DC-Storage: A Fast Permissioned IoT Blockchain Storage with Decoupled Index

Haoyi Ding, Zeshun Peng(✉), Gang Wang, Xiaohua Li, Xiaomei Dong, and Ge Yu

Northeastern University, Shenyang 110819, China
{dinghy,pengzeshun,wanggang}@stumail.neu.edu.cn,
{lixiaohua,dongxiaomei,yuge}@mail.neu.edu.cn

Abstract. Permissioned blockchain has emerged as an effective solution to IoT data security challenges. With its decentralization, immutability, and transparency, integrating blockchain with IoT enhances security and improves data availability. Due to the performance limitations of underlying blockchains in handling massive IoT data, existing IoT blockchains typically adopt a two-layer architecture. In this architecture, the actual data is stored off-chain (e.g., in cloud servers or cloud storage) while only lightweight proofs are maintained on-chain. However, this approach faces several challenges. First, off-chain storage introduces vulnerability issues such as data loss and leaks. Second, synchronization between the off-chain data and the on-chain proofs becomes a performance bottleneck. To address these limitations, we propose DC-Storage, a fast permissioned IoT blockchain with a dual-chain architecture. We identify index construction as the primary performance bottleneck and decouple the index chain from the data chain to enable fast data chain construction with asynchronous indexing. Since transactions are considered committed after modifying the data chain, this significantly improves system throughput. Furthermore, we enable parallel index construction, substantially accelerating the indexing process. Experimental results show that DC-Storage outperforms its baseline with 5.03× higher throughput.

Keywords: permissioned blockchain · IoT data · storage · tree index

1 Introduction

Permissioned blockchains offer decentralization, immutability, and auditability, making them an effective solution to enhance data security [8,32], particularly in Internet of Things (IoT) applications [19]. By storing IoT data on-chain, enterprises can ensure data integrity and availability while eliminating the risks of storage failures and data tampering. However, the exponential growth in IoT devices poses significant performance challenges for blockchain-based IoT storage. Industry projections indicate that the number of IoT devices will surpass 75 billion by 2025 [24], representing a threefold increase from 2019 levels. This

© The Author(s), under exclusive license to Springer Nature Singapore Pte Ltd. 2026
J. Li et al. (Eds.): APWeb-WAIM 2025, LNCS 16115, pp. 599–615, 2026.
https://doi.org/10.1007/978-981-95-5719-6_38

massive scale of IoT data is common in real-world scenarios. For instance, in smart grid IoT platforms [2], each device generates real-time telemetry data from 800,000 measurement points. With the connection of 50,000 such devices, an IoT blockchain must store over 4 billion new records daily.

To address these performance challenges, existing IoT blockchains typically adopt a hybrid storage architecture, combining on-chain storage and off-chain storage [15,30]. In this architecture, the actual data is stored off-chain (e.g., in cloud storage services) while only lightweight proofs (i.e., cryptographic hash values) are maintained on-chain. This design allows users to verify data authenticity by comparing the off-chain data with the corresponding on-chain proofs. While this design improves system throughput by reducing on-chain storage overhead, it also introduces several challenges:

First, the off-chain storage lacks data availability. [15] utilizes cloud servers as off-chain storage. Although data integrity can be verified through on-chain proofs, cloud server failures may lead to data loss, preventing users from accessing the original IoT data. Even with data replication mechanisms, cloud servers cannot achieve the same level of availability as blockchain storage. Second, off-chain storage may compromise data privacy. While cloud storage services (e.g., Amazon S3 [20]) offer higher availability, they introduce additional security risks. Cloud storage services typically operate under a shared responsibility model, making them vulnerable to unauthorized access and breaches. Similarly, IPFS-based solutions [17,34,35] suffer from availability and privacy issues, as data accessibility depends on node availability and sensitive information is exposed across public nodes. Third, data persistence introduces significant latency overhead. Before recording the proofs on-chain, the on-chain storage must ensure that the corresponding data is persisted in the off-chain storage. This synchronization process involves multiple round-trip communications between the on-chain and off-chain storage, significantly increasing the overall latency.

