UNIVERSITY OF CALIFORNIA,
IRVINE

RollStore: Hybrid Onchain-Offchain Data Indexing for Decentralized Blockchain Applications

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Qi Lin

2023

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my advisor, Professor Faisal Nawab for his advice, careful editing, and patience during the preparation of this thesis.

I would also like to thank my friend, Binbin Gu for his opinion during the preparation of this thesis.

I am grateful to my parents, and my sister, for their continuous support during my graduate studies at UCI.

ABSTRACT OF THE THESIS

RollStore: Hybrid Onchain-Offchain Data Indexing for Decentralized Blockchain Applications

By

Qi Lin

Master of Science in Computer Science

University of California, Irvine, 2023

Assistant Professor Faisal Nawab, Chair

The interest in building blockchain Decentralized Applications (DApps) has been growing over the past few years. DApps are implemented as smart contracts which are programs that are maintained by a blockchain network. Building DApps, however, faces many challenges—most notably the performance and monetary overhead of writing to blockchain smart contracts. To overcome this challenge, many DApp developers have explored utilizing *off-chain* resources—nodes outside of the blockchain network—to offload part of the processing and storage.

In this paper, we propose RollStore, a data indexing solution for hybrid onchain-offchain DApps. RollStore provides efficiency in terms of reduced cost and latency, as well as security in terms of tolerating byzantine (i.e., malicious) offchain nodes. RollStore achieves this by: (1) a three-stage commitment strategy where each stage represents a point in a performance-security trade-off—i.e., first stage is fast but less secure while the last stage is slower but is more secure. (2) utilizing zero-knowledge (zk) proofs to enable the onchain smart contract to verify offchain operations with a small cost. (3) Combining Log-Structured Merge (LSM) trees and Merkle Mountain Range (MMR) trees to efficiently enable both access and verification of indexed data. We experimentally evaluate the cost and performance benefits of RollStore while comparing with BlockchainDB and BigChainDB.

Chapter 1

Introduction

1.1 Motivation

Decentralized Applications (DApps) are applications that are implemented as smart contracts. A smart contract is a program where its state and logic is maintained by a blockchain network[1]. This makes DApps inherit blockchain features such as decentralization, transparency, and tamper-freedom [12]. Recently, there has been a lot of interest in DApps. Various DApps have amassed hundreds of thousands of users and hundreds of millions of dollars in assets [1, 53, 65]. DApps span many areas such as decentralized finance [16, 55], gaming and metaverses [2, 4, 46], and supply-chain [10, 63].

DApp developers face many challenges including the performance overhead and monetary cost of writing to blockchain smart contracts. Writing to a blockchain smart contracts can take tens of minutes or more to finalize [19]. The cost of a smart contract operation depends on the complexity of the operation and amount of storage, but it is estimated that the average cost of a single smart contract operation is around $3 [3].

[1]In this paper, we consider permissionless blockchain technologies such as Ethereum as they are the ones used predominately by DApps [1].

To overcome these overheads, many DApps are developed using the *hybrid onchain-offchain* model [8, 26, 43, 74], that we call the *hybrid model* for short. In this model, part of the DApp logic is maintained in the on-chain contract—where by on-chain we mean that it is implemented in the blockchain smart contract. The rest of the processing and storage of the DApp is maintained by off-chain nodes—where off-chain nodes are compute nodes that are outside of the blockchain network. By handing off part of the processing and storage to off-chain nodes, the monetary cost and performance overhead is reduced significantly.

The introduction of off-chain nodes, however, raises another challenge—these off-chain nodes might be untrusted. Hybrid DApps explore various methods to enable utilizing off-chain nodes while maintaining the integrity of the application. These methods include using trusted off-chain nodes [41], trusted execution environments [20] such as Intel SGX [18], and authenticated and verifiable data structures [51, 71]. These methods, however, remain limited by strong assumptions (i.e., trusted offchain nodes and execution environments) or high computation complexity (i.e., time needed to verify and prove data correctness).

In this work, we aim to build a data indexing solution for DApps in the hybrid onchain-offchain model. Data indexing is a fundamental problem in data management and a building block for more complex data management functionality. Therefore, the development of an efficient and secure data indexing solution for DApps can have an impact on a wide-range of decentralized systems.

Currently, existing blockchain-based databases (BBDBs) fall under one or more of the following categories [9, 22, 41, 47, 52, 56, 20, 62, 26, 51]:

- BBDBs that do not utilize off-chain nodes efficiently (i.e., by writing all data and/or operations on-chain) [9, 22], making them inefficient in terms of monetary cost and performance overhead.
- BBDBs that utilize off-chain nodes that are assumed to be trusted/permissioned (i.e.,

with closed membership) [41] or utilize trusted execution environments [20]. These are strong assumptions for decentralized environments and limit their practicality for DApps which are widely implemented on permissionless environments.

- BBDBs that utilize authenticated data structures and verification methods to verify query results using on-chain digests or verifiers [51, 71]. However, these BBDBs only consider querying immutable data and suffer from the two limitations above if changes need to be applied to data.

Our solution aims to extend and support the space of blockchain-based data management systems.

1.2 Approach

We propose RollStore, a data indexing solution for hybrid blockchain DApps that overcomes the challenges of prior BBDBs. RollStore has the following properties:

- RollStore utilizes off-chain nodes efficiently by not needing to send raw data or operations to the smart contract. Smart contracts are only used for performing low-overhead operations (i.e., lightweight verification of data digests). This makes RollStore efficient in terms of reducing monetary cost of writing to blockchain.
- RollStore does not assume that any off-chain node is trusted. This makes RollStore practical for DApps that rely on permissionless blockchain.
- RollStore has a key-value interface where users can both read and write data. RollStore is the first data indexing solution for hybrid DApps that can achieve all these three properties. We envision that RollStore can be augmented with existing BBDBs as their

indexing component to help transform them to enjoy the aforementioned RollStore properties.

RollStore can achieve the three properties above by bringing together and innovating in the areas of zero-knowledge (zk) proofs [67], optimistic and zk rollups [57, 61], Log-Structured Merge (LSM) trees [39], and Merkle Mountain Range (MMR) trees [58].

We utilize Zero-Knowledge Succinct Non-interactive Argument of Knowledge (zk-SNARK) [13, 25]. Zk-SNARK allows an untrusted node to perform a computation that changes the state of the data and produce the new state with a proof of the new state's correctness (i.e., that the new state is the result of applying a correct mutating operation on the previous state). The proof can be verified with low overhead, thus allowing cheap verification on-chain.

We also utilize the concept of optimistic rollups (o-rollups) [57]. O-rollups were proposed as a layer-2 scaling solution for blockchain, where off-chain nodes perform compute functions on behalf of the layer-1 blockchain. Then, clients can interact with the blockchain in a challenge period to challenge the correctness of the off-chain outcomes.

Finally, we integrate LSM and MMR tree structures in the design. LSM's append-only nature makes it a good candidate to manage the movement of data from one stage to another (e.g., data in each LSM layer corresponds to a different processing/validation stage). MMR trees allow deconstructing a single MMR tree into smaller merkle trees. This allows better modularity and integration with LSM trees.

RollStore combines the aforementioned technologies in a new design for hybrid DApp data indexing. RollStore consists of three types of nodes: (1) an *updater* node that manages clients requests, (2) a *prover* node that is responsible for generating proofs for operations, and (3) a *backup* node that maintains the verified data and associated proofs. In addition, RollStore includes a smart contract that performs lightweight verification of digests and proofs.

RollStore also proposes a three-stage commitment process to manage the performance-security trade-off of verification methods. In particular, we support three kinds of verification methods for updates to off-chain data:

- Off-chain response proof: this is the weakest guarantee which is for the client to receive a signed response from the off-chain node before any digest or proof is written on-chain. The signed response represents a promise to include and process the request in a specific LSM page. The client can use the signed response to punish the off-chain node in the case it lies or acts maliciously—by not honoring its promise to process the request. (We discuss the punishment smart contract in the paper which withdraws a penalty amount from an escrow fund of the off-chain node if proven to have acted maliciously).
- Optimistic rollups (o-rollups): this method relies on a simple data digest that is written on-chain. Users agree on the digest and the data represented by that digest; however, it is not guaranteed that the corresponding operations and new state are processed correctly.
- Zk-SNARK proofs: this is the most trusted verification as it verifies the correctness of the data operation and new off-chain data state. However, it has high computational complexity requiring a long time to generate proofs.

RollStore three-stage commitment takes an operation through three stages starting from the faster verification method to the slower ones:

- The transaction is stage 0 committed, meaning that it has a signed response from the off-chain node. This signed response can be used later to punish the off-chain node if it has lied. Depending on the operation and punishment, this is designed to be a deterrent of malicious off-chain nodes.

- Once the initial digest of the updated data is written on-chain, it is considered stage 1 committed. Users agree on this digest and the data it represents at this stage because the digest is immutable on-chain. The state corresponding to the digest might still be incorrect—since no verification of how the digest was generated. Punishment strategies can be applied in this case as well if the off-chain node is proved to have acted maliciously.
- After the zk-SNARKS proof is produced and written on-chain, users have complete trust on the correctness of off-chain data. This is because the zk proof proves that the corresponding data is computed correctly.

The contribution of the paper is summarized as the following:

- RollStore is the first *dynamic indexing solution for hybrid DApps and BBDBs that enables efficient utilization of untrusted off-chain nodes* (i.e., without requiring on-chain operations other than simple verification of digests and zk proofs).
- RollStore is the first blockchain-based data management solution that utilizes zk proofs and o-rollups in a three-stage design that manages the performance-security trade-off of these methods.
- RollStore is the first solution to incorporate zk proofs and o-rollups verification in the problem of indexing by a novel design that builds on LSM and MMR trees.

1.3 Outline of the Thesis

In the rest of the thesis, we present the background in Chapter 2. Then, we present the design of RollStore in Chapter 3. An experimental evaluation is presented in Chapter 4. We discuss related work in Chapter 5 and conclude in Chapter 6.

Chapter 2

Background

2.1 LSM Trees

Log-Structured Merge (LSM) Trees are widely used for data indexing [39]. LSM trees are designed to support fast data ingestion by appending entries for write operations instead of updating the corresponding old entry in-place. Periodically, appended data is merged with the rest of the LSM tree. This append-only nature of ingestion makes LSM trees a suitable candidate for write-intensive workloads.

There are many LSM tree variants [39]. Here, we provide a description of the common and typical design aspects of LSM trees. Generally, LSM trees contain several levels, L_0, L_1, $\cdots$, L_k. Level L_0 is maintained in main memory while other levels are persisted on disk. Incoming write operations are appended to an in-memory mutable table. When the mutable table is full, the data in it—represented as key-value pairs—is ordered and inserted to L_0 as a new page. L_0—as well as other levels—has a threshold on the number of pages. Once this threshold is met, the data in L_0 pages is merged with the pages in the next level. This continues until data reaches the final level L_k. When two levels are merged, one of two widely used techniques is used: tiering and leveling [15]. The tiering strategy merges all pages of the two levels and inserts the new (merged) pages in the higher level. Leveling, on the other hand, only merges a subset of the pages in the lower level with the pages in the higher level.

2.2 MMR Trees

The Merkle Mountain Range (MMR) tree [58] is a variant of the merkle tree [42] which is structured as a group of underlying merkle trees (Figure 2.1). A MMR tree is an append-only tree where elements are added as leaf nodes from left to right (Figure 2.1 shows the case of adding items 1 to 7 from left to right.) Once there are two children nodes at a level with no parent node, a parent node for the two children nodes is generated at the higher level. For example, consider the MMR tree in Figure 2.1 where internal node numbers represent the generation order (e.g., $Hash_i$ is the i^{th} generated hash node). $Hash_6$, for example, is generated after $Hash_4$ and $Hash_5$ are added (for items 3 and 4). Adding $Hash_6$ in turn leads to creating $Hash_7$. In the figure, there are three underlying merkle trees with roots $Hash_7$, $Hash_{10}$ and $Hash_{11}$. These roots are also called peak nodes, and each underlying merkle tree is called a *mountain*. The MMR root is calculated as the hash of the peak nodes.

An MMR tree can provide an inclusion proof of a data item in a similar way to merkle trees. The inclusion proof includes the sibling node of every node in the path from the data item to the MMR root. For example, in Figure 2.1, the inclusion proof of item $Item_3$ contains $Hash_5$, $Hash_3$, $Hash_{10}$, and $Hash_{11}$. A client receiving the proof calculates the MMR root using the provided hashes. If the calculated MMR root matches the original MMR root, then the client knows that the received item is correct. A malicious server cannot generate a false inclusion proof of an item that is not in the MMR tree. This is because any change in leaf nodes leads to changing the MMR root.

