinsic to the blockchain world. We established a full Ethereum node on-premises and extracted its data. The dataset, characterized by 19 fields (such as *FROM*, *TO* and *BLOCKHASH*) encompasses thousands of transactions. We organized it into datasets containing *1 million* and *10 million* records. Such kind of data helps to evaluate the `IS` in environment where keys might have up to 28,900 historical versions.

6.2 Query Specifications

We selected several query types to measure blockchain query capabilities, addressing common data retrieval needs.

- **Point Query:** This query takes an asset key, desired version number and retrieves that specific historical version of an asset. To evaluate that query, in addition to *1 million* and *10 million* datasets, we also used intermediate *2 million*, *4 million*, *6 million* and *8 million* for both TPC-H and Ethereum datasets. For TPC-H datasets, we are retrieving the seventh historical version for key 9,013,472 (*1 million*, *2 million*, *4 million*) because all seven versions of this key are spread across the first 4 million dataset records. For the rest of datasets we selected another key, 23,488 (*6 million*, *8 million*, *10 million*). Its historical versions are spread across the entire 10 million records. As for Ethereum datasets there are no limitations on number of historical versions. We chose to query various historical versions of the same key (*0xf89d7b9c864f589bbf53a82105107622b35eaa40*). We start with the 1,900th historical version for the *1 million* dataset and go up to the 14,000th historical version for the *10 million* dataset.

- **Version Query:** This query takes an asset key, desired version range and retrieves all historical versions of an asset in the range. It is useful for analyzing the chronological development of an asset's state. For TPC-H datasets, we query version ranges of 3, 5 and 7 for both *1 million* and *10 million* datasets, using the same keys as in the point query. For the *1 million* Ethereum dataset, we query 1,000, 2,000 and 3,000 ranges. For the *10 million* Ethereum dataset, we query 1,000, 4,000, 7,000 and 10,000 ranges. Asset key used is the same as in point query. `IS` uses **GHFVR** API for both point and version queries.

- **Block Range Query:** This query retrieves all the data records that have undergone at least some specified number of updates within provided block range. Our performance tests involve various block ranges for both TPC-H and Ethereum datasets. For the *1 million* TPC-H dataset, we test a 100 block range. In contrast, for the *10 million* TPC-H dataset, we test 100,

200, 300, 400 and 500 block ranges. For the *1 million* and *10 million* Ethereum datasets, we test 100, 150 and 200 block ranges. `IS` uses **GHFBR** API for block range queries.

- **Key Range Query:** This query retrieves all historical versions for all the keys within specified consecutive key range. We evaluate only Ethereum datasets due to its complexity and extensive amount of historical data. For *1 million* dataset, we test key ranges of 10 (101 historical versions), 300 (974 historical versions) and 1000 (3508 historical versions). For a *10 million* dataset, we test ranges of 10 (100 historical versions), 100 (1048 historical versions), 300 (3384 historical versions) and 1000 keys (4508 historical versions). `IS` utilizes **GHFKR** API for key range queries. It is worth noting that both block range and key range queries are unsupported in *HLF* due to inherent limitations.

6.3 Data Ingestion Process

At first, we were inserting data sequentially in chunks (2,500 records each). However, this method proved to be inefficient due to the bottleneck in data throughput (insert one record at a time), which significantly extends insertion time. For example, it takes 323 minutes to insert *1 million* Ethereum records. To overcome this limitation, we moved to parallel data processing. We use concurrency features, such as *goroutines* and channels. This modification significantly boosts data insertion. For example, it takes only 11.2 minutes to insert *1 million* Ethereum data. For TPC-H datasets, we select *ORDERKEY* field as a key and combine the rest of the fields as a value. For Ethereum datasets, we select the *FROM* address field as a key and combine the rest of data as a value.

Table 6.1: Data Ingestion Time Comparison

Dataset	Original Index				Version-Based Index				Block-Based Index			
	Sequential		Parallel		Sequential		Parallel		Sequential		Parallel	
	1M	10M	1M	10M	1M	10M	1M	10M	1M	10M	1M	10M
TPC-H (Min)	142	1429	**4.2**	**56.1**	525	5287	**16.4**	**210.9**	525	5287	**16.4**	**207.3**
Ethereum (Min)	323	3230	**11.2**	**119.1**	1195	11951	**21.7**	**212.6**	1195	11951	**21.9**	**214.2**

Table 6.1 demonstrates our data insertion time observations. Sequential insertion of data is time-consuming, taking approximately 2.4 hours for *1 million* and 23.8 hours for *10 million* TPC-H records. The insertion times for Ethereum data are notably higher due to its more extensive nature, requiring around 5.3 hours for *1 million* records and 53.8 hours for *10 million* records.

Parallel data insertion approach has led to significant in insertion time. For example, it reduces insertion time for Ethereum data down to 11.2 minutes for *1 million* records and 119.1 minutes for *10 million* records. An introduction of `VBI` and `BBI` adds complexity to the transaction commit and thus, increases the insertion time. Within TPC-H data, insertion time increases to 16.4 minutes for 1 million records for both `VBI` and `BBI`, compared to 4.2 minutes for the *HLF*. For *10 million* records, insertion time increases to 207 or 210 minutes for `VBI` and `BBI`, compared to 56.1 minutes in *HLF*. As for Ethereum data, the insertion time for `VBI` and `BBI` is around 21 minutes for *1 million*, compared to 11.2 minutes in the original index.

Although the increase in insertion time for both datasets is substantial, it is acceptable compared the time it would take to insert data using sequential approach. Sequential insertion with either `VBI` or `BBI` would require at least 19.9 hours for *1 million* records and at least 199 hours for *10 million* Ethereum records. It is worth noting that in the context of permissioned blockchains, data ingestion is one-time event. Once data is inserted into the blockchain, user does not need to repeat that process. Therefore, although proposed `VBI` and `BBI` negatively affect the data insertion time, we utilize a parallel data insertion that mitigate this downside.

6.4 IS Evaluation

6.4.1 Exp 1. Index Space Overhead

The index space overhead for both the TPC-H and Ethereum datasets is depicted on Table 6.2 and Table 6.3. For the TPC-H dataset, the insertion of *1 million* records (436 MB of raw data) results in a space overhead of 11.7 MB for both *HLF* and *HLF+*. This value increases to 129.7 MB for *10 million* records (4.1 GB of raw data). Proposed `VBI` and `BBI` exhibit overheads of 14.9 MB and 17.2 MB for *1 million* records, respectively. These values increase to 165.3 MB and 188.6

MB for *10 million* records. Despite the apparent increase in space, such rise is marginal compared to the size of the raw data.

Although Ethereum dataset is bigger itself in terms of raw data volume (2.225 GB for *1 million* records and and 20.09 GB for *10 million*), an overhead added by `VBI` and `BBI` remains proportionally minor compared to the raw data size. More specifically, *HLF* and *HLF+* create a space overhead of 28.2 MB for *1 million* records, which scales up to 197.1 MB for *10 million* records. The `VBI` creates 45 MB of overhead for *1 million* records and 280.6 MB for *10 million* records. The `BBI`, on the other hand, results in higher overheads of 60.8 MB for *1 million* records and 522.7 MB for *10 million* records. Although `BBI` introduces more space overhead compared to `VBI`, this overhead still represents only a minor portion of the raw data inserted. Figure 6.1 illustrates the variation in the index space overhead across different datasets.

Table 6.2: Point Query Performance on TPC-H Data

System	Index Overhead[1]		Average Query Time (S)	
	1M	10M	1M	10M
HLF	11.7 MB	129.7 MB	0.046	0.049
HLF+			0.029	0.033
VBI	14.9 MB	165.3 MB	**0.019**	**0.019**
BBI	17.2 MB	188.6 MB	0.028	0.027

6.4.2 Exp 2. Point Query Performance

Point query performance observations for both TPC-H and Ethereum datasets are presented in Table 6.2 and Table 6.3. For the TPC-H data it was observed that the average query time for *1 million* and *10 million* data is consistently similar across evaluated systems. For example, *HLF+* takes 0.029 seconds for *1 million* records and 0.033 seconds for *10 million* records, which

[1]Space overhead created by index techniques while inserting 1M (436 MB) and 10M (4.1 GB) TPCH data.

[2]Space overhead created by index techniques while inserting 1M (2.225 GB) and 10M (20.09 GB) Ethereum data.

Table 6.3: Point Query Performance on Ethereum Data

System	Index Overhead[2]		Average Query Time (S)	
	1M	10M	1M	10M
HLF	28.2 MB	197.1 MB	6.621	N/A
HLF+			0.028	0.046
VBI	45 MB	280.6 MB	**0.019**	**0.022**
BBI	60.8 MB	522.7 MB	0.035	0.352

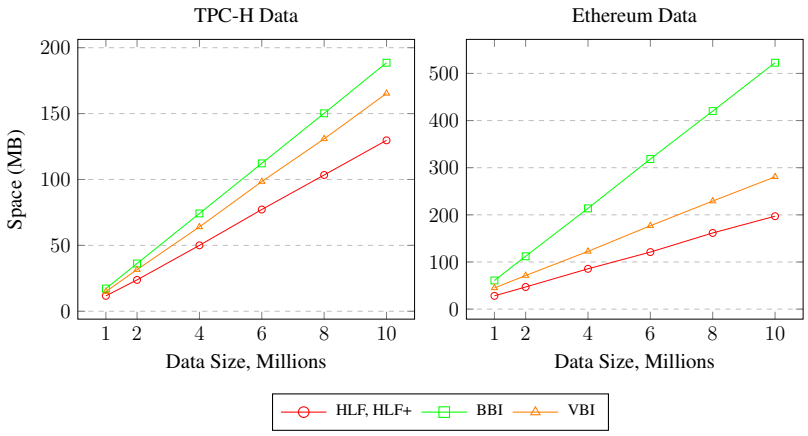

Figure 6.1: Space Overhead.

is significantly faster compared to 0.046 seconds and 0.049 seconds in *HLF*. Our proposed `VBI` and `BBI` achieves even better results. Specifically, `VBI` is the most effective. It takes 0.019 seconds for both *1 million* and *10 million* records, marking a significant enhancement in performance compared to *HLF* and *HLF+*.

In terms of the Ethereum data, it takes 6.621 seconds for *HLF* to execute query on *1 million* records, while *HLF* fails to successfully execute the query on *10 million* records. It happens due to inherent limitations, such as query timeouts. In contrast, *HLF+* not only shows significant performance improvement but also successfully executes query on *10 million* records. Our proposed `BBI` experiences longer query times for both *1 million* and *10 million* records (0.035 seconds and 0.352 seconds, respectively) compared to the *HLF+* results. Such increase is due to `BBI's` methodical traversal through all the blocks in search of required version. Given that the Ethereum

dataset has thousands of historical versions, it takes time to traverse all the blocks. Contrarily, `VBI` demonstrates best performance for the Ethereum data. It achieves 0.019 seconds for the *1 million* records (32% gain compared to *HLF+*) and 0.022 seconds for the *10 million* records (double the performance of *HLF+*). Figure 6.2 illustrates our observations.

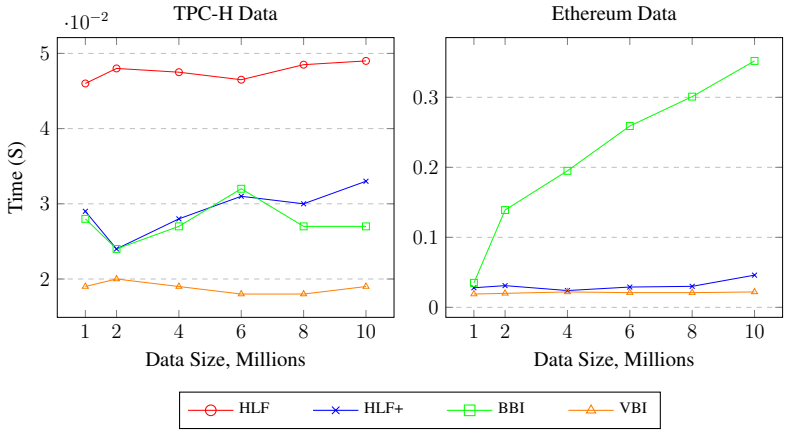

Figure 6.2: Point Query Performance.

6.4.3 Exp 3. Version Query Performance

Version query performance observations for both TPC-H and Ethereum datasets are presented in Table 6.4 and Table 6.5.

For the TPC-H data, we note a comparable level of performance across all evaluated systems. For example, with *10 million* records, *HLF+* achieves 0.033 seconds for a version range of 3, 0.035 seconds for a range of 5 and 0.033 seconds for a range of 7. This pattern mirrors the observations from the point query and can be explained by having only seven historical versions at maximum for both *1 million* and *10 million* datasets. `VBI` shows the best performance, followed by `BBI`, with *HLF+* and *HLF* being a slower options. Figure 6.3 graphically illustrates these findings.