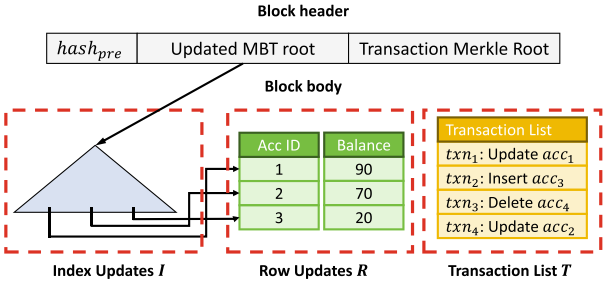


Fig. 1. Block Structure of Permissioned IoT blockchain.

Although existing approaches reduce on-chain storage overhead, they fail to address the fundamental performance bottleneck. Through systematic analysis, we identify that index construction during block creation is the primary performance bottleneck. Specifically, each node maintains a copy of the blockchain

ledger, which is stored as a chain of blocks that stores both transactions (for audit) and the latest ledger state (for query), as shown in Fig. 1. The blockchain also creates an index tree (e.g. Merkle B+ tree) for each specific column to speed up queries, with the tree root's hash stored in the block header. For instance, Ethereum [9] employs a Merkle Patricia Trie for account balances to accelerate balance queries. Since blocks are generated sequentially and all indices must be constructed before creating each block, index construction becomes a bottleneck, particularly in IoT scenarios with high transaction rates.

We validate the above analysis using a synthetic IoT workload on NeuChain [21]. Since the ledger of NeuChain is stored in a hash table, we modify its code to store the ledger in blocks and build a Merkle B+ Tree as the account index for the ledger (detailed setup in Sect. 4). As shown in Fig. 2, as the number of transactions per block increases from 100 to 1500, the index building time consumes 37.4% to 57.9% of the total block building time. This high proportion of index-building time is primarily attributed to the hashing operations, which are necessary for ensuring index verifiability [18]. The increasing proportion of index building overhead occurs because as the number of transactions per block increases, the number of rows that need to be modified increases nonlinearly (due to the $O(n\log(n))$ complexity of tree building operation), increasing the proportion of the index building overhead.

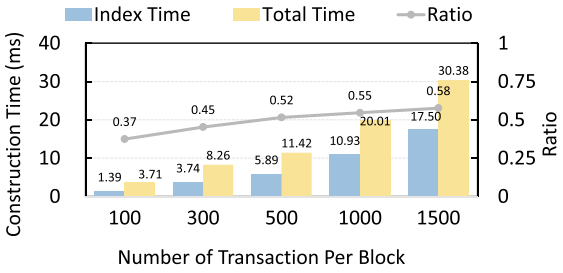


Fig. 2. Index construction time of total time with increasing block size.

To address these challenges, we propose DC-Storage, a fast permissioned IoT blockchain with a dual-chain architecture. Its core idea is to decouple indexing from block building and enable asynchronous index construction, thereby reducing the sequential block-building overhead in traditional IoT blockchains.

Decoupling index construction from block building is not trivial. First, transactions must be executed before returning results to clients. Since index construction (i.e., the index chain) is asynchronous to the blockchain (i.e., the data chain), when a node executes block b_{i+1}, the indexing of its previous block b_i may not yet be completed. Consequently, executing transactions based solely on index chains could result in stale reads, violating transaction consistency. To address this issue, we propose a double indexing technique. Given that transaction execution accesses the local ledger without requiring proofs (as correct nodes will

produce identical execution results), each node maintains a fast in-memory hash table to index the latest updates that have not yet been included in the index chain. During transaction execution, this hash table is queried before the index chain to prevent stale reads.

Second, since index chain construction is time-consuming, its building speed may be slower than that of the data chain. This could prevent clients from obtaining verifiable queries for the most recently modified rows. To address this issue, we introduce parallel index chain construction. When the index chain significantly lags behind the data chain, we fork the index chain to enable concurrent construction, thereby accelerating the index chain building process. This forking process is deterministic to prevent malicious attacks.