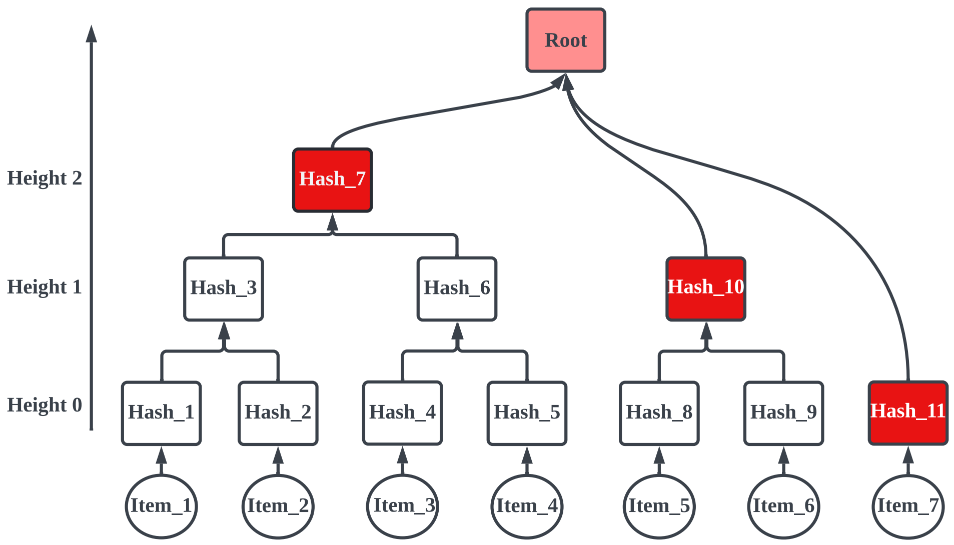

Figure 2.1: An example of the Merkle Mountain Range tree.

2.3 Blockchain Rollups

Rollups is a layer 2 solution to enhance blockchain scalability, which aims to reduce the performance overhead and monetary cost of operations on-chain [57]. In rollups, transactions are aggregated and executed in off-chain nodes, then the on-chain smart contract maintains the root value (e.g., merkle root hash value) which corresponds to the current state (Figure 2.2). The off-chain node publishes a digest for batched transactions, which contains the previous merkle root MMR_{Pre} (0x123456 in the figure) and the computed new merkle root MMR_{New} (0x456789 in the figure). When such digest is written on-chain, the smart contract checks whether the previous merkle root MMR_{Pre} in this digest matches the current merkle root stored in the smart contract; if it does, the smart contract updates its state root to the new MMR root, MMR_{New}. Such a mechanism avoids writing all transaction information on-chain, thus reducing the overhead and transaction fee.

The main challenge with rollups solutions is that the off-chain node might act maliciously and provide a new digest, MMR_{New}, that corresponds to an incorrect new state, i.e., the transactions that leads to the new state with digest MMR_{New} are incorrect or malicious

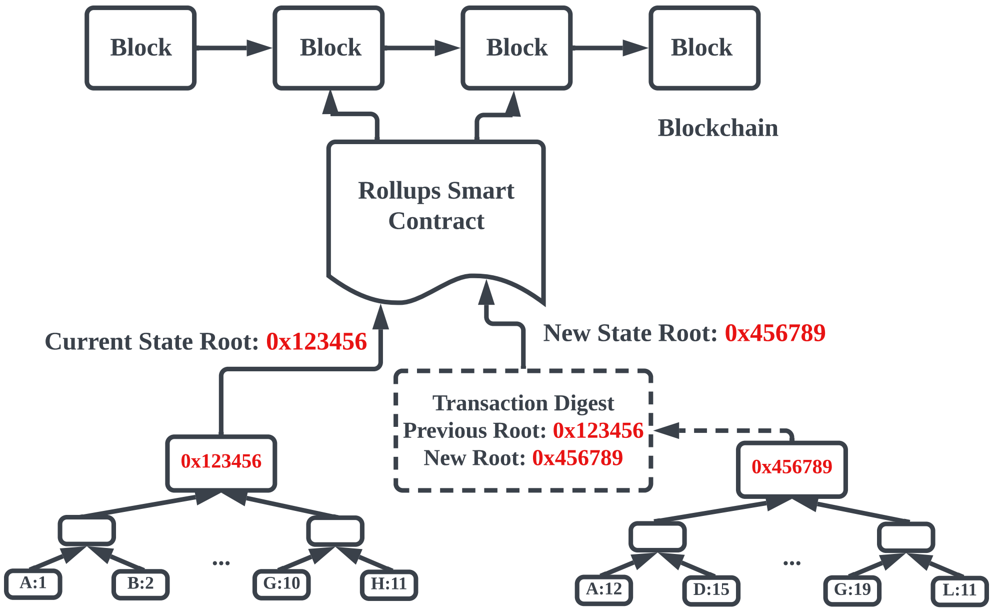

Figure 2.2: An example of blockchain rollups.

transactions. To overcome this challenge, two types of rollups variants are used: optimistic rollups (o-rollups) and zero-knowledge (zk) rollups.

2.3.1 Optimistic Rollups

O-rollups ensures that the new hash that is written on-chain, MMR_{New}, is based on correct computation by using an *interactive fraud-proof mechanism* [5]. In this approach, the new digest is written on-chain before verification (optimistically). Then, off-chain nodes and clients have an opportunity to challenge the correctness of the state that is represented by the new digest, MMR_{New}. This opportunity remains for a pre-defined *challenge period*. After this period expires, if no successful challenges are raised, then the new digest is assumed to be correct. Otherwise, if a client challenges the correctness of MMR_{New}, then a special smart contract verifies the correctness of the challenge. If the new state turns out to be incorrect, the challenge succeeds, and the smart contract reverts the state to a previous correct state. A problem with o-rollups is that the challenge period needs to be long—several days to a

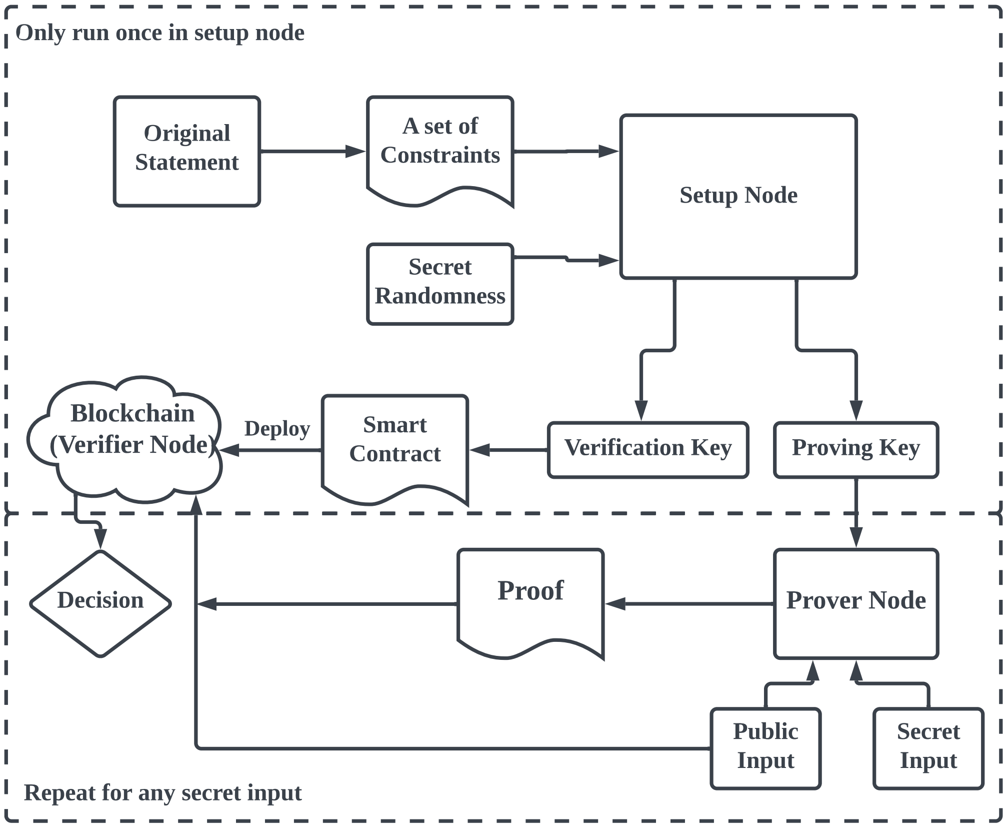

Figure 2.3: Components and flow of zk-SNARK.

week [57]—to provide an opportunity for challengers.

2.3.2 Zero-knowledge Rollups

Zk-rollups is a non-interactive solution based on a zero-knowledge proof mechanism [28, 67]. In zk-rollups, a digest includes a validity proof. The validity proof proves that the generated new digest MMR_{New} corresponds to a state of the data that is correct, i.e., the new state with digest MMR_{New} is the outcome of processing transactions on the previous state with digest MMR_{Pre}. The zk-SNARK protocol is one of the methods used to implement zk-rollups [13, 23, 25, 27, 64].

The zk-SNARK protocol is used in the following way by utilizing three components: a setup node, a prover node, and a verifier node (Figure 2.3):

- The setup node generates a proving key Pk_s and a verification key Vk_s that will be used to generate and verify proofs. Although these two keys can be published, the computation work to generate these two keys should remain a secret. Therefore, for zk-SNARK, the setup—which is a one-time process before operation—must be performed by a trusted node. After setup, there is no need for trusted nodes. The generation of the two keys is influenced by the type of computation that needs to be proven. The user provides the program to be proven/verified as well as the inputs to such computation. In RollStore, for example, the program to prove/verify is the one that updates the LSM tree and produces a new state represented by MMR_{New}; and the inputs to the program are the previous state and its digest MMR_{Pre} as well as the operations that are applied to the previous state to generate the new state.

- The prover node is responsible for generating the correctness proof of the computation. It needs three parameters, the proving key Pk_s, the public information, Inf_{pub}, and the secret information, Inf_{secret}. After collecting these parameters, the prover node generates a proof π_s of the computation outcome.

- The verifier node needs three parameters: the verification key Vk_s, the public information Inf_{pub}, and the proof π_s. After collecting these parameters, the verifier node generates a decision (True or False). In hybrid blockchains, the verifier can be a smart contract. Typical zk-SNARK protocols are designed so that verification is fast at the expense of a more lengthy proof generation process. This is suitable for hybrid blockchains, since generating proofs is performed by off-chain nodes that do not have the constraints of smart contracts, while verification is performed on-chain.

Chapter 3

System Design

In this section, we present the design of RollStore.

3.1 System Model and Interface

System components. RollStore consists of the following components (Figure 3.1):

- **Updater node:** the updater receives the write and read requests from clients. It maintains a mutable table T_{mut}, Level L_0 of the LSM tree, and a MMR tree for data in L_0, called MMR_0. Data in L_0 represents stage 0 committed data.
- **Backup node:** the backup maintains LSM levels (L_1 and L_2), and two MMR trees MMR_1 and MMR_2, each corresponding to an LSM level. L_1 contains data that is stage 1 committed (o-rollups) and L_2 contains data that is stage 2 committed (zk-SNARK).
- **Prover node:** the prover performs zk-SNARK computation to generate proofs of L_2 pages.

- **Smart contracts:** on-chain smart contracts handle the verification and maintenance of digests related to stage 1 and 2 committed data. Also, the smart contract handles the punishment strategy by verifying whether an off-chain node is malicious if a *challenge* is raised during stage 0 or 1 commitment. If the challenge indicates malicious activity, then the smart contract punishes the off-chain node by withdrawing funds from its escrow account.

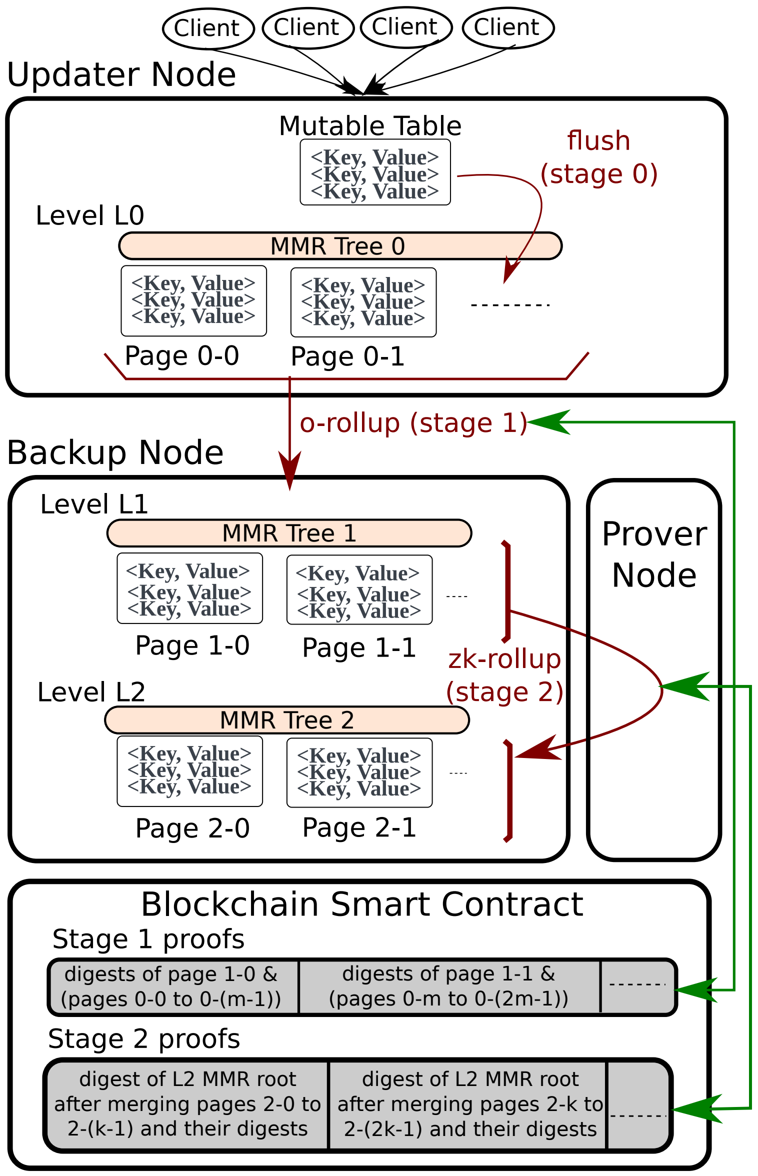

Figure 3.1: Data architecture of RollStore.