In terms of the Ethereum datasets, *HLF* takes more than 6 seconds across all ranges for the *1 million* records to execute query. Similarly to the point query, *HLF* fails to execute query on *10 million* dataset. *HLF+* enables query execution on the *10 million* dataset and enhances the query performance for the *1 million* records. `VBI` achieves best result of 8.8 seconds for the broadest

range tested, thereby realizing a 37% performance gain comparing to *HLF+*. `BBI` demonstrates a similar trend observed in point query and takes more time to execute query. Figure 6.3 and Figure 6.4 visually present our observations.

Table 6.4: Version Query Performance on TPC-H Data

System	Average Query Time (S)					
	1M			10M		
	Range 3 Versions 1-3	Range 5 Versions 1-5	Range 7 Versions 1-7	Range 3 Versions 1-3	Range 5 Versions 1-5	Range 7 Versions 1-7
HLF		0.046			0.049	
HLF+	0.027	0.032	0.035	0.033	0.035	0.033
VBI	**0.021**	**0.024**	**0.026**	**0.02**	**0.019**	**0.021**
BBI	0.025	0.026	0.029	0.026	0.027	0.029

Table 6.5: Version Query Performance on Ethereum Data

System	Average Query Time (S)						
	1M			10M			
	Range 1K 1K-2K	Range 2K 1K-3K	Range 3K 1K-4K	Range 1K 1-1K	Range 4K 1-4K	Range 7K 1-7K	Range 10K 1-10K
HLF		6.621			N/A		
HLF+	1.916	2.968	4.422	2.545	6.741	10.929	14.085
VBI	**1.373**	**2.523**	**3.416**	**1.277**	**3.121**	**6.827**	**8.800**
BBI	2.180	3.209	4.703	2.829	7.230	13.001	17.174

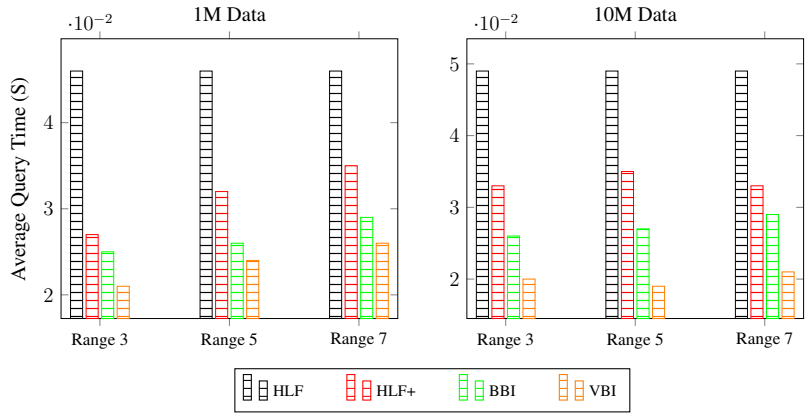

Figure 6.3: Version Query Performance on TPC-H Data.

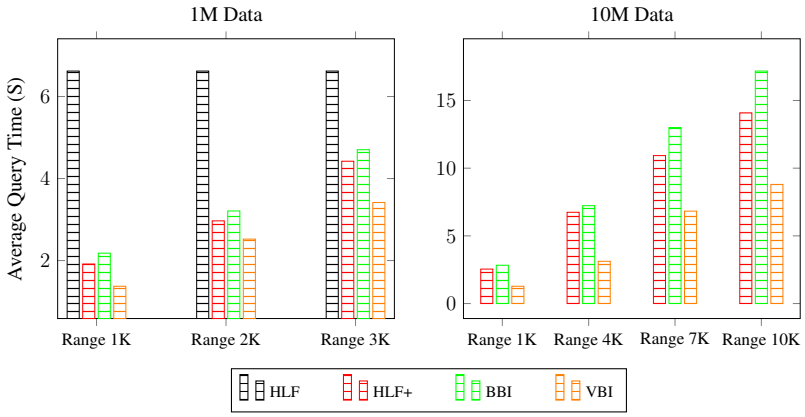

Figure 6.4: Version Query Performance on Ethereum Data.

6.4.4 Exp 4. Block Range Query Performance

Block range query performance observations for both TPC-H and Ethereum datasets are presented in Table 6.6 and Table 6.7. Original *HLF* does not support such operations, so we added block range query support in *HLF+*.

For the TPC-H data and the block range of 100 over *1 million* records, *HLF+* achieves 1.264 seconds. Meanwhile, `VBI` and `BBI` show times of 1.149 and 0.624 seconds, respectively. This demonstrates a nearly twofold performance increase by `BBI` over *HLF+*. For a *10 million* dataset, within the same block range *HLF+* takes 1.335 seconds, `VBI` takes 1.21 seconds and `BBI` takes 0.749 seconds. As the block range increases to 200, the performance gain of `BBI` over *HLF+* achieves 30%. For a block range of 300, this advantage extends to 36%. With a block range of

400 and 500, *BBI's* performance gain reaches 33%. These findings are graphically depicted in Figure 6.5.

For the Ethereum data, employing similar block ranges, we observe the same trend as in TPC-H data. With a block range of 100 for *1 million* records, we observe 2.854 seconds for *HLF+* achieves, 2.836 seconds for *VBI* and 2.013 seconds for *BBI*. These results mark 29% performance boost by *BBI's* over *HLF+*. As the block range extends to 150 and 200, *BBI's* performance gain achieves 30%. For the *10 million* records, *BBI* still shows performance gain of 29%. Our observations underscore *BBI's* superior performance in handling block range queries across all datasets tested, as visually presented in Figure 6.6.

Table 6.6: Block Range Query Performance on TPC-H Data

System	Average Query Time (S)					
	1M	10M				
	Range 100 Blocks 100-200	Range 100 Blocks 100-200	Range 200 Blocks 100-300	Range 300 Blocks 100-400	Range 400 Blocks 100-500	Range 500 Blocks 100-600
HLF+	1.264	1.335	3.518	9.428	16.541	22.98
VBI	1.149	1.210	3.437	7.341	13.905	19.65
BBI	**0.624**	**0.749**	**2.453**	**6.020**	**11.087**	**15.34**

Table 6.7: Block Range Query Performance on Ethereum Data

System	Average Query Time (S)					
	1M			10M		
	Range 100 Blocks 100-200	Range 150 Blocks 100-250	Range 200 Blocks 100-300	Range 100 Blocks 100-200	Range 150 Blocks 100-250	Range 200 Blocks 100-300
HLF+	2.854	7.406	11.973	2.663	7.869	12.528
VBI	2.836	7.074	11.768	2.887	7.595	11.450
BBI	**2.013**	**5.239**	**8.270**	**1.862**	**5.447**	**8.850**

6.4.5 Exp 5. Key Range Query Performance

In terms of key range query, it is important to note that this query requires an access to all historical versions for a range of keys. Such process involves visiting and reading every historical

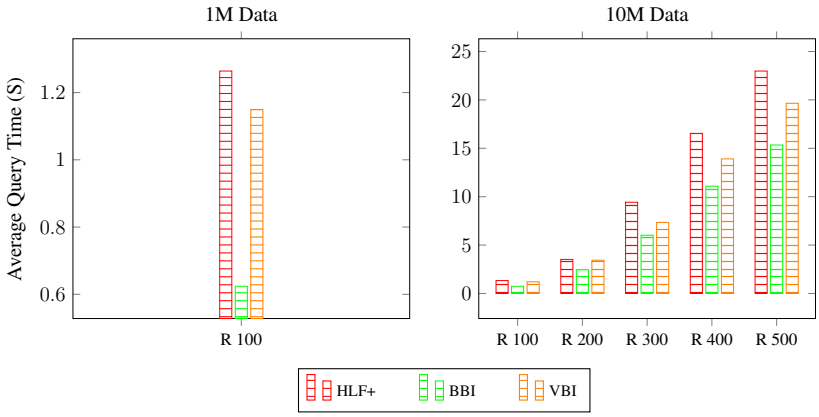

Figure 6.5: Block Range Query Performance on TPC-H Data.

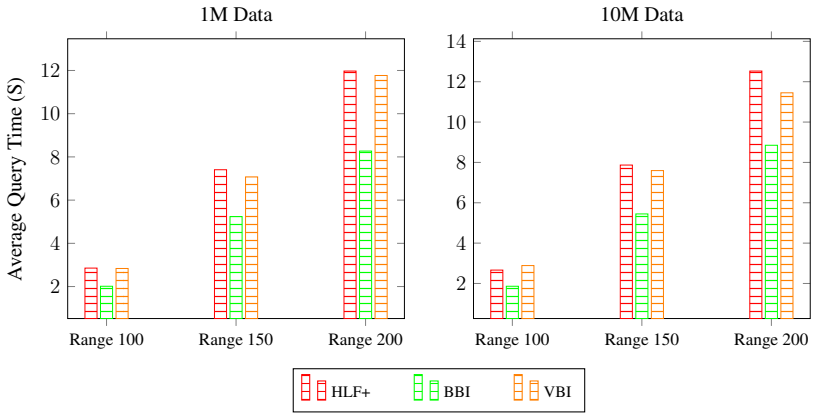

Figure 6.6: Block Range Query Performance on Ethereum Data.

version of a key as well as every indexing record. It means that our proposed indexing techniques do not provide direct performance benefits, however `VBI` and `BBI` are created with several optimizations in mind, so we still observe performance improvement. Table 6.8 summarizes our observations, while Table 6.7 visualizes these findings. For this experiment, we test only Ethereum datasets since they have more historical versions per key.

Performance comparison between *HLF+* and `VBI` reveals that `VBI` shows superior efficiency. For example, in case of *10 million* Ethereum records, *HLF+* shows an execution time of 8.264 seconds for a range of 1,000 keys, whereas `VBI` achieves a better time of 7.564 seconds. This experiment underscores that, even in a worst-case scenario where executing a query necessitates traversing the whole transaction log, `VBI` outperforms the optimized *HLF+* and, by extension, the original *HLF*. Contrarily, `BBI` demonstrates a significant decline in performance. In case of the

same *10 million* records and a range of 1,000 keys, it takes 46.039 seconds for `BBI` to execute query. Such inefficiency happens because `BBI` necessitates block-based traversal, which takes additional time comparing to the original *HLF*.

Table 6.8: Key Range Query Performance on Ethereum Data

System	Average Query Time (S)						
	1M			10M			
	Range 10 101 Entries	Range 300 974 Entries	Range 1000 3508 Entries	Range 10 100 Entries	Range 100 1048 Entries	Range 300 3384 Entries	Range 1000 4508 Entries
HLF+	0.163	1.508	5.517	0.202	1.856	5.84	8.264
VBI	**0.147**	**1.484**	**5.202**	**0.178**	**1.779**	**5.451**	**7.564**
BBI	0.174	1.534	5.667	0.222	1.959	12.767	46.039

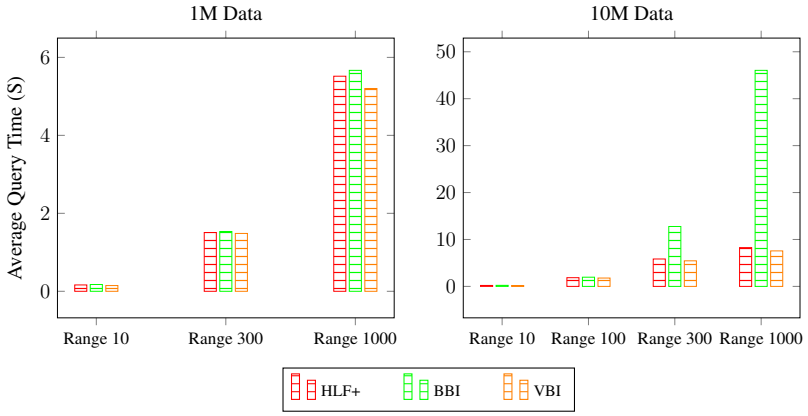

Figure 6.7: Key Range Query Performance.

6.5 Comparison with non-Blockchain Solutions

We extend the evaluation of `IS` by comparing it with non-blockchain solutions: *LevelDB* and *MySQL*. We focus exclusively on the Ethereum dataset due to its complexity and huge number of historical versions. We apply some of the point and version queries previously used to both *LevelDB* and *MySQL*. Observations are presented in Table 6.9.