In summary, the contributions of this paper are as follows:

1. We propose a novel dual-chain architecture that decouples index construction from block building while ensuring transaction execution consistency and query accuracy through a double indexing technique.
2. We introduce parallel index chain construction to accelerate indexing while maintaining correctness through consensus.
3. We implemented DC-Storage, a fast permissioned IoT blockchain using the above techniques. Experiments show that DC-Storage outperforms its baseline with 5.03× higher throughput.

2 Background and Related Work

2.1 Authenticated Blockchain Index

Permissioned blockchains typically accelerate ledger queries and transaction execution by indexing specific columns (e.g., account). For example, a node can leverage account indices to efficiently query account attributes (e.g., name or balance) without traversing the entire ledger during transaction execution. Most permissioned blockchains, such as Hyperledger Fabric [3] and NeuChain [21], employ traditional database indexes, such as Log Structured Merge Tree [1] or in-memory hash tables. However, these indices lack authentication capabilities. Since Byzantine nodes can tamper with ledgers or query results, light clients cannot verify the authenticity of returned values as they only maintain block headers. In contrast, authenticated blockchain indices, such as Merkle B+ tree (MBT) and Merkle Patricia Trie [9], provide cryptographic proofs with query results. Light clients can thus verify returned values by validating these proofs using their block headers, preventing Byzantine attacks.

However, constructing authenticated indices [25] requires a higher overhead than traditional indices due to proof generation. Several research efforts have focused on accelerating index construction and query performance. vChain [31] extends verifiable query semantics and designs intra-block and inter-block index structures to improve query performance. However, it has poor index construction performance, making it unsuitable for high-throughput IoT scenarios. vChain+ [26] improves query performance over vChain at the cost of longer index

construction time. Experiments show that vChain+ takes over 250 ms to construct indices for 50 data objects. To support verifiable multi-dimensional aggregation queries, [36] proposes an accumulator-based index structure. However, the computational overhead of constructing these accumulators is also heavy.

While these prior works focus mainly on query performance, they do not address the high index construction overhead. To handle high-throughput IoT applications, we decouple index construction from block construction and propose parallel index construction.

2.2 Permissioned IoT Blockchain

Table 1. Comparison of permissioned IoT blockchains

Scheme	Off-Chain Storage	Data Availability	Data Sync Overhead
[15]	Cloud Server	Low	Medium
[28]	Database	Vary	LoW
[17]	IPFS	Medium	High
[34]	IPFS	Medium	High
[35]	IPFS	Medium	High
DC-Storage (ours)	Blockchain	High	Zero

Permissioned IoT blockchains typically adopt a hybrid architecture combining on-chain and off-chain storage. Table 1 summarizes the characteristics of existing solutions. [15] employs blockchain to record IoT parameters while utilizing cloud servers to store the data for data mining. However, cloud servers exhibit significantly lower availability than permissioned blockchains, making them susceptible to failures resulting in data loss and reduced availability. Moreover, synchronization overhead between cloud servers and blockchain networks also reduces performance. Similarly, [28] utilizes databases to store data. Data availability varies depending on the underlying database architecture (e.g., standalone or cloud databases). However, the high performance of databases helps reduce data synchronization overhead. To enhance availability, [17,34,35] leverage IPFS for off-chain storage, where the blockchain stores only the hash values of IPFS files containing IoT data. Nevertheless, IPFS also faces availability challenges [23]. Furthermore, IPFS incurs higher synchronization overhead than cloud server or database solutions. In contrast, DC-Storage stores data directly in the blockchain, achieving high availability with zero synchronization overhead.