The three types of off-chain nodes can be co-located or placed across different machines.

Also, the three types of nodes can be elastically scaled, where more nodes of a node type are added to scale its computation, e.g., prover nodes can be added to speed up zk proof generation. We discuss scaling node types in Section 3.3.

Security model. Off-chain nodes (updaters, backups, and provers) are not trusted. They can deviate from the protocol in arbitrary ways, similar to byzantine failures [37]. Off-chain nodes can collude together and with clients. The smart contract logic executes correctly—without deviating from the protocol—due to running on blockchain. Write requests are assumed to be authenticated by a client, which prevents off-chain nodes from fabricating clients requests.

System Interface. RollStore provides a read, and write operation interface for users to read and write data.

1. **Write:** (In: key-value pair, Out: inclusion proof, sequence number): this call takes a key-value pair as the input, the output of this call is the inclusion proof for the key-value pair and the sequence number where it is added. Clients use the write interface to submit write requests to the updater node.
2. **Read:** (In: key, Out: value, inclusion proof): this function takes a key as the input, the output of the read operation is the corresponding value, and the inclusion proof for that value. The inclusion proof might be (1) local (stage 0), (2) global without fully verified (stage 1), or (3) global with fully verified (stage 2). Clients use the read interface to submit the read requests to the updater node.

Data model. The following are the main data structures maintained in RollStore (Figure 3.1):

1. **Distributed LSM tree:** The LSM Tree maintains the key-value pairs appended to RollStore. It has a mutable table and three levels. The mutable table T_{mut} is at the

updater node. T_{mut} holds the most recently appended entries that are being staged to be pushed to level L_0 of the LSM tree. Level L_0 maintains batches of appended data objects and is stored in the updater node. A page is added to L_0 only after a signed response is sent back to the clients with operations corresponding to the page's data objects. Pages are assigned a monotonically increasing sequence number Seq. We denote the i^{th} appended page to L_0 as page $P0_i$.

Level L_1 represents pages that are consolidated from level L_0. Before a page is written to L_1, its digest must be written on-chain as part of stage 1 o-rollups. Pages in L_1 are also assigned monotonically increasing sequence numbers, where the i^{th} page to be added to L_1 is denoted $P1_i$. Each page in L_1 represents a consolidation of pages in L_0. Therefore, page $P1_i$ represents the consolidation of pages $P0_{i*m}$ to $P0_{(i*m)+m-1}$ from L_0, where m is the threshold of the number of pages in L_0 to trigger a consolidation of pages in L_0 and creating a new page in L_1. Note that a new page in L_1 is added to the set of pages in L_1 and not merged with existing pages. Therefore, pages in L_1 may have overlapping key ranges.

Level L_2 represents pages that are merged from level L_1 after a successful zk proof is generated for them. Pages from L_1 are merged into L_2 in the order of their sequence numbers. Specifically, when the next k pages at L_1 are zk proven, then they are merged with the pages that already exist in L_2. Therefore, after i merge steps to L_2 (where $i = 0$ corresponds to the first step), the pages in L_2 contain the merged key-value pairs that represent pages $P1_{(i*k)}$ to $P1_{(i*k)+k-1)}$, which correspond to pages $P0_{i*k*m}$ to $P0_{(i*k*m)+(m*k)-1}$. Pages are merged from L_1 into L_2 which means that key-value pairs in L_2 are ordered across pages and each page has a unique range of key-value pairs that do not overlap with other pages in L_2.

2. **MMR trees:** The MMR trees are used to create compact digests of the data in the LSM tree. There are three MMR trees, each one corresponding to an LSM level;

MMR_0 for L_0 in the updater, and MMR_1 and MMR_2 for L_1 and L_2 in the backup. The digests of this MMR tree are used for the verification process of stage 1 and stage 2 committed data.

3. **On-chain digests:** the smart contract maintains a map of digests and proofs that are related to stage 1 and 2 committed data. Users query these digests to verify the authenticity of responses from off-chain nodes. There are two sets of digests/proofs. The first set is for stage 1 digests. In this set, each smart contract digest $SCDigest_i^1$ corresponds to page $P1_i$, which is a consolidation of pages $P0_{i*m}$ to $P0_{(i*m)+m-1}$, where m is the threshold for the number of pages in L_0 before consolidation. The second set is for stage 2 digests/proofs. In this set, each smart contract digest/proof $SCDigest_i^2$ corresponds to the i^{th} merge operation on L_2. The i^{th} merge operation in L_2 corresponds to merging the key-value pairs that are consolidated in pages $P1_{i*k}$ to $P1_{(i*k)+k-1}$, where k is the threshold of the number of L_1 pages to merge into L_2.

Commitment model. A write operation W of client c goes through three stages of commitment:

1. **Stage 0:** when the updater sends a signed response back to c, W is considered stage 0 committed. This signed response includes acknowledging the operation is received and promising to add it to page $P0_i$. This stage of commitment is the fastest as blockchain smart contracts are not involved. Client c can use the signed response later to prove maliciousness if an updater has lied (e.g., operation W is not included in $P0_i$ and later not included in $P1_{\lfloor i/m \rfloor}$, where m is the page threshold at L_0). A penalty smart contract receives punishment requests from clients that wish to prove maliciousness and punish malicious off-chain nodes.

2. **Stage 1:** consider the page $P0_i$ that includes W and the page $P1_{\lfloor i/m \rfloor}$ that is the consolidated L_1 page that includes $P0_i$. When the digest of $P0_i$ and $P1_{\lfloor i/m \rfloor}$ are

written as $SCDigest_i^1$ to the smart contract, the operation W is considered stage 1 committed. This takes longer than stage 0 commitment since the digests need to be written on-chain. However, it provides a stronger consistency guarantee—if two clients observe the state of a page $P0_i$ that is stage 1 committed, then they agree on the state of the page. However, this stage of commitment does not guarantee that the page itself is the result of correct computations. This is because the off-chain node can create a digest of arbitrary data. The client has to wait for the next stage of commitment to ensure that the derivation of the page is correct. However, if the off-chain nodes lie about stage 1 committed pages, they won't be able to perform stage 2 commitment. This consequently leads to clients sending a challenge request to the penalty smart contract that punishes the off-chain nodes.

3. **Stage 2:** Operation W is considered stage 2 committed when the following is true: the zk proof of a merge that includes page $P1_{\lfloor i/m \rfloor}$ is verified by the smart contract and written as $SCDigest^2_{\lfloor i/(m*k) \rfloor}$, where k is the page threshold at L_1. This is the strongest correctness guarantee as a page that is stage 2 committed is guaranteed to have been computed correctly by zk-SNARK. However, generating such proof is a complex process that may take the prover a long time to generate.

3.2 RollStore Core Design and Protocol

We now provide a description of RollStore's core design and protocols. This includes the protocols for read and write operations with a deployment of one updater, one backup, and one prover. We will describe the protocol as we follow the end-to-end life-cycle of write and read requests (Figure 3.2 shows the flow of operations that we refer to as *steps* in the rest of this section).

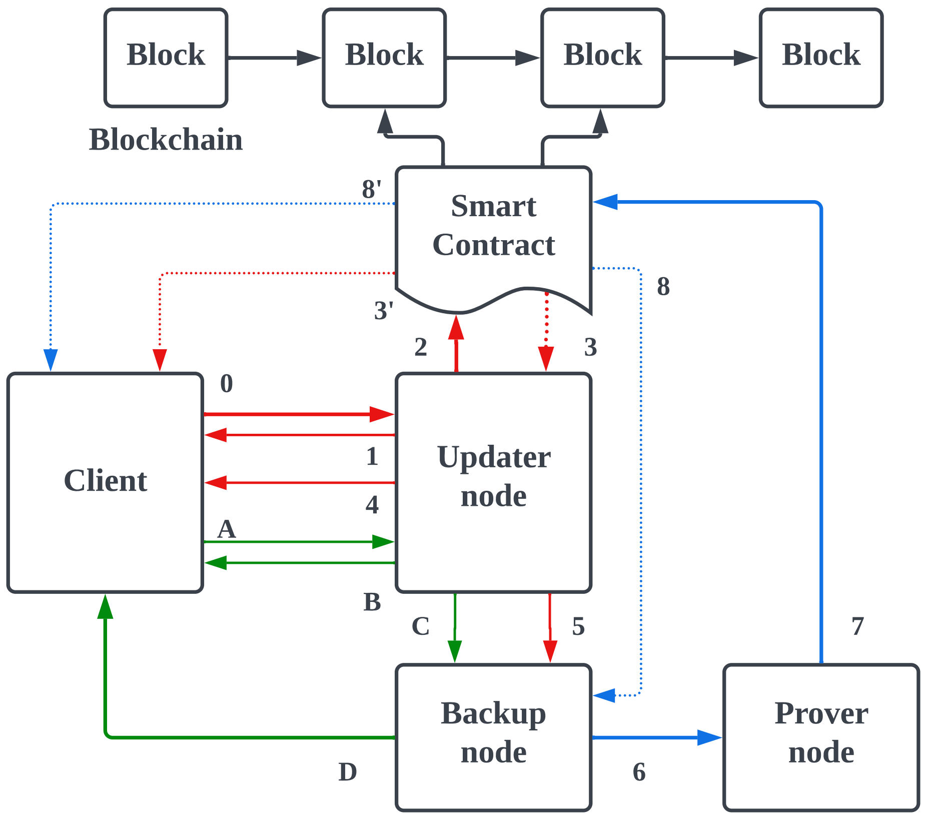

Figure 3.2: Lifecycle of RollStore requests. Red arrows represent stage 0 and 1 steps of write operations; blue arrows represent stage 2 steps of write operations; and green arrows represent steps of read operations.

Stage 0 commitment. A client c creates a signed write request W_i that has a key-value pair, $[K_i, V_i]$, and signature, S_c, as payload; $W_i = (S_c, [K_i, V_i])$. The signed write request is sent to the updater node (step 0 in the figure). The updater node, after receiving the signed request for W_i, adds W_i to the mutable table T_{mut} of the LSM tree. Once T_{mut} is full, the key-value pairs in T_{mut} are reordered by their key and written as a new page $P0_i$ in L_0 of the LSM tree located in the updater node. Each page in L_0 is assigned a monotonically increasing sequence number. This sequence number will be used by clients to track their operations and ensure that they are eventually stage 1 and 2 committed. Page $P0_i$'s sequence number is denoted Seq_i (if not mentioned otherwise, assume that $Seq_i = i$).

The MMR tree in the updater node is updated to include data in $P0_i$. At this point, a signed response, Ack_i, is sent back to client c for stage 0 commitment of W_i (step 1). This response includes: a stage-0 proof of inclusion of W_i in $P0_i$ (using the MMR tree) denoted $Prf^0_{W_i}$; also, Ack_i includes $P0_i$'s sequence number Seq_i and the updater's signature S_u; $Ack_i = (S_u, Seq_i, Prf^0_{W_i})$. At this point, client c considers the operation stage 0 committed and has a signed response that the updater node promised to include W_i as part of page $P0_i$ with sequence number Seq_i in the LSM tree. If the updater node does not honor this promise, then client c can use this signed response to trigger a punishment smart contract.

Stage 1 commitment. The updater node continues adding pages to L_0 until the threshold of the number of pages, m, is exceeded. At this point, Stage 1 commitment of pages in L_0 starts. The pages in L_0—including $P0_i$—are now mapped by the updater's MMR tree, MMR_0. A consolidated page $P1_{\lfloor i/m \rfloor}$ that conslidates the key-value pairs in pages in L_0 is created. The hashes and sequence numbers of the pages in L_0 and the hash of $P1_{\lfloor i/m \rfloor}$ are sent to the smart contract (step 2). The smart contract records this root hash as the stage 1 commitment o-rollups hash for the pages with the corresponding sequence numbers. This hash is recorded as $SCDigest^1_{\lfloor i/m \rfloor}$. Then, the smart contract emits an event to the updater node and clients about the new written hash and the corresponding page sequence numbers[1] (step 3 and 3'). The updater node sends a signed response to the client for stage 1 commitment (step 4). This response includes the digest of page $P1_{\lfloor i/m \rfloor}$ and the operation's inclusion proof.

After stage 1 commitment is performed for pages in L_0, all pages in L_0 and the consolidated page $P1_{\lfloor i/m \rfloor}$ are sent to the backup node to be inserted to L_1 (step 5). The original pages in L_0 are sent with $P1_{\lfloor i/m \rfloor}$ to the backup node as they will be used to provide inclusion proofs for read request as well as used to generate the zk proofs in stage 2 commitment. After sending $P1_{\lfloor i/m \rfloor}$ and the L_0 pages to the backup node, the pages in L_0 are cleared in the updater node. The page $P1_{\lfloor i/m \rfloor}$ will not be merged with pages in L_1, rather it will be

[1]A smart contract in permissionless blockchain cannot communicate directly to off-chain nodes. Here, we use the Ethereum emit operation that allows off-chain nodes to filter and pull emitted data of interest from the smart contract. Emit events in Figure 3.2 are shown as dotted arrows.

inserted as a new page. This means that the key-value pairs range of one page in L_1 may overlap with the ranges of other pages in L_1.