For both dataset sizes, proposed `VBI` and `BBI` demonstrate competitive performance compared to *MySQL*. However, the challenge arises for smaller dataset sizes. For example, it takes 0.425 seconds for *MySQL* to execute a version query for a range of 1,000. `VBI` requires 1.373 seconds to finish the same query. However, `VBI` outperforms *MySQL* as the dataset size increases to *10 million* records. *LevelDB's* performance is incomparably superior not only to our proposed `IS` but also to *MySQL*. In the context of the *LevelDB's*, it is important to acknowledge that *HLF* and by extension `IS`, incorporates *LevelDB* internally. It means that it is unlikely for a system to surpass the performance of one of its components.

Table 6.9: Comparison with Non-Blockchain Solutions

System	Point		Version (Range 1K)	
	1M	10M	1M	10M
HLF	6.621	N/A	6.621	N/A
HLF+	0.028	0.046	1.916	2.545
VBI	0.019	0.022	1.373	1.277
BBI	0.035	0.352	2.180	2.829
LevelDB	**0.00357**	**0.00526**	**0.00385**	**0.00671**
MySQL	0.195	1.112	0.425	2.112

6.6 Comparison with Other Works

We also compare the `IS` against state-of-the-art solutions from academia: *vChain+* and *Lineage Chain*, using Ethereum datasets. These systems are evaluated on some of the metrics from previous experiments. More specifically, we test them in point queries, version queries (range of 1,000) and block range queries (range of 150). Observations are presented in Table 6.10.

Both *vChain+* and *Lineage Chain* create significantly higher space overheads, with *Lineage Chain* and *vChain+* generating 24.5 GB and 30 GB (for *10 million* records). Such rise is attributed to the maintenance of framework-specific indexing structures, such as skip lists in *Lineage Chain*.

In terms of query performance, *Lineage Chain* demonstrates results comparable to `VBI` for both dataset sizes. Specifically, for 10 million records, *Lineage Chain* achieves a point query

execution time nearly identical to the result of `VBI` (0.027 seconds for *Lineage Chain* versus 0.022 seconds for `VBI`). *vChain+* performance depends on the window size and demonstrates a trade-off between query execution time and index overhead. For our experiments, we chose window size of 32, which represents the balance between query performance, size overhead and blockchain build time. *Lineage Chain's* performance in version queries is on par with `VBI`), while *vChain+* struggles. *vChain+* shows an execution time of 18.5684 seconds and 191.422 seconds for *1 million* and *10 million* records, respectively. However, for block range queries, *vChain+* demonstrates better results than `BBI`. Such observation is anticipated since *vChain+* main design focus is on optimizing block range queries. Unlike *vChain+*, `VBI`) and `BBI`), *Lineage Chain* does not support block range queries.

Table 6.10: Comparison with Other Blockchain Solutions

System	Index Overhead		Average Query Time (S)					
			Point		Version (Range 1K)		Block Range (Range 150)	
	1M	10M	1M	10M	1M	10M	1M	10M
HLF	28.2 MB	197.1 MB	6.621	N/A	6.621	N/A	N/A	N/A
HLF+			0.028	0.046	1.916	2.545	7.406	7.869
VBI	45 MB	280.6 MB	**0.019**	**0.022**	**1.373**	**1.277**	7.074	7.595
BBI	60.8 MB	522.7 MB	0.035	0.352	2.180	2.829	5.239	5.447
Lineage Chain	2.2 GB	30 GB	0.024	0.027	1.632	2.072	N/A	N/A
vChain+	1.7 GB	24.5 GB	7.756	95.198	18.568	191.422	**3.257**	**4.51**

Chapter 7

Conclusion

7.1 Lessons Learned

In this thesis we aimed to design and implement an Indexing Solution (*IS*) to overcome the existing challenges of inadequate query retrieval and a limited variety of supported queries in blockchain systems with focus on Hyperledger Fabric *HLF*. In *HLF*, retrieving a specific piece of historical data requires traversing the entire transaction history. Moreover, this system inherently supports only a limited set of queries.

In order to address these challenges, we analysed *HLF's* internal indexing mechanisms and proposed two indexing algorithms: the Version-Based Index (*VBI*) and the Block-Based Index (*BBI*). Furthermore, we introduced several new query APIs, including **GetHistoryForVersionRange**, **GetHistoryForBlockRange** and **GetHistoryForKeyRange**. They efficiently use our new indexing methods to improve query performance. Our approach to the design of *IS* emphasizes total backward compatibility, we do not alter or break any existing subsystem of *HLF*. Such strategy not only preserves the integrity of verification and consensus mechanisms but also allows for the continued use of all existing *HLF* APIs. Lastly, our minimal change philosophy highlights the potential adoption of our proposed solutions in production environments.

Our evaluation of the proposed *IS*, in comparison with the *HLF* and an optimized version, *HLF+*, revealed that *HLF+* significantly improves query execution times. For some queries *HLF+* boosts performance by more than 100 times, from 6.621 seconds to 0.028 seconds. Our indexing methods further improve this performance, reducing query times to 0.019 seconds and enable access to a broader range of queries. We also observed that while *VBI* and *BBI* introduce additional space overhead due to the new indexing structures, this overhead is negligible compared to the size of the inserted data. Although data insertion with *VBI* and *BBI* is significantly slower using

the default sequential insertion method of *HLF*, we mitigated this effect by using a parallel data insertion.

Moreover, we extended our comparison to non-blockchain systems (*LevelDB* and *MySQL*). Our `VBI` and `BBI` outperformed *MySQL* in terms of point and version queries, but could not outperform *LevelDB*, which was more then 100 times faster. We have also compared our `IS` with blockchain solutions (*Lineage Chain* and *vChain+*). Although these systems have their own application areas, our proposed solutions demonstrated superior space efficiency and comparable query performance. For example, while *Lineage Chain* matched the performance of *HLF+*, our `BBI` outperformed *vChain+* for datasets of 10 million entries.

7.2 Future Work

The work presented in this thesis marks a step towards enhancing query performance and broadening the scope of supported queries within *HLF*, without sacrificing its existing functionalities. These achievements not only underscore the potential for practical application in blockchain systems, but also highlight several promising directions for future research.

One pivotal area for future exploration involves a rework of *HLF's* system modules. A comprehensive overhaul of the data layer could potentially lead to even more substantial improvements in query performance and the diversity of supported query types. Such work would require meticulous examination of the underlying *HLF* architecture.

Another promising direction for future work is the generalization and adaptation of `IS` to other blockchain frameworks. This would not only validate the versatility of proposed work but also contribute to a broader understanding of blockchain indexing mechanisms across various platforms.

Furthermore, a deeper investigation into the nature and query requirements for blockchain systems represents a possible area for future research. A detailed analysis based on real-world industry use cases would expose specific query types that are most relevant and critical for industry applications.

Bibliography

[1] Jameela Al-Jaroodi and Nader Mohamed. Blockchain in Industries: A Survey. *IEEE Access*, 7:36500–36515, 2019.

[2] Guy Zyskind, Oz Nathan, and Alex 'Sandy' Pentland. Decentralizing Privacy: Using Blockchain to Protect Personal Data. In *2015 IEEE Security and Privacy Workshops*, pages 180–184, 2015.

[3] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: a Distributed Operating System for Permissioned Blockchains. In *ACM Proceedings*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[4] Linux Foundation. https://www.linuxfoundation.org/, 2000. Accessed: 2024-04-28.

[5] Hyperledger. Benchmarking Hyperledger Fabric 2.5 Performance. https://www.hyperledger.org/blog/2023/02/16/benchmarking-hyperledger-fabric-2-5-performance, February 2023.

[6] Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org. Accessed: 2024-02-21.

[7] Hyperledger. Hyperledger Walmart Case Studies. https://www.hyperledger.org/case-studies/walmart-case-study, 2023. Accessed: 2023-10-19.

[8] Hyperledger. Hyperledger Hitachi Case Studies. https://www.hyperledger.org/case-studies/hitachi-case-study, 2023. Accessed: 2023-10-19.

[9] Hyperledger. Hyperledger Bosch Case Studies. https://www.hyperledger.org/case-studies/bosch-case-study, 2023. Accessed: 2023-10-19.

[10] Hyperledger. Hyperledger Honeywell Case Studies. https://www.hyperledger.org/case-studies/honeywell-case-study, 2023. Accessed: 2023-10-19.

[11] Hyperledger. Hyperledger Tech Mahindra Case Studies. https://www.hyperledger.org/case-studies/techmahindra-case-study, 2023. Accessed: 2023-10-19.

[12] Himanshu Gupta, Sandeep Hans, Kushagra Aggarwal, Sameep Mehta, Bapi Chatterjee, and Praveen Jayachandran. Efficiently Processing Temporal Queries on Hyperledger Fabric. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1489–1494, 2018.

[13] Mohammad Jabed Morshed Chowdhury, Alan Colman, Muhammad Ashad Kabir, Jun Han, and Paul Sarda. Blockchain Versus Database: A Critical Analysis. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 1348–1353, 2018.

[14] Avinash Lakshman and Prashant Malik. Cassandra: a Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[15] MongoDB, Inc. MongoDB Documentation. https://www.mongodb.com/docs/manual/introduction/, 2009. Accessed: 2024-03-10.

[16] Couchbase Under The Hood. https://info.couchbase.com/rs/302-GJY-034/images/Couchbase_Under_The_Hood_WP.pdf. Accessed: 2024-03-12.

[17] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3), August 2013.

[18] Eric Brewer. A Certain Freedom: Thoughts on the CAP Theorem. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, page 335, New York, NY, USA, 2010. Association for Computing Machinery.

[19] Artem Chebotko, A. Kashlev, and Shiyong Lu. A Big Data Modeling Methodology for Apache Cassandra. *2015 IEEE International Congress on Big Data*, pages 238–245, 2015.

[20] A. Wahid and Kanupriya Kashyap. Cassandra—A Distributed Database System: An Overview. *Advances in Intelligent Systems and Computing*, 2018.

[21] Vivek Mishra. *Beginning Apache Cassandra Development*. Apress, 2014.

[22] Salvatore Dipietro, R. Buyya, and G. Casale. PAX: Partition-aware autoscaling for the Cassandra NoSQL database. *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9, 2018.

[23] Deepak Vohra. Using Couchbase Server. In *Pro Docker*, pages 81–93. Apress, 2016.

[24] Raul Estrada and Isaac Ruiz. Storage: Apache Cassandra. In *Big Data SMACK: A Guide to Apache Spark, Mesos, Akka, Cassandra, and Kafka*, pages 67–95. Apress, 2016.

[25] Yasaswi Kishore, N. Datta, K. Subramaniam, and D. Sitaram. QoS Aware Resource Management for Apache Cassandra. *2016 IEEE 23rd International Conference on High Performance Computing Workshops (HiPCW)*, pages 3–10, 2016.

[26] William Schultz, Tess Avitabile, and A. Cabral. Tunable Consistency in MongoDB. *Proc. VLDB Endow.*, 12:2071–2081, 2019.

[27] Hongrong Ouyang, Hengfeng Wei, and Yu Huang. Checking Causal Consistency of MongoDB. *Journal of Computer Science and Technology*, 37:128–146, 2020.

[28] David Hows, Peter Membrey, Eelco Plugge, and Tim Hawkins. Introduction to MongoDB. In *The Definitive Guide to MongoDB: A complete guide to dealing with Big Data using MongoDB*, pages 3–16. Apress, Berkeley, CA, 2 edition, 2013.

[29] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, and Guohui Xiao. Expressivity and Complexity of MongoDB Queries. In *21st International Conference on Database Theory (ICDT 2018)*, volume 98 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

[30] P. Murugesan and I. Ray. Audit Log Management in MongoDB. *2014 IEEE World Congress on Services*, pages 53–57, 2014.

[31] Yunhua Gu, Xing Wang, S. Shen, Jin Wang, and Jeong-Uk Kim. Analysis of Data Storage Mechanism in NoSQL Database MongoDB. *2015 IEEE International Conference on Consumer Electronics - Taiwan*, pages 70–71, 2015.

[32] Ajit Singh. Data Migration from Relational Database to MongoDB using XAMPP and NoSQL. *SSRN Electronic Journal*, 2019. Preprint available at SSRN: https://ssrn.com/abstract=3372802.

[33] Murtadha Al Hubail, Ali Alsuliman, Michael Blow, M. Carey, Dmitry Lychagin, Ian Maxon, and T. Westmann. Couchbase Analytics: NoETL for Scalable NoSQL Data Analysis. *Proc. VLDB Endow.*, 12:2275–2286, 2019.

[34] Deepak Vohra. *Pro Couchbase Development*. Apress, 2015.

[35] Deepak Vohra. Using Couchbase Server. In *Pro Docker*, pages 95–115. Apress, 2016.

[36] Cristina Băzăr and Cosmin Sebastian Iosif. The Transition from RDBMS to NoSQL. A Comparative Analysis of Three Popular Non-Relational Solutions: Cassandra, MongoDB and Couchbase. *Database Systems Journal*, 5:49–59, 2014.