2.3 Merkle B+ Tree

The Merkle B+ tree (MBT) extends the traditional B+ tree by incorporating cryptographic hash functions to enable authentication capabilities. As shown

in Fig. 3, leaf nodes store a set of key-value pairs, where each key represents an indexed attribute (e.g., account) and the corresponding value is the hash pointer of the row (e.g., hash(account||balance)) to ensure data integrity. Each non-leaf node contains multiple key-pointer pairs and a hash value. The hash value of a non-leaf node is computed by concatenating and hashing its children's hash values. This hierarchical hashing structure ensures that any modification to the indexed attributes or the underlying data will propagate up to the root hash, enabling efficient verification of query results.

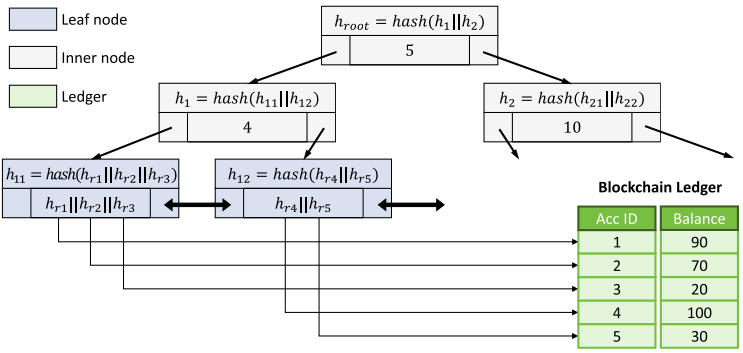


Fig. 3. The structure of Merkle B+ Tree (MBT).

As shown in Fig. 1, the block of IoT blockchains typically comprises three components: (1) a transaction list $\mathcal{T}$ for auditing purposes, (2) incremental row updates $\mathcal{R}$ containing the updated ledger, and (3) incremental index updates $\mathcal{I}$ constructed based on $\mathcal{R}$. While existing research has primarily focused on extending query semantics based on MBTs, they overlook the significant overhead of index construction. By storing all components within the same block, these approaches substantially reduce block generation efficiency.

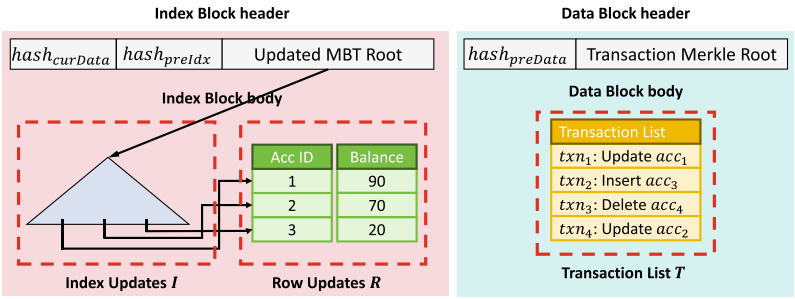


Fig. 4. Block Structure of DC-Storage.

Specifically, EthMB+ [13] introduces a tamper-proof data query model based on MBT to expand blockchain query primitives. EthMB+ employs an extractor to retrieve transaction records from Ethereum and insert them into the MBT. The MBT then processes user query requests and returns results through the query interface. Similarly, EBTree [29] employs MBT to facilitate real-time queries (including top-k, range, and exact-match searches) on Ethereum blockchain data, while utilizing RLP encoding to optimize storage efficiency.

ChainDB [14] adopts the hybrid storage architecture (Sect. 2.2) and utilizes MBTs to provide verifiable range queries. Similarly, [12] adopts the hybrid storage architecture and employs MBT as the index structure to support skyline and top-k queries. This approach addresses multi-dimensional data query challenges by constructing separate MBTs for different attributes of on-chain data. Furthermore, by incorporating bloom filters into each block to streamline the query verification process, the system significantly enhances verification efficiency for light nodes.