Stage 2 commitment. After page $P1_{\lfloor i/m \rfloor}$ is added to L_1, the backup node checks if the page threshold for L_1, denoted k, is met. If it did, the backup node starts the stage 2 commitment process using zk-SNARK for pages in L_1. This process merges the pages in L_1 with the pages in L_2. In the j^{th} merge operation, the pages to be merged from L_1 are from $P1_{j*k}$ to $P1_{(j*k)+k-1}$.

The merge is performed in the backup node. Then, the merge information is sent to the prover to generate a proof of the correctness of the merge. The information to prove the j^{th} merge includes: (1) pages $P1_{j*k}$ to $P1_{(j*k)+k-1}$, (2) pages $P0_{j*k*m}$ to $P0_{(j*k*m)+(k*m)-1}$, (3) pages in L_2, (4) the MMR root of L_2, MMR_{2-pre}, before the merge, and (5) the MMR root of L_2, MMR_{2-new}, after the merge (step 6). The prover node takes all this information to generate a proof that: (1) each page in pages $P1_{j*k}$ to $P1_{(j*k)+k-1}$ is generated correctly from the corresponding L_0 pages, (2) the merge of pages in $P1_{j*k}$ to $P1_{(j*k)+k-1}$ with pages in L_2 (with MMR root MMR_{2-pre}) yields a new state with MMR root MMR_{2-new}.

After the zk-SNARK proof is generated, it is sent to the smart contract to be validated (step 7). The smart contract performs the following: (1) it validates the proof, (2) verifies that the hashes used for L_0 and L_1 pages match the ones in stage 1 commit for the pages with the same sequence numbers, (3) verifies that MMR_{2-pre} corresponds to the previous verification, (4) record the new proof digest on-chain as $SCDigest_j^2$ for future access by clients, and (5) an event is emitted to the backup node and clients with operations in pages $P1_{j*k}$ to $P1_{(j*k)+k-1}$ (step 8 and 8'). The writes in $P1_{j*k}$ to $P1_{(j*k)+k-1}$ are now considered stage 2 committed. The backup node—once the proof is verified by the smart contract—writes the merged pages to L_2 and clear pages $P1_{j*k}$ to $P1_{(j*k)+k-1}$ from L_1.

Read operations. A client wishing to read a key x can request the level of the read request:

stage 0, stage 1, or stage 2 committed. We now show the process for a stage 0 committed read. (Stage 1 and stage 2 committed read follow the same process but starting from the backup node at level L_1 for stage 1, and L_2 for stage 2). First, the read request r is sent to the updater node (step A). When the updater node receives a read request, it responds with a signed response with the corresponding key-value pair and MMR inclusion proof (step B). The implications of this commitment is similar to stage 0 commitment for write operations where a read client can use the signed response as a proof of a lie by the updater node in the future.

If the requested key was not in L_0, then the read request moves to L_1 (this is also the start point of a stage 1 committed read). The client reads the most recent written stage 1 and 2 digests from the smart contract to match them with the response once received. The updater node forwards the request to the backup node (step C). When the backup node receives the read request, it responds with the corresponding key-value pair from a page in level L_1 with the MMR inclusion proof. The guarantee of this read request is similar to a stage 1 committed write where any two read requests would agree on the result but the result is still not verified by a zk proof.

If the requested key was not in L_1, then the read request moves to L_2 (this is also the start point of a stage 2 committed read). The backup node returns the requested key-value pair from L_2 if it exists (step D). The client can check the inclusion proof against the smart contract and verify that the read data object has been verified with a zk proof.

In both the stage 1 and stage 2 reads, the client reads the proof/digest from the smart contract prior to the beginning of the operation (note that unlike writing to blockchain smart contracts, reading data from a smart contract is a fast operation). Consider a read request that goes to a level L_i; if the data object does not exist in that level and the read is forwarded to L_{i+1}, then a proof of non-existence is also returned from L_i. This can be done by returning the pages with the ranges that overlap the requested key so that the client can verify that the key does not exist.

Example: Suppose the threshold of the mutable table T_{mut}, level L_0 of LSM in the updater node, and level L_1 of LSM in the backup node are 2, 3 and 1, respectively. The initial root hash value is $Root_0$. When the updater node received 2 write requests W_i $(i = 1, 2)$ from clients c_i $(i = 1, 2)$, it will store it as one page $P0_1$ and append the page $P0_1$ to its MMR tree. Then it will return the acknowledgement response Ack_i and sequence number Seq_1 to clients c_i $(i = 1, 2)$.

The acknowledgement response Ack_i contains inclusion proofs $Prf^0_{W_i}$; also, Ack_i includes $P0_i$'s sequence number Seq_i and the updater's signature S_u; $Ack_i = (S_u, Seq_i, Prf^0_{W_i})$. When clients c_i received the acknowledgement response Ack_i from updater node, they will mark their write requests W_i as stage 0 commitment. Now the new root hash value is $Root_1$. After collecting 3 pages in level L_0, the updater node uploads the new root hash value $Root_3$ and the range of sequence number $[1, 3]$ to the blockchain. The smart contract will store this new root hash value, change the optimistic root index $Index_{opt}$ to 1, and emit the optimistic root update event $Event_{opt}$. The clients notice the optimistic root update event $Event_{opt}$, and then mark their write requests W_i as stage 1 commitment.

The updater node also notice the optimistic root update event $Event_{opt}$. Now, the optimistic root index $Index_{opt}$ is 1, and the range of sequence number is from 1 to 3. At this time, the zero-knowledge root index $Index_{zk}$ is still 0, not 1, since we haven't proved $Root_3$ is correct or not. the updater node sends all pages in L_0 and the consolidated page $P1_{\lfloor i/m \rfloor}$ to the backup node.

After page $P1_{\lfloor i/m \rfloor}$ is added to level L_1, the backup node checks whether the threshold for level L_1 is met. Here we already set the threshold of level L_1 is 1, thus the backup node starts the stage 2 commitment process (also called merge operation). The backup node sends the prove task $Task_1$ to the prover node, this task tries to generate the proof that proves each page in level L_1 is generated correctly from the corresponding L_0 pages, and the merge of pages in level L_1 with pages in L_2 correctly yields the new state. The prover node finishes the proof task, generates the proof parameters π_1, and then sends it to the blockchain. The smart contract checks the new root hash value $Root_3$ is correct or not. If it is, the smart contract changes the zero-knowledge root index $Index_{zk}$ to 1 and emits the zero-knowledge root update event $Event_{zk}$.

The backup node will notice the zero-knowledge root update event $Event_{zk}$, and then append the page $P1_{\lfloor i/m \rfloor}$ in its level L_2 MMR tree, and merge the page from level L_1 to level L_2. At this time, the zero-knowledge root index $Index_{zk}$ is also changed to 1. The clients also notice the zero-knowledge root update event $Event_{zk}$, and then mark their write requests W_i as stage 2 commitment.

This design leverage the property of the blockchain, which reduces the cooperation works between multiple nodes and achieved the global consensus on the order of pages. The smart contract determined the global order and plays the leader role in this process, such as the leader node in the traditional consensus process.

3.3 Scaling Off-Chain Nodes

In this section, we discuss the scaling strategies for the three node types, updaters, backups, and provers. This allows each node type to utilize multiple nodes—instead of one node—to service requests and improve performance and/or resilience.

Scaling updater and backup nodes. The updater and backup nodes maintain LSM and MMR data. We discuss scaling these two types of nodes—which is increasing the number of updater and backup nodes to achieve higher throughput through distributing the workload. The scaling strategy is based on sharding the data into n shards. Each shard is maintained by a separate set of two nodes, one for the updater and the other for the backup. The smart contract is also deployed as n independent instances, one for each shard.

Although multiple overlapping updater and backup nodes are so attractive since they can scale the collection process more efficiently, thus improving the performance of our system, they also introduce more coordination work. Therefore, we only consider the case that each shard is maintained by a separate set of two nodes, the sets of two nodes handle a mutually exclusive range of the data in our system.

In this configuration, different updater nodes handle a different range of data, thus the order across the updater nodes has not broken the consistency of data, we only need to maintain the order of data in a single shard.

Scaling prover nodes. The prover node is tasked with generating the zk proofs of stage 2 commitment. Scaling the prove operation is important as it is a lengthy process. To scale proving tasks, we maintain n provers and distribute the zk proving workload across the n provers. Specifically, each zk proving task $Task_i$ is divided into n subtasks, $Task_{sub_1} \dots Task_{sub_n}$. Each subtask is responsible for proving N/n data items, where N is the total number of data items in the prove task.

Because we divided the proving task, the input for each subtask is going to be different from the original task. The original prove task $Task_i$ needs two root hash values (the old and new MMR root hashes), a set of peak points (information of the MMR tree), and the pages that contain the data. Each subtask, on the other hand, needs its own parameter. The process of proving subtasks can be considered as a sequence of proving tasks (that can be computed in parallel), each one handling a subset of the data. The proving task takes as input: (1) two root hash values that correspond to the MMR root before and after applying the subset of data items assigned to the subtask. (To provide this information, the backup node records the MMR root value after appending/merging each subset of data items). (2) the peak points that correspond to the state when the data objects where added in the backup node. (3) pages that contain the subset of the data that is assigned to the subtask.

Because the proof is now divided into n proofs for each original task, we need to adapt the verification process. When a subtask is done, the prover sends the partial proof π_i to the verification smart contract. The smart contract caches the partial proof and wait for the rest of the partial proofs to arrive. When all the partial proofs are received, the verification smart contract verifies the proofs and verifies that the partial proofs correspond to a complete original proof. This is performed by checking that there is a hash chain from the initial root of the first subtask (which is equal to MMR_{Pre}) to the root of the final subtask (which is equal to MMR_{New}). Because each partial proof contains the previous and next MMR roots, it is possible to verify that there is a hash chain from the hash of the first partial proof to the hash of the final one. If the chain is verified, the smart contract adopts the zk proof and continue stage 2 commitment.

This modification can distribute the huge prove task into smaller prove tasks, thus can leverage more computation resources to accelerate the proving process. However, it will introduce more transactions to the blockchain for proving one task, which would occupy the limited blockchain throughput, since the block size is limited, the bit of transaction is also limited, if we use too many bits to finish one task, it will definitely reduce the whole throughput. That is a trade-off.

Example: The total task is to verify the process of appending 128 key-value pairs, we can split this task into 4 subtasks, each subtask verifies 32 key-value pairs. The original total task only needs two root hash values (before and after), one set of peak points and 128 key-value pairs; however, each subtask needs its own parameters now, the intermediate parameters. We denote the initial root hash value as R_0, the final root hash value is R_4, 128 key-value pairs as KV_{1-128}, and initial peak points are P_0. The first subtask requires the initial root hash value R_0, first 32 key-value pairs KV_{1-32}, initial peak points P_0, and new root hash value R_1. This root hash value R_1 is calculated when the first 32 key-value pairs KV_{1-32} have been added to the MMR tree. We denote is as $Task_{sub_1} = \{R_0, KV_{1-32}, P_0, R_1\}$. Following the same logic, the second subtask is $Task_{sub_2} = \{R_1, KV_{32-64}, P_1, R_2\}$, the third subtask is $Task_{sub_3} = \{R_2, KV_{64-96}, P_2, R_3\}$, the last subtask is $Task_{sub_4} = \{R_3, KV_{96-128}, P_3, R_4\}$. All of these parameters are provided by the backup node during the stage 2 commitment processing, each prover can verify their own subtask parallelly.

Resilience and availability. Increasing the numbers of nodes can also serve the purpose of increasing the crash resilience and availability of RollStore. Specifically, for stateful node types—updaters and backups—the state of each node can be maintained by a replication cluster [14]. Therefore, the failure of one node can be tolerated by the rest of the nodes in the cluster. For stateless nodes—provers—adding and replacing provers is straight-forward, since the proving task is stateless. Therefore, in the case of a prover failure, it can be replaced by another node that takes over processing the requests from the backup node.

3.4 DApp-Indexing-as-a-Service Model

In this section, we discuss the payment model to enable a DApp-indexing-as-a-service model. In this model, each off-chain node deposits an amount of cryptocurrency to an escrow fund in the penalty smart contract in the setup stage. the addresses (Ethereum addresses and IP addresses) of off-chain nodes that successfully deposited the fund in the smart contract would be stored in the penalty smart contract, smart contract marked these off-chain nodes as valid server nodes.

The penalty smart contract is initialized with the following variables: $off-chain_{address_E}$, $off-chain_{address_{IP}}$, $off-chain_{deposit}$, and $off-chain_{signature}$. The first two variables store the Ethereum address and IP address of the off-chain node that signs up as a server node. The $off-chain_{deposit}$ variable set an amount on how much cryptocurrency (Ether) should be deposited to successfully signup. Variable $off-chain_{signature}$ stores the digital signature of the off-chain node that provides service.

Users can send their requests (writes or reads) to valid server nodes that successfully stored their addresses in the penalty smart contract. Here we assume users subscribe to the service (more details about the price policy are beyond the scope of this paper). The valid server nodes process these requests and interact with other nodes and the blockchain network.