[37] Ciprian-Octavian Truică, F. Rădulescu, A. Boicea, and I. Bucur. Performance Evaluation for CRUD Operations in Asynchronously Replicated Document Oriented Database. *2015 20th International Conference on Control Systems and Computer Science*, pages 191–196, 2015.

[38] MongoDB, Inc. MongoDB Benchmark. https://www.mongodb.com/scale/mongodb-benchmark, 2023. Accessed: 2024-03-13.

[39] Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB, 2019. Accessed: 2024-01-07.

[40] ConsenSys. GoQuorum Documentation. https://docs.goquorum.consensys.io/. Accessed: 2024-03-11.

[41] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, Dec 2008. Accessed: 2024-02-21.

[42] Amjad Aldweesh, Maher Alharby, and Aad van Moorsel. Performance Benchmarking for Ethereum Opcodes. In *2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA)*, pages 1–2, 2018.

[43] Parity Ethereum. https://github.com/openethereum/parity-ethereum, 2020. Accessed: 2024-02-20.

[44] Ethereum Foundation. https://geth.ethereum.org/, 2024. Accessed: 2024-04-28.

[45] Suporn Pongnumkul, Chaiyaphum Siripanpornchana, and Suttipong Thajchayapong. Performance Analysis of Private Blockchain Platforms in Varying Workloads. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6, 2017.

[46] Xiwei Xu, Ingo Weber, Mark Staples, Liming Zhu, Jan Bosch, Len Bass, Cesare Pautasso, and Paul Rimba. A Taxonomy of Blockchain-Based Systems for Architecture Design. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 243–252, 2017.

[47] Carsten Maple and Jack Jackson. Selecting Effective Blockchain Solutions. In *Euro-Par 2018: Parallel Processing Workshops*, pages 392–403, Cham, 2019. Springer International Publishing.

[48] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Ji Wang. Untangling Blockchain: A Data Processing View of Blockchain Systems. *IEEE Transactions on Knowledge and Data Engineering*, 30(7):1366–1385, 2018.

[49] B. C. Ooi et al. BLOCKBENCH: A Framework for Analyzing Private Blockchains. https://github.com/ooibc88/blockbench, 2023. Accessed: 2024-03-20.

[50] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger Fabric: a Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[51] Arati Baliga, Nitesh Solanki, Shubham Verekar, Amol Pednekar, Pandurang Kamat, and Siddhartha Chatterjee. Performance Characterization of Hyperledger Fabric. In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 65–74, 2018.

[52] Murat Kuzlu, Manisa Pipattanasomporn, Levent Gurses, and Saifur Rahman. Performance Analysis of a Hyperledger Fabric Blockchain Framework: Throughput, Latency and Scalability. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 536–540, 2019.

[53] Parth Thakkar and Senthilnathan Natarajan. Scaling Blockchains Using Pipelined Execution and Sparse Peers. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 489–502, New York, NY, USA, 2021. Association for Computing Machinery.

[54] Julian Dreyer, Marten Fischer, and Ralf Tönjes. Performance Analysis of Hyperledger Fabric 2.0 Blockchain Platform. In *Proceedings of the Workshop on Cloud Continuum Services for*

Smart IoT Systems, CCIoT '20, page 32–38, New York, NY, USA, 2020. Association for Computing Machinery.

[55] S. B. Toumia, C. Berger, and H. P. Reiser. Evaluating Blockchain Application Requirements and their Satisfaction in Hyperledger Fabric, 2021.

[56] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. Performance Benchmarking and Optimizing Hyperledger Fabric Blockchain Platform. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 264–276, 2018.

[57] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 455–463, 2019.

[58] Zhijian Qu, Hanlin Wang, Xiang Peng, and Qunfeng Wang. Lineage Chain Mark Fault-Tolerant Method for Micro Batching Monitoring Data in Distribution Power Network. *IEEE Access*, 7:32949–32960, 2019.

[59] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Beng Chin Ooi, and Pingcheng Ruan. Forkbase: an Efficient Storage Engine for Blockchain and Forkable Applications. *Proc. VLDB Endow.*, 11(10):1137–1150, jun 2018.

[60] Bikash Chandra Singh, Qingqing Ye, Haibo Hu, and Bin Xiao. Efficient and Lightweight Indexing Approach for Multi-Dimensional Historical Data in Blockchain. *Future Gener. Comput. Syst.*, 139(C):210–223, February 2023.

[61] Amritesh Kumar, Amrendra Singh Yadav, and Dharmender Singh Kushwaha. VChain: Efficient Blockchain based Vehicular Communication Protocol. In *2020 10th International Conference on Cloud Computing, Data Science and Engineering (Confluence)*, pages 762–768, 2020.

[62] Haixin Wang, Cheng Xu, Ce Zhang, Jianliang Xu, Zhe Peng, and Jian Pei. vChain+: Optimizing Verifiable Blockchain Boolean Range Queries. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*, pages 1927–1940, 2022.

[63] Ce Zhang, Cheng Xu, Jianliang Xu, Yuzhe Richard Tang, and Byron Choi. GEMˆ2-Tree: A Gas-Efficient Structure for Authenticated Range Queries in Blockchain. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 842–853. IEEE, 2019.

[64] Ce Zhang, Cheng Xu, Haixin Wang, Jianliang Xu, and Byron Choi. Authenticated Keyword Search in Scalable Hybrid-Storage Blockchains. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, pages 996–1007, 2021.

[65] Yang Li, Kai Zheng, Ying Yan, Qi Liu, and Xiaofang Zhou. EtherQL: A Query Layer for Blockchain System. In K. Selçuk Candan, Lei Chen, Torben Bach Pedersen, Lijun Chang, and Wen Hua, editors, *Database Systems for Advanced Applications - 22nd International Conference, DASFAA 2017, Suzhou, China, March 27-30, 2017, Proceedings, Part II*, volume 10178 of *Lecture Notes in Computer Science*, pages 556–567. Springer, 2017.

[66] Muhammad Muzammal, Qiang Qu, and Bulat Nasrulin. Renovating Blockchain With Distributed Databases: An Open Source System. *Future Generation Computer Systems*, 90:105–117, 2019.

[67] Ripple Consensus Whitepaper. https://ripple.com/files/ripple_consensus_whitepaper.pdf. Accessed: 2024-03-12.

[68] Jingang Yu, Yongkang Hou, Shu Li, and Zhifeng Wen. A High-Speed Data Retrieval Model on Blockchain. In *2022 11th International Conference of Information and Communication Technology (ICTech))*, pages 101–105, 2022.

[69] Yang Liu, Zehui Xie, Nong Xiao, Zhiguang Chen, and Yang Ou. FlyDB: Query Optimization of Blockchain System Based on Hybrid Storage Architecture. In *2023 4th International Seminar on Artificial Intelligence, Networking and Information Technology (AINIT)*, pages 233–237, 2023.

[70] Gareth Peters and Efstathios Panayi. Understanding Modern Banking Ledgers Through Blockchain Technologies: Future of Transaction Processing and Smart Contracts on the Internet of Money. *SSRN*, 2015.

[71] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, jul 1982.

[72] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

Appendix A

Hyperledger Fabric Code

A.1 Historical Index Construction

```
1  package history
2
3  func (d *DB) Commit(block *common.Block) error {
4      blockNo := block.Header.Number
5      //Set the starting tranNo to 0
6      var tranNo uint64
7
8      dbBatch := d.levelDB.NewUpdateBatch()
9
10     logger.Debugf("Channel [%s]: Updating history database for
       ↪  blockNo [%v] with [%d] transactions", d.name, blockNo,
       ↪  len(block.Data.Data))
11
12     // Get the invalidation byte array for the block
13     txsFilter := txflags.ValidationFlags(block.Metadata)
14
15     // write each tran's write set to history db
16     for _, envBytes := range block.Data.Data {
17         // If the tran is marked as invalid, skip it
18         if txsFilter.IsInvalid(int(tranNo)) {
19             logger.Debugf("Channel [%s]: Skipping history write for
               ↪  invalid transaction number %d", d.name, tranNo)
20             tranNo++
```

```
            continue
        }
        env, err := protoutil.GetEnvelopeFromBlock(envBytes)
        if err != nil {
            return err
        }

        payload, err := protoutil.UnmarshalPayload(env.Payload)
        if err != nil {
            return err
        }

        chdr, err :=
         ↪  protoutil.UnmarshalChannelHeader(payload.Header)
        if err != nil {
            return err
        }

        if common.HeaderType(chdr.Type) ==
         ↪  common.HeaderType_ENDORSER_TRANSACTION {
            // extract RWSet from transaction
            respPayload, err :=
             ↪  protoutil.GetActionFromEnvelope(envBytes)
            if err != nil {
                return err
            }
            txRWSet := &rwsetutil.TxRwSet{}
```

```
45              if err = txRWSet.FromProtoBytes(respPayload.Results);
                ↪  err != nil {
46                  return err
47              }
48              // add a history record for each write
49              for _, nsRWSet := range txRWSet.NsRwSets {
50                  ns := nsRWSet.NameSpace
51
52                  for _, kvWrite := range nsRWSet.KvRwSet.Writes {
53                          dataKey := constructDataKey(ns, kvWrite.Key,
                            ↪  blockNo, tranNo)
54                          // No value is required, write an empty byte
                            ↪  array (emptyValue) since Put() of nil is
                            ↪  not allowed
55                            dbBatch.Put(dataKey, emptyValue)
56                          }
57              }
58
59          } else {
60              logger.Debugf("Skipping transaction [%d] since it is not
                ↪  an endorsement transaction\n", tranNo)
61          }
62              tranNo++
63          }
64
65      height := version.NewHeight(blockNo, tranNo)
66      dbBatch.Put(savePointKey, height.ToBytes())
67
```

```
68      if err := d.levelDB.WriteBatch(dbBatch, true); err != nil {
69          return err
70      }
71
72      logger.Debugf("Channel [%s]: Updates committed to history
        ↪  database for blockNo [%v]", d.name, blockNo)
73      return nil
74  }
```

A.2 GetHistoryForKey Implementation

```
1   package history
2
3   type QueryExecutor struct {
4       levelDB    *leveldbhelper.DBHandle
5       blockStore *blkstorage.BlockStore
6   }
7
8   func (q *QueryExecutor) GetHistoryForKey(namespace string, key
    ↪  string) (commonledger.ResultsIterator, error) {
9       rangeScan := constructRangeScan(namespace, key)
10      dbItr, err := q.levelDB.GetIterator(rangeScan.startKey,
        ↪  rangeScan.endKey)
11      if err != nil {
12          return nil, err
13      }
14
```

```
15      // By default, dbItr is in the orderer of oldest to newest and
        ↪  its cursor is at the beginning of the entries.
16      // Need to call Last() and Next() to move the cursor to the end
        ↪  of the entries so that we can iterate
17      // the entries in the order of newest to oldest.
18      if dbItr.Last() {
19          dbItr.Next()
20      }
21      return &historyScanner{rangeScan, namespace, key, dbItr,
        ↪  q.blockStore}, nil
22  }
23
24  type historyScanner struct {
25      rangeScan  *rangeScan
26      namespace  string
27      key        string
28      dbItr      iterator.Iterator
29      blockStore *blkstorage.BlockStore
30  }
31
32  // Next iterates to the next key, in the order of newest to oldest,
    ↪  from history scanner.
33  // It decodes blockNumTranNumBytes to get blockNum and tranNum,
34  // loads the block:tran from block storage, finds the key and
    ↪  returns the result.
35  func (scanner *historyScanner) Next() (commonledger.QueryResult,
    ↪  error) {
```

```
36      // call Prev because history query result is returned from
        ↪  newest to oldest
37      if !scanner.dbItr.Prev() {
38          return nil, nil
39      }
40
41      historyKey := scanner.dbItr.Key()
42      blockNum, tranNum, err :=
        ↪  scanner.rangeScan.decodeBlockNumTranNum(historyKey)
43      if err != nil {
44          return nil, err
45      }
46      logger.Debugf("Found history record for namespace:%s key:%s at
        ↪  blockNumTranNum %v:%v\n",
47      scanner.namespace, scanner.key, blockNum, tranNum)
48
49      tranEnvelope, err :=
        ↪  scanner.blockStore.RetrieveTxByBlockNumTranNum(blockNum,
        ↪  tranNum)
50      if err != nil {
51          return nil, err
52      }
53
54      queryResult, err := getKeyModificationFromTran(tranEnvelope,
        ↪  scanner.namespace, scanner.key)
55      if err != nil {
56          return nil, err
57      }
```

```
58      if queryResult == nil {
59          logger.Errorf("No namespace or key is found for namespace
            ↪  %s and key %s with decoded blockNum %d and tranNum %d",
            ↪  scanner.namespace, scanner.key, blockNum, tranNum)
60          return nil, errors.Errorf("no namespace or key is found for
            ↪  namespace %s and key %s with decoded blockNum %d and
            ↪  tranNum %d", scanner.namespace, scanner.key, blockNum,
            ↪  tranNum)
61      }
62      logger.Debugf("Found historic key value for namespace:%s key:%s
        ↪  from transaction %s",
63      scanner.namespace, scanner.key,
        ↪  queryResult.(*queryresult.KeyModification).TxId)
64      return queryResult, nil
65  }
66
67  func (scanner *historyScanner) Close() {
68      scanner.dbItr.Release()
69  }
70
71  func getKeyModificationFromTran(tranEnvelope *common.Envelope,
    ↪  namespace string, key string) (commonledger.QueryResult, error)
    ↪  {
72      logger.Debugf("Entering getKeyModificationFromTran %s:%s",
        ↪  namespace, key)
73
74      payload, err :=
        ↪  protoutil.UnmarshalPayload(tranEnvelope.Payload)
```

```
75      if err != nil {
76      return nil, err
77      }
78
79      tx, err := protoutil.UnmarshalTransaction(payload.Data)
80      if err != nil {
81          return nil, err
82      }
83
84      _, respPayload, err := protoutil.GetPayloads(tx.Actions[0])
85      if err != nil {
86          return nil, err
87      }
88
89      chdr, err := protoutil.UnmarshalChannelHeader(payload.Header)
90      if err != nil {
91          return nil, err
92      }
93
94      txID := chdr.TxId
95      timestamp := chdr.Timestamp
96
97      txRWSet := &rwsetutil.TxRwSet{}
98
99      if err = txRWSet.FromProtoBytes(respPayload.Results); err !=
        ↪  nil {
100         return nil, err
101     }
```

```
102
103     for _, nsRWSet := range txRWSet.NsRwSets {
104         if nsRWSet.NameSpace == namespace {
105             for _, kvWrite := range nsRWSet.KvRwSet.Writes {
106                 if kvWrite.Key == key {
107                     return &queryresult.KeyModification{TxId: txID,
                        ↪  Value: kvWrite.Value, Timestamp: timestamp,
                        ↪  IsDelete:
                        ↪  rwsetutil.IsKVWriteDelete(kvWrite)}, nil
108                 }
109             }
110             logger.Debugf("key [%s] not found in namespace [%s]'s
                ↪  writeset", key, namespace)
111             return nil, nil
112         }
113     }
114     logger.Debugf("namespace [%s] not found in transaction's
        ↪  ReadWriteSets", namespace)
115     return nil, nil
116 }
```