3 DC-Storage Design

To prevent index construction from becoming a performance bottleneck, we propose a dual-chain structure that decouples index construction from block generation. The dual-chain structure consists of a data chain containing transaction list $\mathcal{T}$ and an index chain comprising rows updates $\mathcal{R}$ and index updates $\mathcal{I}$, as shown in Fig. 4. Our key observation is that transaction commitment only requires the transaction to be stored on the blockchain (i.e., executing t, adding it to $\mathcal{T}$, and reaching consensus on the block containing $\mathcal{T}$), while $\mathcal{R}$ and $\mathcal{I}$ are constructed solely to support verifiable queries. Honest nodes can maintain lightweight, non-verifiable versions of $\mathcal{R}$ and $\mathcal{I}$ locally for transaction execution while constructing authenticated indices asynchronously to enhance performance. Since $\mathcal{R}$ and $\mathcal{I}$ are irrelevant to building $\mathcal{T}$, the data chain can be generated asynchronously from the index chain, thereby improving performance.

3.1 DC-Storage Overview

The dual-chain structure of DC-Storage is shown in Fig. 5. The nodes first achieve consensus (e.g., PBFT [11]) on the next data block containing a list of transactions from clients. Then, each node executes the block and generates an index block containing the ledger and index updates. To maintain the authenticity of the index chain, the nodes subsequently reach consensus on the index block. Since executing transactions in the $(i+1)$-th block requires looking up the index of the i-th block, each node maintains a local hash table that caches recent ledger updates to prevent transaction execution from being blocked by index chain generation. Finally, to enable verifiable queries for the latest updates, we introduce a concurrent index chain construction scheme.

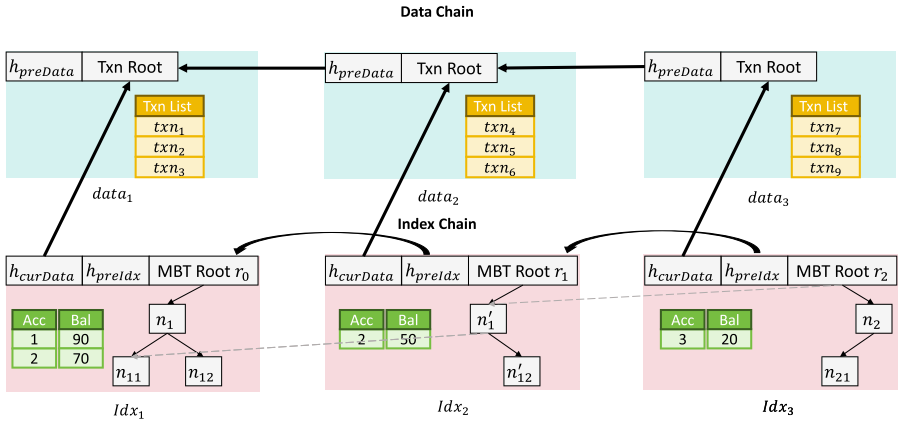


Fig. 5. The dual chain structure of DC-Storage.

3.2 Fast Decoupled Data Chain Construction

Since the construction of the index chain may be slower than that of the data chain, the latest updates may not be indexed immediately. This asynchronicity could lead to stale reads when querying through the index chain, violating transaction consistency. To address this issue, we propose a double indexing mechanism. By maintaining a local hash table for the latest updated rows, each node ensures that transaction execution always accesses the most recent version of the row. The workflow of transaction execution is shown in Algorithm 1.

Specifically, during transaction execution, read operations first query the local hash table. If a hit occurs, the latest row is returned directly. Otherwise, the node proceeds to query its index chain. Since each node maintains and trusts its local hash table and index chain, read operations do not require proof of authenticity during transaction execution. Write operations directly add the key-value pair to the row update map and the local hash table.

Delta-Based Hash Table Maintenance. The local hash table may consume excessive memory and impact query performance if it grows unbounded. To address this, we employ a delta approach for hash table maintenance, as shown in Algorithm 2. Let data_i and idx_i denote the i-th data block and index block.