As mentioned in section 3.2, the guarantees in stage 0 and stage 1 commitment are not final. This may lead to an off-chain node breaking the promise it provided during these two stages. To prevent this from happening, we incorporate a punishment strategy that is based on the monetary penalty that is managed by a penalty smart contract. If a malicious act is performed by the off-chain node, then the penalty contract is triggered—withdrawing the penalty from the escrow fund, and excluding this malicious node from valid server nodes. Victimized users can be compensated by the escrow fund.

3.5 Failure Examples

In this section, we discuss various malicious failure scenarios and explain how RollStore overcomes these failures. RollStore allows a server to act maliciously behaviors but it can detect and execute punishment on dishonest servers, thus preventing malicious behaviors.

Scenario 1: Incorrect Stage 0 commitment.

A malicious updater node can respond with incorrect responses for the stage 0 commitment. For example, given a writes request W_i, the malicious updater returns a wrong sequence number Seq_w, or wrong inclusion proofs $Prf^{w}_{W_i}$.

In stage 0 commitment, the updater provides a signed response back to the client that its write request W_i is part of a page $P0_i$ in L_0 with sequence number Seq_i. Prf^0_{Wi} is the signed inclusion proof of the write in page $P0_i$. An updater must use this page $P0_i$ during the o-rollups operation of stage 1 commitment. The client can verify that this is the case by observing the hashes that were written on-chain for stage 1 commitment. If the hash that corresponds to sequence number Seq_i is the same as the one received in the stage 0 response, then the promise is honored. Otherwise, the client starts the penalty process. The client sends a request to the penalty smart contract with the following input: the received stage 0 response received from the updater node, i.e., $Ack_i = (S_u, Seq_i, Prf^0_{W_i})$. The penalty smart contract verifies whether the penalty should be applied by verifying the authenticity of Ack_i (that it is indeed signed by the updater signature S_u), and checking whether the MMR root hash in Prf^0_{Wi} equals the o-rollups hash in the smart contract for the page with sequence number Seq_i. If the hash is different, then the penalty is applied.

Scenario 2: Incorrect Stage 1 commitment.

A malicious off-chain node can upload the wrong digest of stage 1 commitment or send the wrong pages used in the proof generation. For example, the client's write operation W_i is in page $P0_i$ with Seq_i. The stage 1 commitment hashes $Hash_o$ in the smart contract include the hash for $P0_i$ as well as $P1_i$ which is the merge result of page $P0_i$ and other L_0 pages. The malicious off-chain node can upload a wrong digest $Hash_w$ rather than $Hash_o$ or send wrong pages $P1_w$ used in zk proof generation

In stage 1 commitment, the client observes the o-rollups hash that is written to the smart contract for stage 1 commitment. The client's write operation W_i is in page $P0_i$ with Seq_i. The stage 1 commitment hashes in the smart contract includes the hash for $P0_i$ as well as $P1_i$ which is the merge result of page $P0_i$ and other L_0 pages. RollStore protocols ensure that these hashes are the same ones that will be used in stage 2 commitment. This is done because the smart contract—when verifying the proof in stage 2 commitment—verifies that the hashes used to generate the zk proof are identical to the ones used in stage 1 commitment by o-rollups. This is performed by checking which hashes were written to the smart contract during stage 1 for the corresponding pages used in the proof generation. In the case of W_i, this includes the hashes for pages $P0_i$ and $P1_i$. Since this is guaranteed by the verification process in stage 2 commitment, the off-chain nodes must keep their promise in using the stage 1 pages in stage 2 commitment. If they stage 1 commit a false digest then they would have to indefinitely delay the stage 2 commitment. Next, we show how to handle such impact.

Scenario 3: Delaying.

In both stage 0 and stage 1 commitment, another type of malicious act that the off-chain nodes may do is to delay the next stages of commitment indefinitely. In this case, the client performs a two-step process to prove and punish the off-chain nodes. Consider the case of a client—with operation W_i in $P0_i$—that received a stage 0 or stage 1 response r at time t. If the user suspects that the off-chain nodes are not continuing the processing of the request and future stages of $P0_i$. The first step is to send a delay-notification request to the penalty smart contract. The input to this notification is a proof that a signed response is received from the off-chain node for page $P0_i$. The smart contract records this notification with the blockchain block number that it was written in, b_i^1. Now, the off-chain node have an opportunity to finalize the commitment of $P0_i$ before the second step.

The second step can only be triggered at a future block b_i^2, where $b_i^2 - b_i^1 > bt$, where bt is a threshold on how many blockchain blocks should have passed before the second step can be triggered. This is a predefined number that is agreed on by the off-chain nodes and should be sufficiently large to allow for processing requests. If bt blocks passed and the client still observes that $P0_i$ is not committed, it starts the second step by sending a delay-followup request. This request references the first step. The penalty smart contract checks that bt blocks have been committed since the previous notification and if $P0_i$ is still not committed. If both conditions are true, then the penalty logic is applied and funds are withdrawn from the off-chain node escrow fund.

This strategy can be applied separately for stage 0 and 1 commitment delays where there is a threshold *bt* for each type of commitment. We use a block number threshold as it is a standard practice in smart contract development. The reason for using block numbers between requests is that it is predictable since the commitment of a block typically takes a predefined amount of time. Also, the block numbers cannot be manipulated by miners, whereas block timestamps might not be accurate.

3.6 Safety

In this section, we discuss the safety of read and write operations in RollStore. In particular, we prove that the guarantees of each level of commitment are met. Then, we discuss the data consistency properties of RollStore.

3.6.1 Stage 0 Safety

The following is stage 0's safety guarantee:

Theorem 3.1. ***(Stage 0 safety guarantee)*** *For a write w that is stage 0 committed in page $P0_i$ with sequence number i, either (1) the write w is going to be part of page $P0_i$ that is committed in stage 1 as part of the o-rollups in the consolidated page $P1_{\lfloor i/m \rfloor}$, where m is the threshold of the number of pages in L_0; or (2) the client can prove that the updater provided a false promise to include w in page $P0_i$.*

Proof. We prove this statement by contradiction. Assume to the contrary to the defined guarantee that there is a write w that is stage 0 committed as part of page $P0_i$, however, (1) the off-chain node used another page $P0'_i$ (with the same sequence number as $P0_i$) during stage 1 commitment for page $P1_{\lfloor i/m \rfloor}$, and (2) the client cannot prove the fake promise about w.

If page $P0'_i$ was used in stage 1 o-rollups of $P1_{\lfloor i/m \rfloor}$ instead of $P0_i$, this means that the stage 1 digest written on-chain, $SCDigest^1_{\lfloor i/m \rfloor}$ foe page $P0_i$ is different from the one returned to the client during the response (step 1 in Figure 3.2). This is because any change to the contents of the page would lead to a different digest. Therefore, the client knows that the off-chain node lied by detecting the different digests. The client can then prove that the off-chain node promised to include w as part of $P0_i$ by showing the signed response received in step 1. This is a contradiction, which proves the guarantee. □

3.6.2 Stage 1 Safety

The following is stage 1's safety guarantee:

Theorem 3.2. ***(Stage 1 safety guarantee)*** *For a write w that is stage 1 committed in page $P1_j$ with sequence number j, the following is guaranteed: the write w in $P1_j$ is going to be part of the $\frac{j}{k}^{th}$ merge to L_2, where k is the threshold of the number of pages in L_1.*

Proof. We prove this statement by contradiction. Assume to the contrary to the defined guarantee that another page $P1'_j$ with sequence number j—that does not include w—was included in the merge to L_2. This means that the digest $SCDigest^1_j$ (which corresponds to $P1_j$) in the smart contract is different than the digest of the page $P1'_j$. However, during the smart contract verification of the merge proof, part of the verification is that the digest of L_1 pages used in the merge are equivalent to the ones that were written to the smart contract during stage 1 commitment; this includes $SCDigest^1_j$. This means that the proof verification in the smart contract will fail, which is a contradiction, which proves the guarantee. □

3.6.3 Stage 2 Safety

The following is stage 2's safety guarantee:

Theorem 3.3. ***(Stage 2 safety guarantee)*** *For a write w that is stage 2 committed (i.e., the corresponding L_1 page $P1_i$ is stage 2 committed as part of merge number j), the following is true: any stage 2 read operation will receive the key-value pair of w if it reads from any merge starting from merge j to merge $j' - 1$ where the first write w' that overwrites w is in merge j'.*

Proof. We prove this by contradiction. Assume to the contrary that a stage 2 read operation that reads from merge $j*$, where $j < j* < j'$, observes a value written by $w*$ that is different from the value written by w.[2] As part of the assumption, w is part of the state of L_2 as of merge j. Therefore, returning another write value $w*$ after merge $j*$, but before merge j' can happen in one of two ways: (1) a write $w*$ is introduced in a merge $\mathcal{J}$ between j and $j*$. This means that $w*$ is part of the L_0 pages that correspond to merge $\mathcal{J}$. This is a contradiction since we assume that the first write to overwrite w, w', is performed as part of merge j' that is after $j*$. (2) the updater returns the value of $w*$ that is not part of any merges between j and $j*$. However, to be returned and verified by the reader, the write $w*$ must be part of L_2. Being incorporated in L_2 necessitates that a zk proof was obtained for it in some merge $\mathcal{J}$ between j and $j*$. This is a contradiction since we assume that the first write to overwrite w, w', is performed as part of merge j' that is after $j*$. □

3.6.4 Isolation Guarantee

RollStore guarantees linearizability [32] of operations that are stage 2 committed. We focus our isolation guarantee discussion on stage 2 commitment since it represents the point of

[2]We ignore the trivial case when there are multiple writes to the same key of w in the merge j. In such a case, the most recent write—the one in the highest sequence numbered L_0 page—overwrites the others.

final commitment and verification of read and write operations.

Theorem 3.4. ***(Consistency of stage 2 operations)*** *Any history H of stage 2 operations is linearizable.*

Proof. A history H of read and write operations is linearizable [32] if (1) H is equivalent to some sequential history S, and (2) the partial time order $<_H$ is a subset of the equivalent sequential history order $<_S$.

First, we prove the first property—H is equivalent to some sequential history S. RollStore performs stage 2 merge operations one-by-one in the order of the pages in L_1 which in turn are consolidations of ordered L_0 pages. In particular, the j^{th} stage 2 merge operation commits the operations consolidated in pages $P1_{j*k}$ to $P1_{(j*k)+k-1}$, where k is the threshold of the number of pages in L_1. Now, we construct the equivalent sequential history S. Consider the commit point for write operations in $P1_{j*k}$ to $P1_{(j*k)+k-1}$ to be the time when the verification is performed and the proof is written on-chain. A read operation that reads a value written by w that is committed as part of the j^{th} merge is ordered in the sequential history to be between the j^{th} and $(j+1)^{th}$ merge. The history H is equivalent to this constructed sequential history S.

Second, we show that $<_H$ is a subset of $<_S$. This is trivial for write operations as the commit points are ordered by the smart contract so that the values committed in the j^{th} merge precedes the values committed in the $(j+1)^{th}$ merge. For read operations, consider a read r that reads a value committed in the j^{th} merge. Consider the following partial time order in H. The read r starts at time t_r^{start} and terminates at time t_r^{end}. The read algorithm checks the smart contract first to inquire about the most recent successful stage 2 merge in L_2. It receives the proof and digest for the j^{th} merge. Then, the read operation is serviced from the backup node. From this timeline, we deduce the following about the partial time ordering in H: (1) $t_j^{commit} < t_r^{end}$, where t_j^{commit} is the commit time of the j^{th} merge in the smart contract.

This is true because the read observes the commit digest/proof. (2) $t_{j+1}^{commit} > t_r^{start}$. This is true because the read observed the j^{th} merge in the smart contract, which is a point after t_r^{start}, therefore, the next merge must have happened after the start of the read operation. Therefore, r can be assigned a commit time in the history in any point between t_r^{start} and $minimum(t_r^{end}, t_{j+1}^{commit})$. This partial time ordering is part of the constructed sequential ordering in S. $\square$

Chapter 4

Evaluation

In this section, we experimentally evaluate the performance of RollStore in comparison to two bockchain-based databases (BBDBs): BlockchainDB [22] and BigchainDB [41]. We perform our experiments by deploying off-chain nodes on Chameleon cloud machines [35]. Each machine has two 64-bit Skylake CPUs with 192 GB of RAM and 300 GB of storage. We used the Zokrates [21] framework to implement the zk-SNARK proof mechanism. If not specifically stated, the underlying blockchain network we evaluate is the Ropsten network, a widely used Ethereum test network.

Default configuration. For each experiment, we use the following default configuration. The threshold of the mutable table T_{mut}, level L_0, and level L_1 are set to 64 writes, 7 pages, and 3 pages, respectively. The default batch size is 512. The main variables we vary are the batch size and the number of server nodes.

Benchmark. We use the Yahoo! Cloud Serving Benchmark (YCSB) to generate the workload for experiments [17]. YCSB is a key-value store benchmark that offers various workloads. In our experiments, we choose two types of workload: (1) Workload A, which consists of 50% write operations and 50% read operations, and (2) Workload C, which consists of all read operations. We use a uniform distribution to choose keys for operations.