Appendix B

Version-Based Index Implementation

B.1 Version-Based Index Construction

```
package history

func (d *DB) Commit(block *common.Block) error {
    blockNo := block.Header.Number
    //Set the starting tranNo to 0
    var tranNo uint64

    dbBatch := d.levelDB.NewUpdateBatch()
    dataKeys := make(map[string][]uint64)

    logger.Debugf("Channel [%s]: Updating history database for
     ↪ blockNo [%v] with [%d] transactions", d.name, blockNo,
     ↪ len(block.Data.Data))

    // Get the invalidation byte array for the block
    txsFilter := txflags.ValidationFlags(block.Metadata)

    // write each tran's write set to history db
    for _, envBytes := range block.Data.Data {
        // If the tran is marked as invalid, skip it
        if txsFilter.IsInvalid(int(tranNo)) {
            logger.Debugf("Channel [%s]: Skipping history write for
             ↪ invalid transaction number %d", d.name, tranNo)
```

```
            tranNo++
            continue
        }

        env, err := protoutil.GetEnvelopeFromBlock(envBytes)
        if err != nil {
            return err
        }

        payload, err := protoutil.UnmarshalPayload(env.Payload)
        if err != nil {
            return err
        }

        chdr, err :=
        ↪  protoutil.UnmarshalChannelHeader(payload.Header)
        if err != nil {
            return err
        }

        if common.HeaderType(chdr.Type) ==
        ↪  common.HeaderType_ENDORSER_TRANSACTION {
            // extract RWSet from transaction
            respPayload, err :=
            ↪  protoutil.GetActionFromEnvelope(envBytes)
            if err != nil {
                return err
            }
```

```
46              txRWSet := &rwsetutil.TxRwSet{}
47              if err = txRWSet.FromProtoBytes(respPayload.Results);
                ↪  err != nil {
48                  return err
49              }
50              // add a history record for each write
51              for _, nsRWSet := range txRWSet.NsRwSets {
52               ns := nsRWSet.NameSpace
53               for _, kvWrite := range nsRWSet.KvRwSet.Writes {
54                   var (
55                                   versions     uint64
56                                   transactions []uint64
57                               )
58                               // Get returns nil if key not found
59                               GIkey := constructGlobalIndex(ns,
                                 ↪  kvWrite.Key)
60                               versionsBytes, err := d.levelDB.Get(GIkey)
61                               if err != nil {
62                                   return err
63                               }
64                               if versionsBytes != nil {
65                                    versions, _, err =
                                      ↪  util.Decode(versionsBytes)
66                                    if err != nil {
67                                        return err
68                                    }
69                                    recordedTransactions, present :=
                                      ↪  dataKeys[kvWrite.Key]
```

```
70                              if present {
71                                  transactions =
                                    ↪  recordedTransactions
72                              }
73                          }
74                          versions++
75                          transactions = append(transactions, tranNo)
76
77                          dataKeys[kvWrite.Key] = transactions
78
79                          updatedVersionsBytes :=
                            ↪  util.Encode(versions)
80                          err = d.levelDB.Put(GIkey,
                            ↪  updatedVersionsBytes, true)
81                          if err != nil {
82                              return err
83                          }
84
85                          minVersion := versions -
                            ↪  uint64(len(transactions)) + 1
86                          dataKey := constructDataKey(ns,
                            ↪  kvWrite.Key, minVersion)
87                          indexVal := constructLocalIndex(blockNo,
                            ↪  transactions)
88                          logger.Debugf("Added to dbBatch: %s~%d:
                            ↪  %d~%v\n", kvWrite.Key, minVersion,
                            ↪  blockNo, transactions)
89                              dbBatch.Put(dataKey, indexVal)
```

```
                        }
                }

            } else {
                logger.Debugf("Skipping transaction [%d] since it is not
                ↪  an endorsement transaction\n", tranNo)
            }
                tranNo++
        }

        // add savepoint for recovery purpose
        height := version.NewHeight(blockNo, tranNo)
        dbBatch.Put(savePointKey, height.ToBytes())

        // write the block's history records and savepoint to LevelDB
        // Setting snyc to true as a precaution, false may be an ok
        ↪  optimization after further testing.
        if err := d.levelDB.WriteBatch(dbBatch, true); err != nil {
            return err
        }

        logger.Debugf("Channel [%s]: Updates committed to history
        ↪  database for blockNo [%v]", d.name, blockNo)
        return nil
}
```

B.2 GetHistoryForVersionRange Implementation

```
1  package history
2
3  type versionScanner struct {
4      rangeScan     *rangeScan
5      namespace     string
6      key           string
7      dbItr         iterator.Iterator
8      blockStore    *blkstorage.BlockStore
9      currentBlock uint64
10     transactions []uint64
11     txIndex       int
12     start         uint64
13     end           uint64
14 }
15
16 func (q *QueryExecutor) GetHistoryForVersionRange(namespace string,
   ↪  key string, start uint64, end uint64)
   ↪  (commonledger.ResultsIterator, error) {
17     if end < start {
18         return nil, errors.Errorf("Start: %d is not less than or
           ↪  equal to end: %d", start, end)
19     }
20
21     if end <= 0 || start <= 0 {
22         return nil, errors.Errorf("Start: %d, end: %d cannot be
           ↪  less than 1", start, end)
23     }
```

```
24
25      GIkey := constructGlobalIndex(namespace, key)
26      versionsBytes, err := q.levelDB.Get(GIkey)
27      if err != nil {
28          return nil, errors.Errorf("Error reading from history
            ↪  database for key: %s", key)
29      }
30      if versionsBytes != nil {
31          maxVersion, _, err :=
            ↪  util.DecodeOrderPreservingVarUint64(versionsBytes)
32          if err != nil {
33             return nil, errors.Errorf("Error decoding lasts known
               ↪  version for key: %s", key)
34          }
35          if maxVersion < start {
36             return nil, errors.Errorf("Start: %d is greater than the
               ↪  last existing version: %d", start, maxVersion)
37          }
38      }
39
40      rangeScan := constructRangeScan(namespace, key)
41      endKey := append(rangeScan.startKey,
        ↪  util.EncodeOrderPreservingVarUint64(end+1)...)
42
43      dbItr, err := q.levelDB.GetIterator(rangeScan.startKey, endKey)
44      if err != nil {
45          return nil, err
46      }
```

```
47      var (
48          currentBlock uint64
49          transactions []uint64
50          txIndex      int
51      )
52      if dbItr.Last() {
53          historyKey := dbItr.Key()
54          firstVersionInBlock, err :=
            ↪  rangeScan.decodeMinVersion(historyKey)
55          if err != nil {
56             return nil, err
57          }
58          indexVal := dbItr.Value()
59          currentBlock, transactions, err =
            ↪  decodeLocalIndex(indexVal)
60          if err != nil {
61             return nil, err
62          }
63          lastVersionInBlock := firstVersionInBlock +
            ↪  uint64(len(transactions)-1)
64          if end > lastVersionInBlock {
65             txIndex = int(len(transactions) - 1)
66          } else {
67             txIndex = int(end - firstVersionInBlock)
68          }
69      } else {
70          txIndex = -1
71      }
```

```
72
73      return &versionScanner{rangeScan, namespace, key, dbItr,
        ↪  q.blockStore, currentBlock, transactions, txIndex, start,
        ↪  end}, nil
74  }
75
76  func (scanner *versionScanner) Next() (commonledger.QueryResult,
    ↪  error) {
77      if scanner.txIndex <= -1 {
78          if !scanner.dbItr.Prev() {
79              return nil, nil
80          }
81          indexVal := scanner.dbItr.Value()
82          currentBlock, transactions, err :=
            ↪  decodeLocalIndex(indexVal)
83          if err != nil {
84              return nil, err
85          }
86          scanner.currentBlock = currentBlock
87          scanner.transactions = transactions
88          scanner.txIndex = len(scanner.transactions) - 1
89
90          logger.Debugf("Updated scanner for key: %s, block: %d,
            ↪  transactions: %v, txIndex: %d\n", scanner.key,
            ↪  scanner.currentBlock, scanner.transactions,
            ↪  scanner.txIndex)
91      }
92
```

```
93      historyKey := scanner.dbItr.Key()
94      firstVersionInBlock, err :=
        ↪  scanner.rangeScan.decodeMinVersion(historyKey)
95      if err != nil {
96          return nil, err
97      }
98      currentVersionNum := firstVersionInBlock +
        ↪  uint64(scanner.txIndex)
99      logger.Debugf("First version in block: %d, Current version:
        ↪  %d\n", firstVersionInBlock, currentVersionNum)
100     if currentVersionNum < scanner.start {
101         logger.Debugf("First requested version %d found for key:
            ↪  %s", scanner.start, scanner.key)
102         return nil, nil
103     }
104
105     blockNum := scanner.currentBlock
106     tranNum := scanner.transactions[scanner.txIndex]
107     scanner.txIndex--
108
109     logger.Debugf("Found history record for namespace:%s key:%s at
        ↪  blockNumTranNum %v:%v\n, firstVersionInBlock: %d,
        ↪  currentVersion: %d\n",
110     scanner.namespace, scanner.key, blockNum, tranNum,
        ↪  firstVersionInBlock, currentVersionNum)
111
112     // Get the transaction from block storage that is associated
        ↪  with this history record
```

```
113     tranEnvelope, err :=
        ↪  scanner.blockStore.RetrieveTxByBlockNumTranNum(blockNum,
        ↪  tranNum)
114     if err != nil {
115         return nil, err
116     }
117
118     // Get the txid, key write value, timestamp, and delete
        ↪  indicator associated with this transaction
119     queryResult, err := getKeyModificationFromTran(tranEnvelope,
        ↪  scanner.namespace, scanner.key)
120     if err != nil {
121         return nil, err
122     }
123     if queryResult == nil {
124         // should not happen, but make sure there is inconsistency
            ↪  between historydb and statedb
125         logger.Errorf("No namespace or key is found for namespace
            ↪  %s and key %s with decoded blockNum %d and tranNum %d",
            ↪  scanner.namespace, scanner.key, blockNum, tranNum)
126         return nil, errors.Errorf("no namespace or key is found for
            ↪  namespace %s and key %s with decoded blockNum %d and
            ↪  tranNum %d", scanner.namespace, scanner.key, blockNum,
            ↪  tranNum)
127     }
128     logger.Debugf("Found historic key value for namespace:%s key:%s
        ↪  from transaction %s", scanner.namespace, scanner.key,
        ↪  queryResult.(*queryresult.KeyModification).TxId)
```

```
129     return queryResult, nil
130 }
131
132 func (scanner *versionScanner) Close() {
133     scanner.dbItr.Release()
134 }
135
136 func getKeyModificationFromTran(tranEnvelope *common.Envelope,
    ↪  namespace string, key string) (commonledger.QueryResult, error)
    ↪  {
137     logger.Debugf("Entering getKeyModificationFromTran %s:%s",
        ↪  namespace, key)
138
139     // extract action from the envelope
140     payload, err :=
        ↪  protoutil.UnmarshalPayload(tranEnvelope.Payload)
141     if err != nil {
142         return nil, err
143     }
144
145     tx, err := protoutil.UnmarshalTransaction(payload.Data)
146     if err != nil {
147         return nil, err
148     }
149
150     _, respPayload, err := protoutil.GetPayloads(tx.Actions[0])
151     if err != nil {
152         return nil, err
```

```
153     }
154
155     chdr, err := protoutil.UnmarshalChannelHeader(payload.Header)
156     if err != nil {
157         return nil, err
158     }
159
160     txID := chdr.TxId
161     timestamp := chdr.Timestamp
162
163     txRWSet := &rwsetutil.TxRwSet{}
164
165     // Get the Result from the Action and then Unmarshal
166     // it into a TxReadWriteSet using custom unmarshalling
167     if err = txRWSet.FromProtoBytes(respPayload.Results); err !=
        ↪  nil {
168         return nil, err
169     }
170
171     // look for the namespace and key by looping through the
        ↪  transaction's ReadWriteSets
172     for _, nsRWSet := range txRWSet.NsRwSets {
173         if nsRWSet.NameSpace == namespace {
174             // got the correct namespace, now find the key write
175                 for _, kvWrite := range nsRWSet.KvRwSet.Writes {
176                     if kvWrite.Key == key {
```

```
                        return &queryresult.KeyModification{TxId:
                        ↪  txID, Value: kvWrite.Value, Timestamp:
                        ↪  timestamp, IsDelete:
                        ↪  rwsetutil.IsKVWriteDelete(kvWrite)},
                        ↪  nil
                        }
                } // end keys loop
                logger.Debugf("key [%s] not found in namespace
                ↪  [%s]'s writeset", key, namespace)
                return nil, nil
        } // end if
    } //end namespaces loop
    logger.Debugf("namespace [%s] not found in transaction's
    ↪  ReadWriteSets", namespace)
    return nil, nil
}
```