Specifically, after executing the transaction list $\mathcal{T}$ from the latest data block data_i, the node appends the updates of the committed transactions to the row updates list $\mathcal{R}$ (which will subsequently be stored in the corresponding index block idx_i) and updates the hash table indices accordingly. Upon receiving an index block idx_j, the node iterates through the $\mathcal{R}$ within the block and removes each row update r from the hash table. To prevent incorrect deletion of the latest version, a row is removed from the hash table only when r exactly matches the most recent version r' indexed in the hash table.

An Example of Transaction Execution. Consider a ledger state where block data_1 and idx_1 have achieved consensus. The ledger contains a table with

Algorithm 1. Transaction Execution

Require: Txn list $\mathcal{T}$ in data block, local hash table lt, MBT of index chain mbt
Ensure: Execution results $\mathcal{E}$, row updates $\mathcal{R}$

$\mathcal{E} \leftarrow \emptyset$; $\mathcal{R} \leftarrow \emptyset$ ▷ Initialize empty result set and row updates set
for each transaction $t \in \mathcal{T}$ **do**
  $(e, r) \leftarrow$ EXECUTE(t) ▷ Get the execution result and updated rows of t
  $\mathcal{E} \leftarrow \mathcal{E} \cup \{e\}$ ▷ Add execution result e to result set $\mathcal{E}$
  $\mathcal{R} \leftarrow \mathcal{R} \cup r$ ▷ Add updated rows to row updates $\mathcal{R}$
return $(\mathcal{E}, \mathcal{R})$
function EXECUTE(t) ▷ Execute single transaction
  $r \leftarrow \emptyset$ ▷ Initialize empty row updates map
  for each operation $op \in t$.operations **do** ▷ Process read and write operations
    if op.type = READ **then**
      if op.key $\in lt$ **then** ▷ Query the local hash table first
        read(lt[op.key])
        continue
      if op.key $\in mbt$ **then** ▷ Then query the index chain
        read(mbt[op.key])
        continue
      rollback()
      return (ABORT, $\emptyset$) ▷ The read row does not exist, abort
    else if op.type = WRITE **then**
      r[op.key] $\leftarrow$ op.value ▷ Add to row updates
      lt[op.key] $\leftarrow$ op.value ▷ Update local hash table
  return (COMMIT, r)

Algorithm 2. Prune Hash Table

Require: Row updates $\mathcal{R}$ in index block, local hash table ht
Ensure: Pruned local hash table

for each row $r \in \mathcal{R}$ **do**
  if r.key $\notin ht$ **then continue** ▷ Skip if key does not exist in hash table
  if ht[r.key] $\neq r$.value **then**
    continue ▷ Skip if row was updated after this version
  ht.remove(r.key) ▷ Remove entry from hash table

account ID and balance columns, comprising two initial records: $\langle \text{acc}_1, 100 \rangle$ and $\langle \text{acc}_2, 50 \rangle$. A node receives and executes block data_2 containing transactions txn_4: $\text{acc}_1.\text{bal} - 10$ and txn_5: $\text{acc}_2.\text{bal} + 10$. During the read operations for acc_1 and acc_2, since the local hash table is empty, the node queries the index chain. The execution results in the updated row list $\mathcal{R}_2$ containing $\langle \text{acc}_1, 90 \rangle$ and $\langle \text{acc}_2, 70 \rangle$, which are subsequently cached in the local hash table.

Subsequently, the node executes data_3 containing transaction txn_6: $\text{acc}_1.\text{bal} - 20$. Following a similar process, this generates $\mathcal{R}_3$ containing $\langle \text{acc}_1, 70 \rangle$. At this point, when idx_2 (containing $\mathcal{R}_2$ and $\mathcal{I}_2$) reaches consensus and is received by the node, it can safely remove acc_2 from the hash table since its pointed value

matches that in $\mathcal{R}_2$. However, acc_1 must be retained in the hash table as it has been updated by txn_6 in data_3.

Recall that, during transaction execution, read operations only need to access the most recent version of rows and do not require proof of authenticity. Therefore, entries in the hash table can be safely removed when their pointed values match those in $\mathcal{R}$.