Evaluation objectives. Our evaluation answers the questions:

- What are the performance characteristics of RollStore in terms of throughput, transaction cost, and latency?
- How does varying the batch size and the number of server nodes influence RollStore's performance?
- How does the performance of our system compare to other hybrid blockchain-based database systems?

Metrics. The metrics we measure are:

- Throughput: this metric represents the throughput in terms of operations per second. We measure and report the throughput for each stage of commitment.
- Transaction cost: the transaction fee cost incurred by optimistic rollups (stage 1 commit) and zk proving (stage 2 commit) in terms of dollars per thousand operations (here we assume that the Ether price is \$1500). Although the base gas fee may fluctuate, it is relatively stable when we performed our experiments. The cost in Ether indicates the resource consumption on blockchain for methods accurately.
- Latency: the average latency to perform the three stages of commitments for writes and the average time to serve read requests.

Comparisons. We compare with two blockchain-based databases, BlockchainDB (Ethereum-based, permissionless) and BigChainDB (Tendermint-based, permissioned) with different characteristics as we describe next:

- BlockchainDB [22]: BlockchainDB is a hybrid BBDB that utilizes a blockchain layer as a storage layer and builds a database layer on top of it. We focus on the performance of read and write operations in BlockchainDB when a permissionless blockchain is used[1]. BlockchainDB stores data on-chain. This all on-chain solution would lead to high monetary costs and latency overhead for write operations. For both reads and writes, the operation is first performed off-chain nodes (BlockchainDB-1), which is done fast, and then performed on-chain (BlockchainDB-2), which is fully on-chain execution. Here we used the Ethereum testnet, Ropsten network.

- BigChainDB [41]: BigChainDB is a BBDB that is implemented as a permissioned blockchain. A byzntine agreement protocol, Tendermint, is used to implement a blockchain ledger and a database layer runs on top of this blockchain. This makes BigChainDB not suitable for DApps that require decentralization and open membership (permissionless) blockchains. However, we include it in our evaluation to understand the differences in performance characteristics compared to RollStore. Being on a permissioned blockchain, BigChainDB does not incur monetary costs. Also, because the database layer is integrated with the permissioned blockchain layer, the performance of operations is dependent on the performance of the underlying consensus mechanism.

We utilize and adapt available implementations of both BlockchainDB [22] and BigChainDB [41] that are made as a part of a study of hybrid BBDBs performance [26].

[1]BlockchainDB is designed for permissioned settings. We make it with permissionless settings here as it is the closest BBDB that can be adapted to utilize permissionless settings. We also compare with BigChainDB while maintaining its permissioned settings.

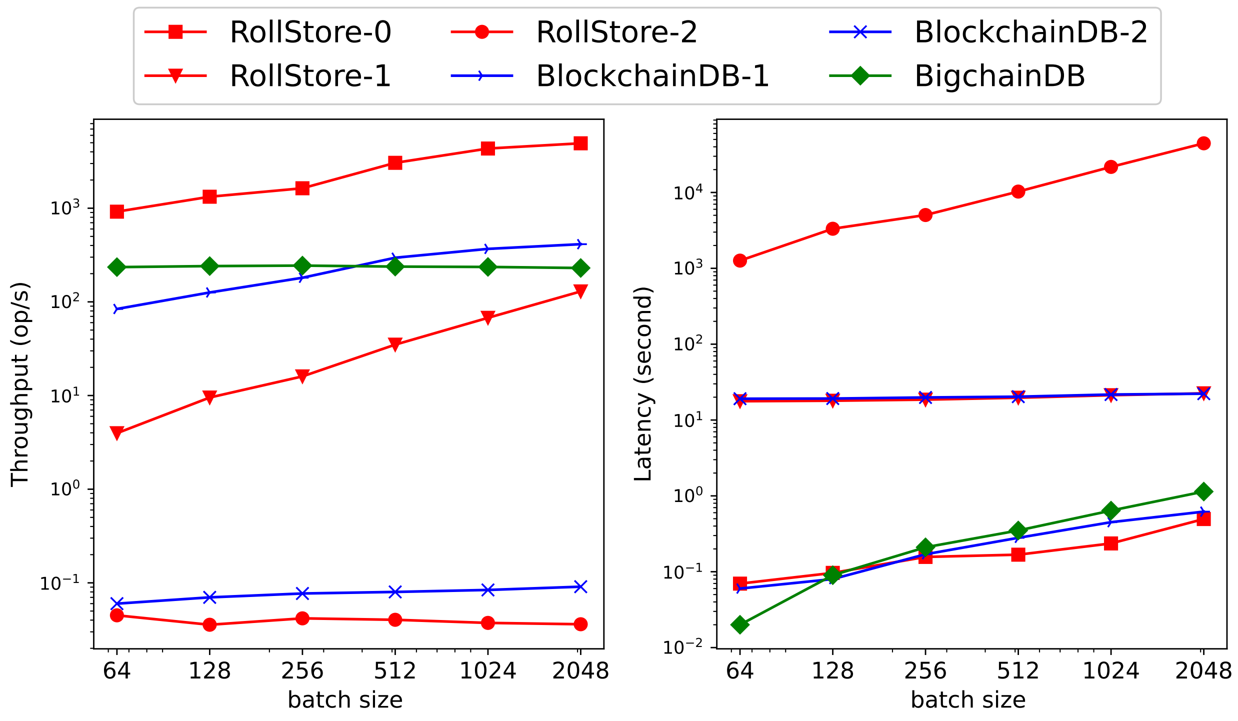

Figure 4.1: Throughput and latency in different batch sizes.

4.1 Baseline performance

In baseline experiments, we configure RollStore to have one updater node, one prover node and one backup node. The three nodes are located on three different machines and we use YCSB's workload A. The following experiments are performed while varying the batch size from 64 to 2048 operations/batch.

Throughput: The left side of Figure 4.1 show the throughput results. RollStore stage 0 commitment (RollStore-0) achieves the highest throughput (4931 TPS). This is because all processes in stage 0 are performed locally and do not need to coordinate with the smart contract. Both stage 0 and stage 1 (RollStore-1) throughputs increase with the increase in the batch size. Batching amortizes the cost of committing operations. In the case of RollStore-1, when batches are bigger, this means that the number of writes to blockchain is lower, which increases performance. Stage 2 (RollStore-2) throughput is the lowest as it requires performing a compute-intensive proof generation process. Unlike RollStore-0 and RollStore-1, RollStore-2's performance does not change significantly while increasing the batch size; this is because the performance is dominated by the time to perform proof generation. Also, as the batch size increases, the difficulty of the proof generation increases as well.

The local (off-chain) throughput of BlockchainDB (BlockchainDB-1) achieves a higher performance compared to RollStore-1 because it does not require interacting with the blockchain. However, for operations to be committed on-chain (BlockchainDB-2), it performs worse than RollStore-1 and is close to RollStore-2. This is because BlockchainDB-2 writes raw data on-chain which increases the overhead of interacting with blockchain. BigChainDB performance is between RollStore-0 and RollStore-1. This is because it does not utilize permissionless blockchain, which means that it does not suffer from the high overhead associated with it. However, BigChainDB still needs to incur the latency overhead of the underlying permissioned blockchain and the consensus mechanism, which leads to it performing worse than RollStore-0.

Latency: The right part of Figure 4.1 shows the latency results. The latency of off-chain operations—RollStore-0, BlockchainDB-1, and BigChainDB—are the lowest as they do not need to write to a permissionless blockchain smart contract. The latency of RollStore-1 and BlockchainDB-1—both requiring a write to the smart contract—is similar at around 20 seconds, which is proportional to the time to write to the smart contract. RollStore-2 is the slowest, even worst than the fully on-chain solution (BlockchainDB-2) as it requires going through the zk proof-generation process. However, this latency can be reduced by adding more prover nodes (see Sections 3.3).

Transaction cost: Figure 4.2 shows the monetary cost results. In RollStore-1, each batch requires sending one transaction only—that writes a simple set of digests—to blockchain. Therefore, the transaction cost in stage 1 will decrease when the batch size becomes larger.

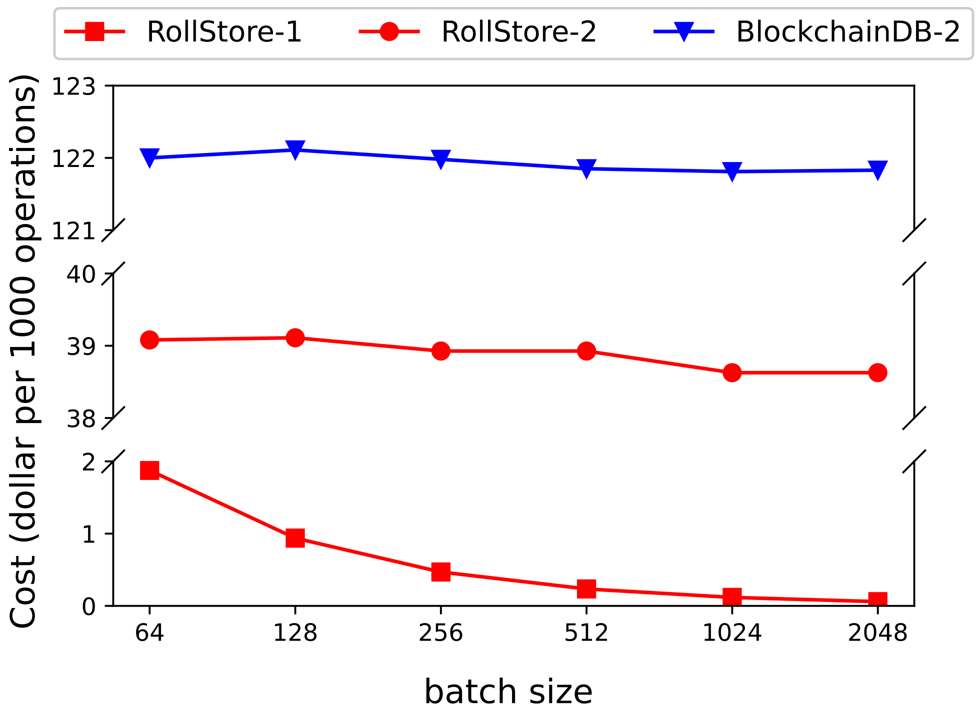

Figure 4.2: Cost in different batch sizes.

This is not the case in stage 2. Since we need to send the proof parameters to blockchain, the size of these parameters also increases with the increase in the batch size; this increases the cost. For this reason, the transaction cost per thousand operations in stage 2 does not change significantly when the batch size becomes larger. For BlockchainDB, the monetary cost is the largest (around $122 per 1000 operations). This is because raw operations are written on-chain, unlike RollStore that only writes digests and verifies proofs. (RollStore-0, BlockchainDB-1, and BigChainDB are not shown in this figure since they do not interact with a permissionless blockchain that requires monetary fees).

Read latency: Figure 4.3 shows the read latency while varying the batch sizes. The average read latency of RollStore becomes larger when increasing the batch size of reading requests. This is because when the batch size becomes larger, the backup node needs to spend more time searching and generating the related proofs. The read latency in BlockchainDB is higher than RollStore when the batch size is larger than 512; we attribute this to the *Verify*

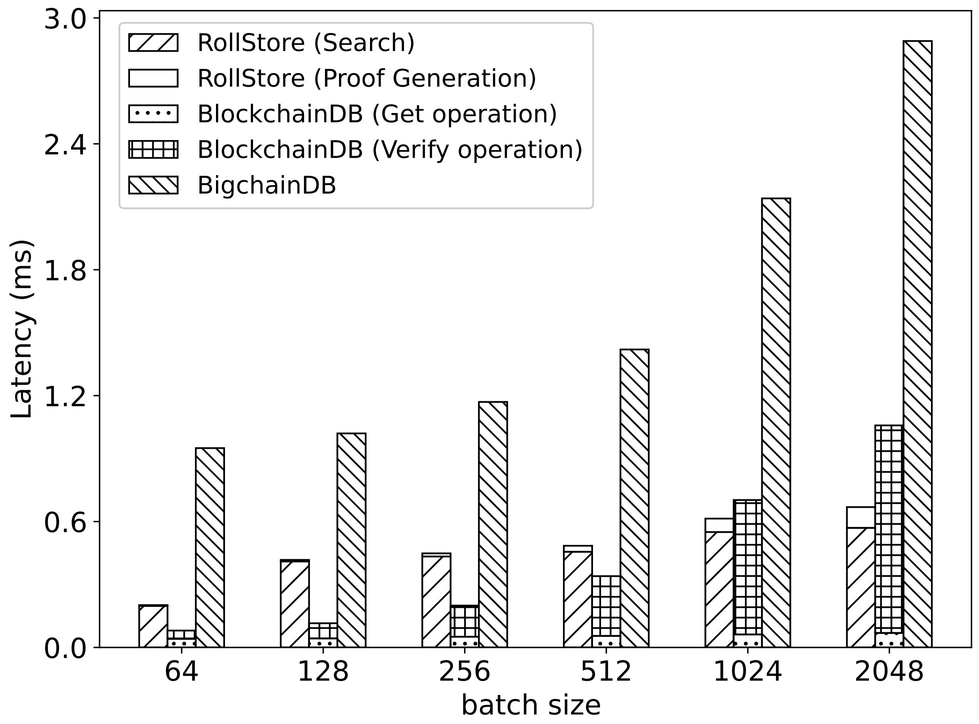

Figure 4.3: Read latency in different batch sizes.

operation in BlockchainDB. This operation spends more time to verify the read result when the batch size becomes larger. The read latency in BigchainDB is the longest and becomes longer when the batch size increases; this is because—although it does not need to perform a consensus round for reads—BigchainDB needs to build the block to record the read request; this process increases the read latency.