B.3 GetHistoryForBlockRange Implementation

```
package history

import "sort"

func (q *QueryExecutor) GetHistoryForBlockRange(namespace string,
↪  start uint64, end uint64, updates uint64)
↪  (commonledger.ResultsIterator, error) {
    if end < start {
```

```
7           return nil, errors.Errorf("Start: %d is not less than or
            ↪  equal to end: %d", start, end)
8       }
9
10      if end <= 0 || start <= 0 {
11          return nil, errors.Errorf("Start: %d, end: %d cannot be
            ↪  less than 1", start, end)
12      }
13
14      scanner := &blockRangeScanner{namespace, q.levelDB,
        ↪  q.blockStore, start, end, nil, -1, nil, 0, nil, 0}
15
16      err := scanner.countKeyUpdates(updates)
17      if err != nil {
18          return nil, err
19      }
20
21      _, err = scanner.nextKey()
22      if err != nil {
23          return nil, err
24      }
25
26      logger.Debugf("Initialized block scanner over range from start:
        ↪  %d to end: %d seeking results with at least %d updates",
        ↪  start, end, updates)
27
28      return scanner, nil
29  }
```

```
30
31 // blockRangeScanner implements ResultsIterator for iterating
   ↪ through history results
32 type blockRangeScanner struct {
33     namespace     string
34     levelDB       *leveldbhelper.DBHandle
35     blockStore    *blkstorage.BlockStore
36     start         uint64
37     end           uint64
38     keys          []string
39     keyIndex      int
40     currentKeyItr iterator.Iterator
41     blockNum      uint64
42     transactions  []uint64
43     txIndex       int
44 }
45
46 func (scanner *blockRangeScanner) Next() (commonledger.QueryResult,
   ↪ error) {
47     if scanner.txIndex >= len(scanner.transactions) {
48         keyExhausted, err := scanner.updateCurrentKeyData()
49         if err != nil {
50            return nil, err
51         }
52         if keyExhausted {
53            hasNext, err := scanner.nextKey()
54            if err != nil {
55                return nil, err
```

```
56              }
57              if !hasNext {
58                  logger.Debugf("Query completed")
59                      // All iterators exhausted
60                      return nil, nil
61              }
62          }
63      }
64      key := scanner.keys[scanner.keyIndex]
65
66      blockNum := scanner.blockNum
67      tranNum := scanner.transactions[scanner.txIndex]
68      scanner.txIndex++
69
70      logger.Debugf("Found history record for namespace:%s key:%s at
        ↪  blockNumTranNum %v:%v\n", scanner.namespace, key, blockNum,
        ↪  tranNum)
71
72      // Get the transaction from block storage that is associated
        ↪  with this history record
73      tranEnvelope, err :=
        ↪  scanner.blockStore.RetrieveTxByBlockNumTranNum(blockNum,
        ↪  tranNum)
74      if err != nil {
75          return nil, err
76      }
77
```

```
78      // Get the txid, key write value, timestamp, and delete
        ↪  indicator associated with this transaction
79      queryResult, err := getKeyModificationFromTran(tranEnvelope,
        ↪  scanner.namespace, key)
80      if err != nil {
81          return nil, err
82      }
83      if queryResult == nil {
84          // should not happen, but make sure there is inconsistency
            ↪  between historydb and statedb
85          logger.Errorf("No namespace or key is found for namespace
            ↪  %s and key %s with decoded blockNum %d and tranNum %d",
            ↪  scanner.namespace, key, blockNum, tranNum)
86          return nil, errors.Errorf("no namespace or key is found for
            ↪  namespace %s and key %s with decoded blockNum %d and
            ↪  tranNum %d", scanner.namespace, key, blockNum, tranNum)
87      }
88      logger.Debugf("Found historic key value for namespace:%s key:%s
        ↪  from transaction %s", scanner.namespace, key,
        ↪  queryResult.(*queryresult.KeyModification).TxId)
89      return queryResult, nil
90  }
91
92  func (scanner *blockRangeScanner) Close() {
93      if scanner.currentKeyItr != nil {
94          scanner.currentKeyItr.Release()
95      }
96  }
```

```
97
98  func (scanner *blockRangeScanner) countKeyUpdates(updates uint64)
    ↪  error {
99      keyCounts := make(map[string]int)
100
101     bcInfo, err := scanner.blockStore.GetBlockchainInfo()
102     if err != nil {
103         return err
104     }
105     lastBlockNum := bcInfo.Height - 1
106
107     endBlock := scanner.end
108     if scanner.end > lastBlockNum {
109         endBlock = lastBlockNum
110     }
111
112     for i := scanner.start; i <= endBlock; i++ {
113         nextBlockBytes, err :=
            ↪  scanner.blockStore.RetrieveBlockByNumber(i)
114         if err != nil {
115            return err
116         }
117         for _, txEnvelopeBytes := range nextBlockBytes.Data.Data {
118            tranEnvelope, err :=
               ↪  protoutil.GetEnvelopeFromBlock(txEnvelopeBytes)
119            if err != nil {
120                return err
121            }
```

```
122             err = countKeyUpdatesForTran(tranEnvelope,
                ↪  scanner.namespace, keyCounts)
123             if err != nil {
124                 return err
125             }
126         }
127     }
128     for key, count := range keyCounts {
129         logger.Debugf("Key: %s updated %d times\n", key, count)
130         if count >= int(updates) {
131             scanner.keys = append(scanner.keys, key)
132         }
133     }
134     sort.Strings(scanner.keys)
135     logger.Debugf("%d keys found meeting the update threshold in
        ↪  block range", len(scanner.keys))
136     return nil
137 }
138
139 func (scanner *blockRangeScanner) nextKey() (bool, error) {
140     logger.Debugf("Entering nextKey")
141     if scanner.currentKeyItr != nil {
142         scanner.currentKeyItr.Release()
143     }
144     scanner.keyIndex++
145     if scanner.keyIndex >= len(scanner.keys) {
146         return false, nil
147     }
```

```
148     key := scanner.keys[scanner.keyIndex]
149     rangeScan := constructRangeScan(scanner.namespace, key)
150     var err error
151     scanner.currentKeyItr, err =
        ↪  scanner.levelDB.GetIterator(rangeScan.startKey,
        ↪  rangeScan.endKey)
152     if err != nil {
153         return false, err
154     }
155     for scanner.currentKeyItr.Next() {
156         indexVal := scanner.currentKeyItr.Value()
157         blockNum, transactions, err := decodeLocalIndex(indexVal)
158         if err != nil {
159            return false, err
160         }
161         if blockNum >= scanner.start {
162            scanner.blockNum = blockNum
163            scanner.transactions = transactions
164            scanner.txIndex = 0
165            return true, nil
166         }
167     }
168     // Shouldn't ever reach this line as we precheck that all keys
        ↪  are within block range
169     return false, nil
170 }
171
```

```
172 func (scanner *blockRangeScanner) updateCurrentKeyData() (bool,
    ↪  error) {
173     logger.Debugf("Entering updateCurrentKeyData")
174     if scanner.currentKeyItr == nil ||
        ↪  !scanner.currentKeyItr.Next() {
175         return true, nil
176     }
177     indexVal := scanner.currentKeyItr.Value()
178     blockNum, transactions, err := decodeLocalIndex(indexVal)
179     if err != nil {
180         return true, err
181     }
182     if blockNum > scanner.end {
183         return true, nil
184     }
185     scanner.blockNum = blockNum
186     scanner.transactions = transactions
187     scanner.txIndex = 0
188     return false, nil
189 }
190
191 func countKeyUpdatesForTran(tranEnvelope *common.Envelope,
    ↪  namespace string, keys map[string]int) error {
192     logger.Debugf("Entering getKeysFromTran %s", namespace)
193
194     // extract action from the envelope
195     payload, err :=
        ↪  protoutil.UnmarshalPayload(tranEnvelope.Payload)
```

```
196     if err != nil {
197         return err
198     }
199 
200     tx, err := protoutil.UnmarshalTransaction(payload.Data)
201     if err != nil {
202         return err
203     }
204 
205     _, respPayload, err := protoutil.GetPayloads(tx.Actions[0])
206     if err != nil {
207         return err
208     }
209 
210     txRWSet := &rwsetutil.TxRwSet{}
211 
212     // Get the Result from the Action and then Unmarshal
213     // it into a TxReadWriteSet using custom unmarshalling
214     if err = txRWSet.FromProtoBytes(respPayload.Results); err !=
        ↪  nil {
215         return err
216     }
217 
218     // Read the keys by looping through the transaction's
        ↪  ReadWriteSets
219     for _, nsRWSet := range txRWSet.NsRwSets {
220         if nsRWSet.NameSpace == namespace {
221             // got the correct namespace, now find the key write
```

```
222             for _, kvWrite := range nsRWSet.KvRwSet.Writes {
223                 keys[kvWrite.Key] += 1
224             } // end keys loop
225             return nil
226         } // end if
227     } //end namespaces loop
228     logger.Debugf("namespace [%s] not found in transaction's
        ↪ ReadWriteSets", namespace)
229     return nil
230 }
```