4.2 Scalability performance

In this section, we present a set of experiments to test the scalability performance of our system. In this configuration, multiple updater nodes, multiple prover nodes, and multiple backup nodes are located on three different machines. Each machine contains multiple instances of a type of node. We evaluate scalability by changing the number of server nodes and fixing the batch size of requests at 2048. We vary the number of server nodes from 4 to

13.

Throughput: In this set of experiments, we measure the throughput (left side of Figure 4.4). When the number of server nodes becomes larger, RollStore-0 and RollStore-1's throughput becomes larger. This is because more updater and backup nodes are working in-parallel. This observation also shows that we can achieve higher throughput with smaller batch sizes by adding more server nodes, which would also lead to reducing the latency in stage 2. The throughput of stage 2 commitment (RollStore-2) increases by a factor of 18.3X when increasing the number of server nodes from 1 to 13. This is because we can leverage the computation resources of multiple prover nodes to accelerate the proof generation process (see Section 3.3). Although this throughput is not very high compared with stage 0 and stage 1, it still significantly reduces the waiting time (challenge period) for verifying the results of stage 1 commitment (see Section 2.3.1).

Another observation is that the throughput of BlockchainDB and BigchainDB decreases when adding more server nodes compared to one server node. We attribute this as the cost of the underlying consensus mechanism—where a larger number of nodes causes the overhead of coordination it increases.

Latency: As shown in the right side of Figure 4.4, the reduction of latency in RollStore-0 and RollStore-1 is not significant. This is because the latency is mainly determined by the updater node processing for RollStore-0 and the blockchain confirmation time for RollStore-1. The latency of RollStore-2, on the other hand, is reduced significantly because multiple prover nodes work in parallel to generate the proof of one task.

BlockchainDB and BigchainDB do not benefit significantly from adding more server nodes. This is due to the increase of coordination overhead in the underlying consensus mechanism. RollStore leverages uploading the digest of a batch of transactions to speed up the consensus process and splitting the whole proof task into several smaller subtasks thus can benefit from

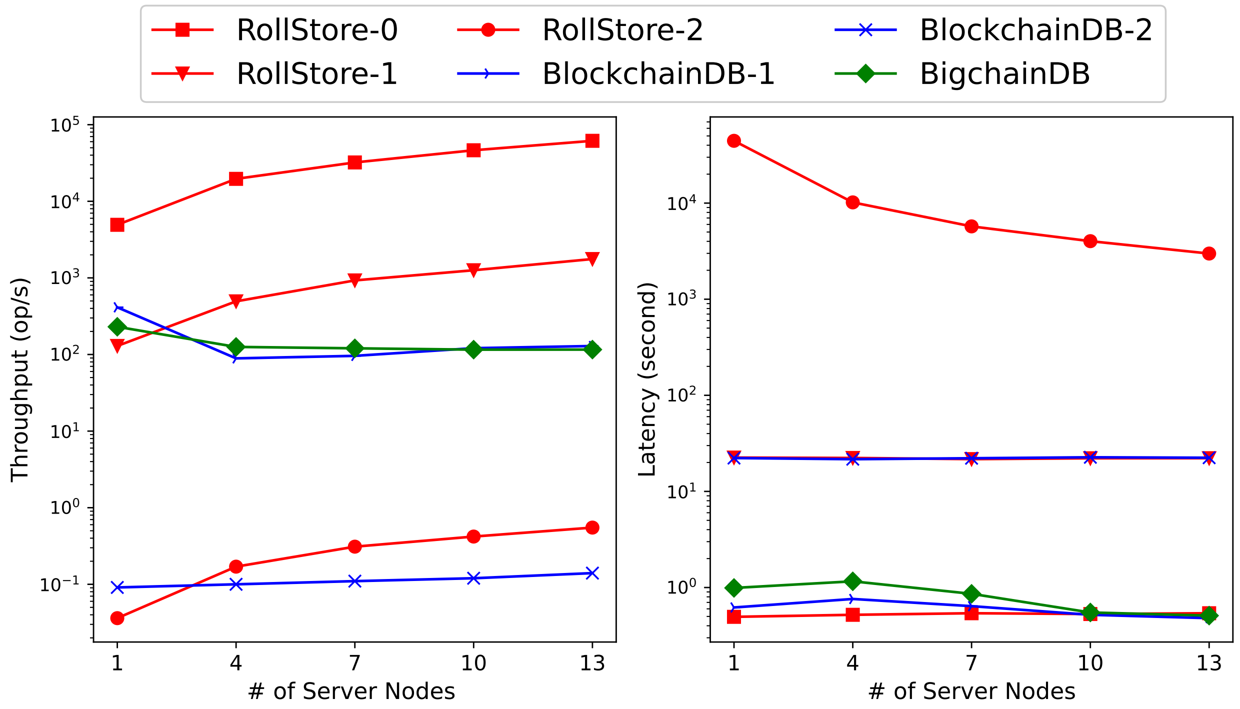

Figure 4.4: Throughput and latency in multiple server nodes.

adding more server nodes.

Transaction cost: Since the content of transactions and smart contracts do not change when we add multiple server nodes, the change of transaction fee (Ether cost) is negligible and is only due to the fluctuation of gas fees. The Ether cost in our system is determined by the structure of Ethereum and the complexity of the transaction content, adding more serevr nodes would not change anything in these two things.

Read throughput: Figure 4.5 shows the read throughput while varying the number of server nodes. RollStore and BlockchainDB are not impacted by the increase in the number of server nodes. This is because the throughput is determined by the overhead of assembling the read responses and verifying reads. BlockchainDB achieves lower performance than RollStore due to its verification step that takes more latency than RollStore for large batch sizes. BigchainDB achieves the lowest throughput due to the added overhead to synchronize the response to the read operations. As the number of nodes increases, this overhead increases

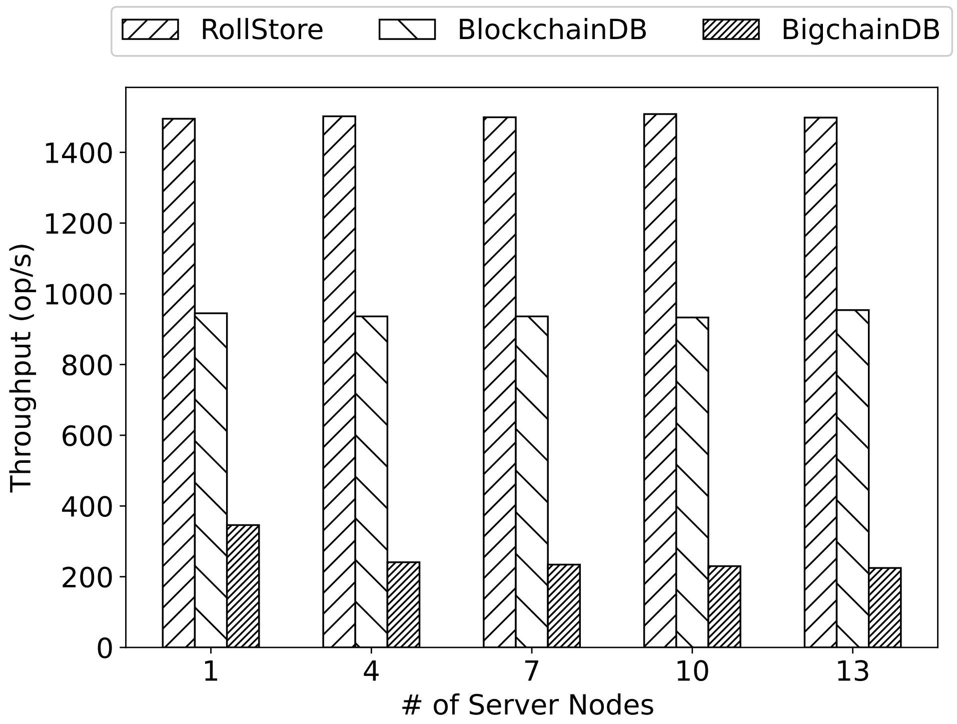

Figure 4.5: Read throughput in multiple server nodes.

and lowers the throughput of BigChainDB.

An example of result is shown in Table 4.1. It shows that in multiple server nodes configuration, RollStore achieves better performance than the baseline design, with improvement in stage 0 throughput by up to 12.4-times (from 4931.26 operations per second to 61574.76 operations per second), stage 1 throughput by up to 13.6-times (from 129.31 operations per second to 1762.36 operations per second), stage 2 throughput by up to 18.3-times (from 0.03 operations per second to 0.55 operations per second). As for latency, multiple server nodes configuration also achieves better performance than single server node, reducing stage 2 latency by up to 14.9-times compared to the baseline case (from 44473.41 seconds to 2986.21 seconds).

Combining these results, RollStore can achieve higher throughput, lower stage 2 latency by adding more server nodes, it shows the scalability of RollStore.

Table 4.1: Experimental Results of RollStore for Batch Size 2048

Metrics	**Baseline**	**13 nodes**
Stage 0 Throughput (op/s)	4931.26	61574.76
Stage 1 Throughput (op/s)	129.31	1762.36
Stage 2 Throughput (op/s)	0.03	0.55
Stage 1 Average Cost (USD)	0.0586	0.0584
Stage 2 Average Cost (USD)	38.5276	38.5321
Stage 0 Latency (s)	0.49	0.54
Stage 1 Latency (s)	22.49	21.68
Stage 2 Latency (s)	44473.41	2986.21
Read Throughput (op/s, with proof)	1494.99	1498.29

Chapter 5

Related Works

RollStore is related to prior work primarily in the areas of blockchain-based databases (BBDBs) and blockchain rollups; it also draws inspiration from the areas of secure and authenticated processing, distributed indexing, and monitoring to detect byzantine behaviour.

Blockchain-based databases (BBDBs). BBDBs are databases that utilize blockchains in various ways to utilize blockchain's features such as transparency and immutability [9, 20, 22, 41, 47, 52, 56]. Most of this work targets *permissioned* blockchain settings, where the blockchain network has a closed-membership assumption, i.e., all the participants in the blockchain network are authenticated and known. This permissioned setting allows faster and cheaper processing which makes it suitable for enterprise and consortium (multi-organization) applications. However, the closed-membership assumption of permissioned blockchain prevents their use in DApps that require open-membership and not relying on a single or group of fixed members. We target supporting these DApps which is now a large market with hundreds of thousands of users and hundreds of millions of dollars in assets [1, 53, 65]. For this reason, we tackle the unique challenges that are faced when building a BBDB over permissionless blockchains. Due to their focus on permissioned blockchains, prior

BBDBs do not factor in the monetary cost and latency challenges of using permissionless blockchain. This led to them being unsuitable for DApps due to high costs and latency from writing raw data directly to blockchain [9, 20, 22, 41].

Related to this category are databases that are built as a permissioned blockchain [11, 30, 49]. These are distributed ledger technologies that aim to mimic the immutability and trust features of blockchain and typically use byzantine agreement protocols. Similar to BBDBs, these solutions cannot be used in DApps due to their permissioned settings which does not factor the monetary and performance cost of utilizing a permissionless blockchain.

Blockchain rollups. Blockchain rollups was proposed as a layer-2 scaling solution for blockchains [57] (see Section 2.3 for an overview). RollStore builds on the advances of optimistic and zk rollups. RollStore advances the state of the art in rollups by extending them to be used within a hybrid data indexing solution. Also, RollStore proposes a multi-stage processing solution that combines optimistic *and* zk rollups. Prior work utilizes either one of the two rollups strategies—suffering from the disadvantages of the chosen method. RollStore combines the two in a manner that allows benefiting from their advantages while masking their disadvantages. In particular, o-rollups digests can be written faster on-chain but their challenge period takes a long time up to days [57]. On the other hand, zk rollups' time to generate the digest/proof to be written on-chain is longer than o-rollups time to write the digest; but, that proof is sufficient to finalize the commitment of the operation without having to wait for days in a challenge period. RollStore's design allows enjoying the benefits of fast o-rollups digest writing (stage 1) as well as the finality of zk rollups (stage 2). Finally, RollStore introduces a new kind of rollups that we utilize in stage 0 that is much faster than other kinds of rollups as it does not require writing on-chain. This is possible via a penalty strategy using a penalty smart contract.

Secure and authenticated off-chain processing. There have been a lot of recent work on utilizing off-chain nodes to perform compute and storage tasks for blockchain applications [7, 29, 36, 43, 45]. This is because utilizing off-chain nodes can reduce the monetary cost and performance overhead of blockchain applications. The challenge that is faced by many works in this category is how to utilize off-chain nodes that might be untrusted. For this reason, trusted and authenticated data structures were used to provide trust on the outcome of off-chain nodes' processing [59, 66, 69, 70]. These solutions focus on querying and storing data securely off-chain, but do not support operations that mutate the state of data, unlike RollStore and blockchain rollups.

Related to this category is the plethora of work in authenticated data and query processing [34, 38, 50, 68, 72, 73, 75]. These methods can be inherited and utilized in the context of querying and processing data in hybrid onchain-offchain applications [9, 59, 60].