B.4 GetHistoryForKeyRange Implementation

```
1  package history
2
3  func (q *QueryExecutor) GetHistoryForKeyRange(namespace string,
   ↪ keys []string) (commonledger.ResultsIterator, error) {
4      scanners := make(map[string]*historyScanner)
5      for _, key := range keys {
6          scanner, err := q.GetHistoryForKey(namespace, key)
7          if err != nil {
8              return nil, err
9          }
10         var ok bool
11         scanners[key], ok = scanner.(*historyScanner)
12         if !ok {
13             return nil, errors.Errorf("Error converting
               ↪ commonledger.ResultsIterator to historyScanner")
```

```
14          }
15      }
16      scanner := &multipleHistoryScanner{namespace, keys, scanners,
        ↪  0}
17      return scanner, nil
18  }
19
20  // historyScanner implements ResultsIterator for iterating through
    ↪  history results
21  type multipleHistoryScanner struct {
22      namespace       string
23      keys            []string
24      scanners        map[string]*historyScanner
25      currentKeyIndex int
26  }
27
28  func (scanner *multipleHistoryScanner) Next()
    ↪  (commonledger.QueryResult, error) {
29      key := scanner.keys[scanner.currentKeyIndex]
30
31      queryResult, err := scanner.scanners[key].Next()
32      if err != nil {
33          return nil, err
34      }
35
36      for queryResult == nil {
37          scanner.currentKeyIndex++
38          if scanner.currentKeyIndex >= len(scanner.keys) {
```

```
39              return nil, nil
40          }
41
42          key := scanner.keys[scanner.currentKeyIndex]
43
44          queryResult, err = scanner.scanners[key].Next()
45          if err != nil {
46              return nil, err
47          }
48      }
49
50      logger.Debugf("Found historic key value for namespace:%s key:%s
        ↪ from transaction %s", scanner.namespace, key,
        ↪ queryResult.(*queryresult.KeyModification).TxId)
51
52      return queryResult, nil
53  }
54
55  func (scanner *multipleHistoryScanner) Close() {
56      for _, key := range scanner.keys {
57          scanner.scanners[key].dbItr.Release()
58      }
59  }
```

Appendix C

Block-Based Index Implementation

C.1 Block-Based Index Construction

```
package history

func (d *DB) Commit(block *common.Block) error {
    blockNo := block.Header.Number
    //Set the starting tranNo to 0
    var tranNo uint64

    dbBatch := d.levelDB.NewUpdateBatch()
    dataKeys := make(map[string]localIndex)

    logger.Debugf("Channel [%s]: Updating history database for
     ↪ blockNo [%v] with [%d] transactions", d.name, blockNo,
     ↪ len(block.Data.Data))

    // Get the invalidation byte array for the block
    txsFilter := txflags.ValidationFlags(block.Metadata)

    // write each tran's write set to history db
    for _, envBytes := range block.Data.Data {
        // If the tran is marked as invalid, skip it
        if txsFilter.IsInvalid(int(tranNo)) {
        logger.Debugf("Channel [%s]: Skipping history write for
         ↪ invalid transaction number %d", d.name, tranNo)
```

```
21          tranNo++
22          continue
23          }
24
25          env, err := protoutil.GetEnvelopeFromBlock(envBytes)
26          if err != nil {
27             return err
28          }
29
30          payload, err := protoutil.UnmarshalPayload(env.Payload)
31          if err != nil {
32             return err
33          }
34
35          chdr, err := protoutil.UnmarshalChannelHeader(payload)
36          if err != nil {
37             return err
38          }
39
40          if common.HeaderType(chdr.Type) ==
            ↪  common.HeaderType_ENDORSER_TRANSACTION {
41             // extract RWSet from transaction
42             respPayload, err :=
               ↪  protoutil.GetActionFromEnvelope(envBytes)
43             if err != nil {
44                 return err
45             }
46             txRWSet := &rwsetutil.TxRwSet{}
```

```
47              if err = txRWSet.FromProtoBytes(respPayload.Results);
                ↪  err != nil {
48                  return err
49              }
50              // add a history record for each write
51              for _, nsRWSet := range txRWSet.NsRwSets {
52                  ns := nsRWSet.NameSpace
53
54                      for _, kvWrite := range nsRWSet.KvRwSet.Writes {
55                         var (
56                                  prev         uint64
57                                  numVersions  uint64
58                                  transactions []uint64
59                                  )
60                          GIkey := constructGlobalIndexKey(ns,
                            ↪  kvWrite.Key)
61                          localIndexVal, present :=
                            ↪  dataKeys[kvWrite.Key]
62                          if present {
63                               prev, numVersions, transactions, err =
                                 ↪  decodeLocalIndex(localIndexVal)
64                          } else {
65                               // Get returns nil if key not found
66                               globalIndexBytes, err :=
                                 ↪  d.levelDB.Get(GIkey)
67                               if err != nil {
68                                   return err
69                               }
```

```
70                          if globalIndexBytes != nil {
71                              prev, numVersions, err =
                                ↪  decode(globalIndexBytes)
72                              if err != nil {
73                                  return err
74                              }
75                          } else {
76                              prev = blockNo
77                              // numVersions is initialized to
                                ↪  0
78                          }
79                      }
80                      if err != nil {
81                          return err
82                      }
83
84                      transactions = append(transactions, tranNo)
85                      numVersions++
86
87                      indexVal := constructLocalIndex(prev,
                        ↪  numVersions, transactions)
88                      dataKeys[kvWrite.Key] = indexVal
89
90                      updatedGlobalIndexBytes :=
                        ↪  constructGlobalIndexVal(blockNo,
                        ↪  numVersions)
91                      err = d.levelDB.Put(GIkey,
                        ↪  updatedGlobalIndexBytes, true)
```

```
92                          if err != nil {
93                              return err
94                          }
95                  dataKey := constructDataKey(ns, blockNo,
                    ↪  kvWrite.Key)
96                          dbBatch.Put(dataKey, indexVal)
97                  }
98          }
99
100         } else {
101             logger.Debugf("Skipping transaction [%d] since it is
                ↪  not an endorsement transaction\n", tranNo)
102         }
103         tranNo++
104     }
105     // add savepoint for recovery purpose
106     height := version.NewHeight(blockNo, tranNo)
107     dbBatch.Put(savePointKey, height.ToBytes())
108
109     // write the block's history records and savepoint to LevelDB
110     // Setting snyc to true as a precaution, false may be an ok
        ↪  optimization after further testing.
111     if err := d.levelDB.WriteBatch(dbBatch, true); err != nil {
112         return err
113     }
114     logger.Debugf("Channel [%s]: Updates committed to history
        ↪  database for blockNo [%v]", d.name, blockNo)
```

```
115     return nil
116 }
```

C.2 GetHistoryForVersionRange Implementation

```
1  package history
2
3  func (q *QueryExecutor) GetHistoryForVersionRange(namespace string,
   ↪  key string, start uint64, end uint64)
   ↪  (commonledger.ResultsIterator, error) {
4      if end < start {
5          return nil, errors.Errorf("Start: %d is not less than or
           ↪  equal to end: %d", start, end)
6      }
7
8      if end <= 0 || start <= 0 {
9          return nil, errors.Errorf("Start: %d, end: %d cannot be
           ↪  less than 1", start, end)
10     }
11
12     dbItr, err := q.levelDB.GetIterator(nil, nil)
13     if err != nil {
14         return nil, err
15     }
16
17     GIkey := constructGlobalIndexKey(namespace, key)
18     globalIndexBytes, err := q.levelDB.Get(GIkey)
19     if err != nil {
```

```
20          return nil, errors.Errorf("Error reading from history
            ↪  database for key: %s", key)
21      }
22      if globalIndexBytes == nil {
23          logger.Debugf("Key not present in GI. Initialized nil
            ↪  version scanner for key %s.", key)
24          // This scanner will return nil upon first call to Next()
25          return &versionScanner{namespace, key, dbItr, q.blockStore,
            ↪  start, nil, -1, start, end}, nil
26      }
27      blockNum, _, err := decodeGlobalIndex(globalIndexBytes)
28      if err != nil {
29          return nil, errors.Errorf("Error decoding lasts known
            ↪  version for key: %s", key)
30      }
31
32      dataKey := constructDataKey(namespace, blockNum, key)
33      found := dbItr.Seek(dataKey)
34      if !found {
35          return nil, errors.Errorf("Error from dbItr.Seek() for key:
            ↪  %s, block: %d", key, blockNum)
36      }
37
38      indexVal := dbItr.Value()
39      prev, _, transactions, err := decodeLocalIndex(indexVal)
40      if err != nil {
41          return nil, err
42      }
```

```
43
44      txIndex := len(transactions) - 1
45
46      logger.Debugf("Initialized version scanner for key %s,
        ↪  currentBlock: %d, previousBlock: %d", key, blockNum, prev)
47
48      scanner := &versionScanner{namespace, key, dbItr, q.blockStore,
        ↪  blockNum, indexVal, txIndex, start, end}
49      // Find first block containing end version in range
50      for {
51          _, numVersions, transactions, err :=
            ↪  decodeLocalIndex(scanner.indexVal)
52          if err != nil {
53              return nil, err
54          }
55          firstVersionInBlock := numVersions -
            ↪  uint64(len(transactions)) + 1
56          if firstVersionInBlock <= scanner.end {
57              if numVersions >= scanner.end {
58                  scanner.txIndex = int(scanner.end -
                    ↪  firstVersionInBlock)
59              } else {
60                  scanner.txIndex = len(transactions) - 1
61              }
62              return scanner, nil
63          }
64
65          oldBlockNum := scanner.currentBlock
```

```
66          err = scanner.updateBlock()
67          if err != nil {
68              return nil, err
69          }
70          if scanner.currentBlock == oldBlockNum {
71              // Iterator exhausted
72              scanner.txIndex = -1
73              return scanner, nil
74          }
75      }
76  }
77
78  type versionScanner struct {
79      namespace     string
80      key           string
81      dbItr         iterator.Iterator
82      blockStore    *blkstorage.BlockStore
83      currentBlock uint64
84      indexVal      localIndex
85      txIndex       int
86      start         uint64
87      end           uint64
88  }
89
90  func (scanner *versionScanner) Next() (commonledger.QueryResult,
    ↪  error) {
91      if scanner.indexVal == nil {
92          return nil, nil
```

```
93      }
94      _, numVersions, transactions, err :=
        ↪  decodeLocalIndex(scanner.indexVal)
95      if err != nil {
96          return nil, err
97      }
98      firstVersionInBlock := numVersions - uint64(len(transactions))
        ↪  + 1
99      currentVersionNum := firstVersionInBlock +
        ↪  uint64(scanner.txIndex)
100     if currentVersionNum < scanner.start {
101         logger.Debugf("First requested version %d found for key:
            ↪  %s", scanner.start, scanner.key)
102         return nil, nil
103     }
104     if scanner.txIndex == -1 {
105         oldBlockNum := scanner.currentBlock
106         err := scanner.updateBlock()
107         if err != nil {
108            return nil, err
109         }
110         if scanner.currentBlock == oldBlockNum {
111            // Iterator exhausted
112            return nil, nil
113         }
114         _, _, transactions, err =
            ↪  decodeLocalIndex(scanner.indexVal)
115         if err != nil {
```

```
116             return nil, err
117         }
118     }
119
120     blockNum := scanner.currentBlock
121     tranNum := transactions[scanner.txIndex]
122     scanner.txIndex--
123
124     logger.Debugf("Found history record for namespace:%s key:%s at
        ↪  blockNumTranNum %v:%v\n", scanner.namespace, scanner.key,
        ↪  blockNum, tranNum)
125
126     // Get the transaction from block storage that is associated
        ↪  with this history record
127     tranEnvelope, err :=
        ↪  scanner.blockStore.RetrieveTxByBlockNumTranNum(blockNum,
        ↪  tranNum)
128     if err != nil {
129         return nil, err
130     }
131
132     // Get the txid, key write value, timestamp, and delete
        ↪  indicator associated with this transaction
133     queryResult, err := getKeyModificationFromTran(tranEnvelope,
        ↪  scanner.namespace, scanner.key)
134     if err != nil {
135         return nil, err
136     }
```

```
137     if queryResult == nil {
138         // should not happen, but make sure there is inconsistency
            ↪ between historydb and statedb
139         logger.Errorf("No namespace or key is found for namespace
            ↪ %s and key %s with decoded blockNum %d and tranNum %d",
            ↪ scanner.namespace, scanner.key, blockNum, tranNum)
140         return nil, errors.Errorf("no namespace or key is found for
            ↪ namespace %s and key %s with decoded blockNum %d and
            ↪ tranNum %d", scanner.namespace, scanner.key, blockNum,
            ↪ tranNum)
141     }
142     logger.Debugf("Found key version %d for namespace:%s key:%s
        ↪ from transaction %s", currentVersionNum, scanner.namespace,
        ↪ scanner.key,
        ↪ queryResult.(*queryresult.KeyModification).TxId)
143     return queryResult, nil
144 }
145
146 func (scanner *versionScanner) Close() {
147     scanner.dbItr.Release()
148 }
149
150 func (scanner *versionScanner) updateBlock() error {
151     prev, _, _, err := decodeLocalIndex(scanner.indexVal)
152     if err != nil {
153         return err
154     }
```

```
155     logger.Debugf("Updating block for key %s. currentBlock %d,
        ↪ previousBlock %d", scanner.key, scanner.currentBlock, prev)
156     scanner.currentBlock = prev
157     dataKey := constructDataKey(scanner.namespace, prev,
        ↪ scanner.key)
158     found := scanner.dbItr.Seek(dataKey)
159     if !found {
160         return errors.Errorf("Error from dbItr.Seek() for key: %s,
            ↪ block: %d", scanner.key, prev)
161     }
162     scanner.indexVal = scanner.dbItr.Value()
163     _, _, transactions, err := decodeLocalIndex(scanner.indexVal)
164     if err != nil {
165         return err
166     }
167     scanner.txIndex = len(transactions) - 1
168     logger.Debugf("Fished updating block for key %s. currentBlock
        ↪ %d, previousBlock %d", scanner.key, scanner.currentBlock,
        ↪ prev)
169     return nil
170 }
171
172 func getKeyModificationFromTran(tranEnvelope *common.Envelope,
    ↪ namespace string, key string) (commonledger.QueryResult, error)
    ↪ {
173     logger.Debugf("Entering getKeyModificationFromTran %s:%s",
        ↪ namespace, key)
174     // extract action from the envelope
```

```
175     payload, err :=
        ↪  protoutil.UnmarshalPayload(tranEnvelope.Payload)
176     if err != nil {
177         return nil, err
178     }
179     tx, err := protoutil.UnmarshalTransaction(payload.Data)
180     if err != nil {
181         return nil, err
182     }
183
184     _, respPayload, err := protoutil.GetPayloads(tx.Actions[0])
185     if err != nil {
186         return nil, err
187     }
188
189     chdr, err := protoutil.UnmarshalChannelHeader(payload.Header)
190     if err != nil {
191         return nil, err
192     }
193
194     txID := chdr.TxId
195     timestamp := chdr.Timestamp
196
197     txRWSet := &rwsetutil.TxRwSet{}
198
199     // Get the Result from the Action and then Unmarshal
200     // it into a TxReadWriteSet using custom unmarshalling
```

```
201     if err = txRWSet.FromProtoBytes(respPayload.Results); err !=
        ↪  nil {
202         return nil, err
203     }
204
205     // look for the namespace and key by looping through the
        ↪  transaction's ReadWriteSets
206     for _, nsRWSet := range txRWSet.NsRwSets {
207         if nsRWSet.NameSpace == namespace {
208             // got the correct namespace, now find the key write
209             for _, kvWrite := range nsRWSet.KvRwSet.Writes {
210                  if kvWrite.Key == key {
211                       return &queryresult.KeyModification{TxId:
                          ↪  txID, Value: kvWrite.Value, Timestamp:
                          ↪  timestamp, IsDelete:
                          ↪  rwsetutil.IsKVWriteDelete(kvWrite)}, nil
212                       }
213             } // end keys loop
214             logger.Debugf("key [%s] not found in namespace [%s]'s
                ↪  writeset", key, namespace)
215             return nil, nil
216         } // end if
217     } //end namespaces loop
218     logger.Debugf("namespace [%s] not found in transaction's
        ↪  ReadWriteSets", namespace)
219     return nil, nil
220 }
```

C.3 GetHistoryForBlockRange Implementation

```go
package history

func (q *QueryExecutor) GetHistoryForBlockRange(namespace string,
    start uint64, end uint64, updates uint64)
    (commonledger.ResultsIterator, error) {
    if end < start {
        return nil, errors.Errorf("Start: %d is not less than or
            equal to end: %d", start, end)
    }

    if end <= 0 || start <= 0 {
        return nil, errors.Errorf("Start: %d, end: %d cannot be
            less than 1", start, end)
    }

    // Create iterator over range from ns~startBlock to ns~endBlock
    startKey := append([]byte(namespace), compositeKeySep...)
    startKey = append(startKey,
        util.EncodeOrderPreservingVarUint64(start)...)

    // End key is exclusive, end+1 results in range including all
        entries from 'start' to 'end'
    endKey := append([]byte(namespace), compositeKeySep...)
    endKey = append(endKey,
        util.EncodeOrderPreservingVarUint64(end+1)...)

    dbItr, err := q.levelDB.GetIterator(startKey, endKey)
```

```
21      if err != nil {
22          return nil, err
23      }
24
25      logger.Debugf("Initialized block scanner over range from start:
        ↪  %d to end: %d seeking results with at least %d updates",
        ↪  start, end, updates)
26
27      keys := make(map[string]bool)
28      scanner := &blockRangeScanner{namespace, dbItr, q.blockStore,
        ↪  keys, start, "", nil, 0}
29
30      err = scanner.countKeyUpdates(updates)
31      if err != nil {
32          return nil, err
33      }
34      return scanner, nil
35  }
36
37  // blockRangeScanner implements ResultsIterator for iterating
    ↪  through history results
38  type blockRangeScanner struct {
39      namespace    string
40      dbItr        iterator.Iterator
41      blockStore   *blkstorage.BlockStore
42      keys         map[string]bool
43      currentBlock uint64
44      currentKey   string
```

```
45      transactions []uint64
46      txIndex      int
47  }
48
49  func (scanner *blockRangeScanner) Next() (commonledger.QueryResult,
    ↪  error) {
50      if scanner.txIndex >= len(scanner.transactions) {
51          hasNext, key, err := scanner.nextKey()
52          if err != nil {
53              return nil, err
54          }
55          if !hasNext {
56              // Iterator exhausted
57              return nil, nil
58          }
59          scanner.currentKey = key
60      }
61
62      blockNum := scanner.currentBlock
63      tranNum := scanner.transactions[scanner.txIndex]
64      scanner.txIndex++
65
66      logger.Debugf("Found history record for namespace:%s key:%s at
        ↪  blockNumTranNum %v:%v\n", scanner.namespace,
        ↪  scanner.currentKey, blockNum, tranNum)
67
68      // Get the transaction from block storage that is associated
        ↪  with this history record
```

```
69      tranEnvelope, err :=
        ↪  scanner.blockStore.RetrieveTxByBlockNumTranNum(blockNum,
        ↪  tranNum)
70      if err != nil {
71          return nil, err
72      }
73
74      // Get the txid, key write value, timestamp, and delete
        ↪  indicator associated with this transaction
75      queryResult, err := getKeyModificationFromTran(tranEnvelope,
        ↪  scanner.namespace, scanner.currentKey)
76      if err != nil {
77          return nil, err
78      }
79      if queryResult == nil {
80          // should not happen, but make sure there is inconsistency
            ↪  between historydb and statedb
81          logger.Errorf("No namespace or key is found for namespace
            ↪  %s and key %s with decoded blockNum %d and tranNum %d",
            ↪  scanner.namespace, scanner.currentKey, blockNum,
            ↪  tranNum)
82          return nil, errors.Errorf("no namespace or key is found for
            ↪  namespace %s and key %s with decoded blockNum %d and
            ↪  tranNum %d", scanner.namespace, scanner.currentKey,
            ↪  blockNum, tranNum)
83      }
84      logger.Debugf("Found historic key value for namespace:%s key:%s
        ↪  from transaction %s",
```

```
85      scanner.namespace, scanner.currentKey,
        ↪  queryResult.(*queryresult.KeyModification).TxId)
86      return queryResult, nil
87  }
88
89  func (scanner *blockRangeScanner) Close() {
90      scanner.dbItr.Release()
91  }
92
93  func (scanner *blockRangeScanner) countKeyUpdates(updates uint64)
    ↪  error {
94      keyCounts := make(map[string]int)
95      for scanner.dbItr.Next() {
96      _, key, err := decodeDataKey(scanner.namespace,
        ↪  scanner.dbItr.Key())
97      if err != nil {
98          return err
99      }
100     _, _, transactions, err :=
        ↪  decodeLocalIndex(scanner.dbItr.Value())
101     if err != nil {
102         return err
103     }
104     keyCounts[key] += len(transactions)
105     if keyCounts[key] >= int(updates) {
106         scanner.keys[key] = true
107         }
108     }
```

```
109     scanner.dbItr.First()
110     scanner.dbItr.Prev()
111     return nil
112 }
113
114 func (scanner *blockRangeScanner) nextKey() (bool, string, error) {
115     for {
116         if !scanner.dbItr.Next() {
117             return false, "", nil
118         }
119         dataKey := scanner.dbItr.Key()
120         blockNum, key, err := decodeDataKey(scanner.namespace,
            ↪  dataKey)
121         if err != nil {
122             return false, "", err
123         }
124         scanner.currentBlock = blockNum
125         if scanner.keys[key] {
126             indexVal := scanner.dbItr.Value()
127             _, _, transactions, err := decodeLocalIndex(indexVal)
128             if err != nil {
129                 return false, "", err
130             }
131             scanner.transactions = transactions
132             scanner.txIndex = 0
133             logger.Debugf("Next key: %s, appears in block %d
                ↪  transactions: %v\n", key, scanner.currentBlock,
                ↪  scanner.transactions)
```

```
134             return true, key, nil
135         }
136     }
137 }
138
139 func getKeyModificationFromTran(tranEnvelope *common.Envelope,
    ↪  namespace string, key string) (commonledger.QueryResult, error)
    ↪  {
140     logger.Debugf("Entering getKeyModificationFromTran %s:%s",
        ↪  namespace, key)
141     // extract action from the envelope
142     payload, err :=
        ↪  protoutil.UnmarshalPayload(tranEnvelope.Payload)
143     if err != nil {
144         return nil, err
145     }
146
147     tx, err := protoutil.UnmarshalTransaction(payload.Data)
148     if err != nil {
149         return nil, err
150     }
151
152     _, respPayload, err := protoutil.GetPayloads(tx.Actions[0])
153     if err != nil {
154         return nil, err
155     }
156
157     chdr, err := protoutil.UnmarshalChannelHeader(payload.Header)
```

```
158     if err != nil {
159         return nil, err
160     }
161
162     txID := chdr.TxId
163     timestamp := chdr.Timestamp
164
165     txRWSet := &rwsetutil.TxRwSet{}
166
167     // Get the Result from the Action and then Unmarshal
168     // it into a TxReadWriteSet using custom unmarshalling
169     if err = txRWSet.FromProtoBytes(respPayload.Results); err !=
        ↪  nil {
170         return nil, err
171     }
172
173     // look for the namespace and key by looping through the
        ↪  transaction's ReadWriteSets
174     for _, nsRWSet := range txRWSet.NsRwSets {
175         if nsRWSet.NameSpace == namespace {
176             // got the correct namespace, now find the key write
177             for _, kvWrite := range nsRWSet.KvRwSet.Writes {
178                 if kvWrite.Key == key {
179                         return &queryresult.KeyModification{TxId:
                            ↪  txID, Value: kvWrite.Value, Timestamp:
                            ↪  timestamp, IsDelete:
                            ↪  rwsetutil.IsKVWriteDelete(kvWrite)}, nil
180                     }
```

```
            } // end keys loop
            logger.Debugf("key [%s] not found in namespace [%s]'s
            ↪  writeset", key, namespace)
            return nil, nil
        } // end if
    } //end namespaces loop
    logger.Debugf("namespace [%s] not found in transaction's
    ↪  ReadWriteSets", namespace)
    return nil, nil
}
```

C.4 GetHistoryForKeyRange Implementation

```
package history

func (q *QueryExecutor) GetHistoryForKeyRange(namespace string,
↪  keys []string) (commonledger.ResultsIterator, error) {
    scanners := make(map[string]*historyScanner)
    for _, key := range keys {
        scanner, err := q.GetHistoryForKey(namespace, key)
        if err != nil {
            return nil, err
        }
        var ok bool
        scanners[key], ok = scanner.(*historyScanner)
        if !ok {
            return nil, errors.Errorf("Error converting
            ↪  commonledger.ResultsIterator to historyScanner")
```

```
        }
    }
    scanner := &multipleHistoryScanner{namespace, keys, scanners,
    ↪  0}
    return scanner, nil
}

// historyScanner implements ResultsIterator for iterating through
↪  history results
type multipleHistoryScanner struct {
    namespace       string
    keys            []string
    scanners        map[string]*historyScanner
    currentKeyIndex int
}

func (scanner *multipleHistoryScanner) Next()
↪  (commonledger.QueryResult, error) {
    key := scanner.keys[scanner.currentKeyIndex]
    queryResult, err := scanner.scanners[key].Next()
    if err != nil {
        return nil, err
    }

    for queryResult == nil {
        scanner.currentKeyIndex++
        if scanner.currentKeyIndex >= len(scanner.keys) {
           return nil, nil
```

```
39          }
40
41          key := scanner.keys[scanner.currentKeyIndex]
42
43          queryResult, err = scanner.scanners[key].Next()
44          if err != nil {
45             return nil, err
46          }
47      }
48      logger.Debugf("Found historic key value for namespace:%s key:%s
        ↪  from transaction %s", scanner.namespace, key,
        ↪  queryResult.(*queryresult.KeyModification).TxId)
49
50      return queryResult, nil
51  }
52
53  func (scanner *multipleHistoryScanner) Close() {
54      for _, key := range scanner.keys {
55          scanner.scanners[key].dbItr.Release()
56      }
57  }
```