Distributed indexing. RollStore can be viewed as a distributed index across an updater and backup node to store information, and smart contracts to manage data verification meta-information. Further, each node type can be distributed to scale for more workloads and/or more resilience. The distributed index of RollStore has unique properties when compared to existing distributed indexing solutions [6, 24, 33, 44] due to its focus on authentication in hybrid DApps. RollStore also proposes an integration of a regular indexing structure (LSM) and an authenticated indexing structure (MMR) to enable combining the performance and authentication properties of both types of solutions. This is related to some recent work that combines LSM with authenticated data structure for the same purpose [48, 54].

Byzantine monitoring. In stage 0, RollStore utilizes the concept of allowing malicious acts to be performed, but to guarantee detecting and punishing the malicious actor. This is related to the area of monitoring distributed systems for byzantine behavior [31, 40, 48]. These systems also allow nodes to act maliciously—to improve performance—while employing a lazy detection mechanism to ensure that malicious acts are detected, their impacts are retracted, and the culprit to be punished. These systems, however, consider traditional byzantine replication and logging applications and are not directly applicable to the case of hybrid indexing for DApps.

Chapter 6

Conclusion

In this paper, we propose RollStore, a data indexing solution for hybrid onchain-offchain DApps. RollStore builds on advances in blockchain scaling solutions such as rollups, as well as indexing and authenticated data structures. The outcome is a three-stage commit protocol that allows balancing the trade-off between security and performance for hybrid blockchain methods. RollStore utilizes zero-knowledge proofs for blockchain-based databases. This enables complete verification of data operations without the need to write or process raw data on-chain. Instead, the on-chain smart contract is only tasked with simple verification of proofs and maintenance of commit digests. These properties allow RollStore to reduce the monetary cost and performance overhead compared to existing blockchain-based databases. Our evaluations demonstrate the advantages of RollStore in terms of cost and performance while comparing with two blockchain-based databases, BlockchainDB and BigChainDB.

Bibliography

[1] Dapp radar rankings. https://dappradar.com/rankings, .

[2] decentraland. https://decentraland.org/, .

[3] Ethereum charts and statistics. https://etherscan.io/charts, .

[4] The sandbox game. https://www.sandbox.game/, .

[5] J. Adler and M. Quintyne-Collins. Building scalable decentralized payment systems. *arXiv preprint arXiv:1904.06441*, 2019.

[6] M. Y. Ahmad and B. Kemme. Compaction management in distributed key-value datastores. *Proceedings of the VLDB Endowment*, 8(8):850–861, 2015.

[7] H. Al-Breiki, M. H. U. Rehman, K. Salah, and D. Svetinovic. Trustworthy blockchain oracles: review, comparison, and open research challenges. *IEEE Access*, 8:85675–85685, 2020.

[8] A. Alkhateeb, C. Catal, G. Kar, and A. Mishra. Hybrid blockchain platforms for the internet of things (iot): A systematic literature review. *Sensors*, 22(4):1304, 2022.

[9] L. Allen, P. Antonopoulos, A. Arasu, J. Gehrke, J. Hammer, J. Hunter, R. Kaushik, D. Kossmann, J. Lee, R. Ramamurthy, et al. Veritas: Shared verifiable databases and tables in the cloud. In *9th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[10] N. Alzahrani and N. Bulusu. Block-supply chain: A new anti-counterfeiting supply chain using nfc and blockchain. In *Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems*, pages 30–35, 2018.

[11] M. J. Amiri, D. Agrawal, and A. El Abbadi. Sharper: Sharding permissioned blockchains over network clusters. In *Proceedings of the 2021 International Conference on Management of Data*, pages 76–88, 2021.

[12] A. M. Antonopoulos and G. Wood. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[13] P. Biel, S. Zhang, and H.-A. Jacobsen. A zero-knowledge proof system for openlibra. In *Proceedings of the 22nd International Middleware Conference: Demos and Posters*, pages 3–4, 2021.

[14] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a {High-Performance} data store. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[15] Z. Cao, S. Dong, S. Vemuri, and D. H. Du. Characterizing, modeling, and benchmarking {RocksDB}{Key-Value} workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[16] Y. Chen and C. Bellavitis. Blockchain disruption and decentralized finance: The rise of decentralized business models. *Journal of Business Venturing Insights*, 13:e00151, 2020.

[17] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[18] V. Costan and S. Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[19] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. Gün Sirer, et al. On scaling decentralized blockchains. In *International conference on financial cryptography and data security*, pages 106–125. Springer, 2016.

[20] H. Desai, M. Kantarcioglu, and L. Kagal. A hybrid blockchain architecture for privacy-enabled and accountable auctions. In *2019 IEEE International Conference on Blockchain (Blockchain)*, pages 34–43. IEEE, 2019.

[21] J. Eberhardt and S. Tai. Zokrates-scalable privacy-preserving off-chain computations. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1084–1091. IEEE, 2018.

[22] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy. Blockchaindb: A shared database on blockchains. *Proceedings of the VLDB Endowment*, 12(11):1597–1609, 2019.

[23] U. Feige, A. Fiat, and A. Shamir. Zero-knowledge proofs of identity. *Journal of cryptology*, 1(2):77–94, 1988.

[24] P. Garefalakis, P. Papadopoulos, and K. Magoutis. Acazoo: A distributed key-value store based on replicated lsm-trees. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 211–220. IEEE, 2014.

[25] A. Garoffolo, D. Kaidalov, and R. Oliynykov. Zendoo: A zk-snark verifiable cross-chain transfer protocol enabling decoupled and decentralized sidechains. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1257–1262. IEEE, 2020.

[26] Z. Ge, D. Loghin, B. C. Ooi, P. Ruan, and T. Wang. Hybrid blockchain database systems: design and performance. *Proceedings of the VLDB Endowment*, 15(5):1092–1104, 2022.

[27] O. Goldreich and H. Krawczyk. On the composition of zero-knowledge proof systems. In *International Colloquium on Automata, Languages, and Programming*, pages 268–282. Springer, 1990.

[28] O. Goldreich and Y. Oren. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7(1):1–32, 1994.

[29] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais. Sok: Off the chain transactions. *IACR Cryptol. ePrint Arch.*, 2019:360, 2019.

[30] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi. Resilientdb: Global scale resilient blockchain fabric. *Proceedings of the VLDB Endowment*, 13(6).

[31] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: practical accountability for distributed systems. *ACM SIGOPS Operating Systems Review*, 41(6):175–188, Oct. 2007.

[32] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[33] H. Huang and S. Ghandeharizadeh. Nova-lsm: A distributed, component-based lsm-tree key-value store. In *Proceedings of the 2021 International Conference on Management of Data*, pages 749–763, 2021.

[34] R. Jain and S. Prabhakar. Trustworthy data from untrusted databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 529–540. IEEE, 2013.

[35] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, et al. Lessons learned from the chameleon testbed. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 219–233, 2020.

[36] R. Kumar, N. Marchang, and R. Tripathi. Distributed off-chain storage of patient diagnostic reports in healthcare system using ipfs and blockchain. In *2020 International Conference on COMmunication Systems & NETworkS (COMSNETS)*, pages 1–5. IEEE, 2020.

[37] L. LAMPORT, R. SHOSTAK, and M. PEASE. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[38] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 121–132, 2006.

[39] C. Luo and M. J. Carey. Lsm-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418, 2020.

[40] S. Maiyya, D. H. B. Cho, D. Agrawal, and A. El Abbadi. Fides: managing data on untrusted infrastructure. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 344–354. IEEE, 2020.

[41] T. McConaghy, R. Marques, A. Müller, D. De Jonghe, T. McConaghy, G. McMullen, R. Henderson, S. Bellemare, and A. Granzotto. Bigchaindb: a scalable blockchain database. *white paper, BigChainDB*, 2016.

[42] R. C. Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[43] A. Miller, I. Bentov, S. Bakshi, R. Kumaresan, and P. McCorry. Sprites and state channels: Payment networks that go faster than lightning. In *International Conference on Financial Cryptography and Data Security*, pages 508–526. Springer, 2019.

[44] N. Mittal and F. Nawab. Coolsm: Distributed and cooperative indexing across edge and cloud machines. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 420–431. IEEE, 2021.

[45] R. Mühlberger, S. Bachhofner, E. Castelló Ferrer, C. D. Ciccio, I. Weber, M. Wöhrer, and U. Zdun. Foundational oracle patterns: Connecting blockchain to the off-chain world. In *International Conference on Business Process Management*, pages 35–51. Springer, 2020.

[46] K. B. Muthe, K. Sharma, and K. E. N. Sri. A blockchain based decentralized computing and nft infrastructure for game networks. In *2020 Second International Conference on Blockchain Computing and Applications (BCCA)*, pages 73–77. IEEE, 2020.

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

[48] F. Nawab. Wedgechain: A trusted edge-cloud store with asynchronous (lazy) trust. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 408–419. IEEE, 2021.

[49] F. Nawab and M. Sadoghi. Blockplane: A global-scale byzantizing middleware. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 124–135. IEEE, 2019.

[50] D. Papadopoulos, S. Papadopoulos, and N. Triandopoulos. Taking authenticated range queries to arbitrary dimensions. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 819–830, 2014.

[51] Q. Pei, E. Zhou, Y. Xiao, D. Zhang, and D. Zhao. An efficient query scheme for hybrid storage blockchains based on merkle semantic trie. In *2020 International Symposium on Reliable Distributed Systems (SRDS)*, pages 51–60. IEEE, 2020.

[52] Y. Peng, M. Du, F. Li, R. Cheng, and D. Song. Falcondb: Blockchain-based collaborative database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 637–652, 2020.

[53] C. Pop, T. Cioara, I. Anghel, M. Antal, and I. Salomie. Blockchain based decentralized applications: Technology review and development guidelines. *arXiv preprint arXiv:2003.07131*, 2020.

[54] P. Raju, S. Ponnapalli, E. Kaminsky, G. Oved, Z. Keener, V. Chidambaram, and I. Abraham. {mLSM}: Making authenticated storage faster in ethereum. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[55] F. Schär. Decentralized finance: On blockchain-and smart contract-based financial markets. *FRB of St. Louis Review*, 2021.

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

[57] C. Sguanci, R. Spatafora, and A. M. Vergani. Layer 2 blockchain scaling: A survey. *arXiv preprint arXiv:2107.10881*, 2021.

[58] P. Todd. Making utxo set growth irrelevant with low-latency delayed txo commitments (2016).

[59] H. Wang, C. Xu, C. Zhang, J. Xu, Z. Peng, and J. Pei. vchain+: Optimizing verifiable blockchain boolean range queries. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 1927–1940. IEEE, 2022.

[60] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, B. C. Ooi, and P. Ruan. Forkbase: An efficient storage engine for blockchain and forkable applications. *Proceedings of the VLDB Endowment*, 11(10), 2018.

[61] Y. Wang, Z. Tu, Y. Bai, H. Yuan, X. Xu, and Z. Wang. A blockchain-based infrastructure for distributed internet of services. In *2021 IEEE World Congress on Services (SERVICES)*, pages 108–114. IEEE, 2021.

[62] Q. Wei, B. Li, W. Chang, Z. Jia, Z. Shen, and Z. Shao. A survey of blockchain data management systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 21(3):1–28, 2022.

[63] H. Wu, J. Cao, Y. Yang, C. L. Tung, S. Jiang, B. Tang, Y. Liu, X. Wang, and Y. Deng. Data management in supply chain using blockchain: Challenges and a case study. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–8. IEEE, 2019.

[64] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica. {DIZK}: A distributed zero knowledge proof system. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 675–692, 2018.

[65] K. Wu. An empirical study of blockchain-based decentralized applications. *arXiv preprint arXiv:1902.04969*, 2019.

[66] C. Xu, C. Zhang, J. Xu, and J. Pei. Slimchain: scaling blockchain transactions through off-chain storage and parallel processing. *Proceedings of the VLDB Endowment*, 14(11):2314–2326, 2021.

[67] K. Yang, P. Sarkar, C. Weng, and X. Wang. Quicksilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2986–3001, 2021.

[68] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 5–18, 2009.

[69] C. Zhang, C. Xu, H. Wang, J. Xu, and B. Choi. Authenticated keyword search in scalable hybrid-storage blockchains. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 996–1007. IEEE, 2021.

[70] C. Zhang, C. Xu, J. Xu, Y. Tang, and B. Choi. Gemˆ 2-tree: A gas-efficient structure for authenticated range queries in blockchain. In *2019 IEEE 35th international conference on data engineering (ICDE)*, pages 842–853. IEEE, 2019.

[71] Q. Zhang, Y. He, R. Lai, Z. Hou, and G. Zhao. A survey on the efficiency, reliability, and security of data query in blockchain systems. *Reliability, and Security of Data Query in Blockchain Systems.*

[72] Y. Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vsql: Verifying arbitrary sql queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 863–880. IEEE, 2017.

[73] Y. Zhang, J. Katz, and C. Papamanthou. Integridb: Verifiable sql for outsourced databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1480–1491, 2015.

[74] Q. Zhou, H. Huang, Z. Zheng, and J. Bian. Solutions to scalability of blockchain: A survey. *Ieee Access*, 8:16440–16455, 2020.

[75] W. Zhou, Y. Cai, Y. Peng, S. Wang, K. Ma, and F. Li. Veridb: An sgx-based verifiable database. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2182–2194, 2021.