3.3 Concurrent Index Construction

To enable verifiable queries of the latest ledger state for clients, the index chain construction must keep pace with the data chain, even though transaction execution can access the latest state without relying on the index chain. To address this, we propose a concurrent index chain construction strategy.

The leader node monitors the construction progress of both chains. When the gap between the data chain and index chain exceeds a predefined threshold δ, DC-Chain triggers a forking mechanism to accelerate index chain construction. Specifically, DC-Chain forks a temporary side chain to construct the lagging index blocks. We only construct one side chain for each index to prevent unlimited fork attacks. Moreover, to prevent a malicious leader node, the fork request must achieve consensus before execution.

Query Processing. For query processing, a naive approach would be to merge the side chain back into the main chain upon completion, allowing clients to query a single, up-to-date index chain. However, merging two MBTs incurs substantial computational overhead and could block the construction of the main chain. Therefore, we propose a pragmatic solution where nodes query both the main chain and side chain simultaneously, selecting the most recent version of records and returning the corresponding records and proofs to clients.

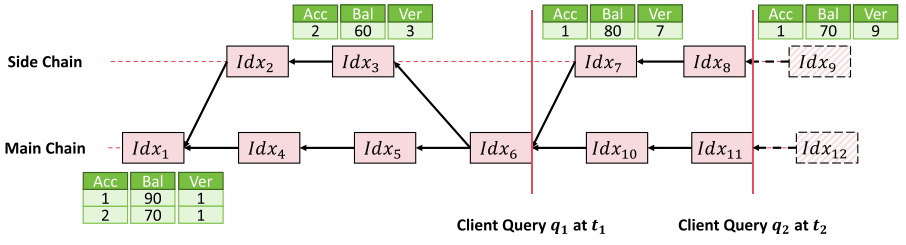


Fig. 6. Example of concurrent index construction.

To ensure consistency during side chain construction, queries can only be processed based on the last block with *all its predecessor blocks constructed*, rather than the latest blocks in either the main chain or side chain. As shown in Fig. 6, when block idx_9 and idx_{12} are being constructed concurrently in the index chain, queries must be based on block idx_8. This is necessary because block

idx_9 has not been constructed yet. Querying based on block idx_{11} would omit the ledger updates in block idx_9, resulting in incorrect results. In other words, a node can provide authenticated queries up to block height 8.

Since clients maintain lightweight block headers of the index chain, rows stored in both the side chain and main chain can be authenticated. Moreover, each row maintains a version number to prevent Byzantine nodes from returning stale versions. The most recent row is determined by comparing versions of the rows retrieved from both chains.

An Example. As shown in Fig. 6, the ledger initially contains two records stored in index block idx_1: $\langle acc_1, 90\rangle$ and $\langle acc_2, 70\rangle$. At time t_1, when a client queries acc_2, the node performs a search based on block height 6, retrieving $acc_2 = 60$ from the side chain and $acc_2 = 70$ from the main chain. By comparing the version numbers of these two results, the client determines the final value as $acc_2 = 60$, which represents the most recent update.

Since idx_9 is still under construction at time t_2, query q_2 for acc_1 cannot be done based on block height 11. Therefore, the node process q_2 is based on block height 8 (main chain up to idx_6 and side chain up to idx_8). The value of acc_1 is 80 in the side chain and 90 in the main chain. The client determines the final value as 80 by comparing their version numbers (1 and 7).

3.4 Security Analysis

This section analyzes the system's security mechanism when facing Byzantine behavior in the query processing and index construction.

Verifiable Query Processing. In the traditional model, users trust the query results returned by the service node by default, but in the Byzantine environment, the node may return tampered or incomplete data due to program failure, security vulnerabilities or malicious motives, which seriously threatens the security of the system. To this end, the system introduces the Verification Object (VO) as a structured proof of each query result, which is generated by an authenticated data structure (such as a Merkle tree or Merkle B tree). At the same time, the user maintains a trusted root hash value (Root Hash) locally as a light node. After receiving the query results and VO, the user can independently reconstruct and compare the root hash to verify the integrity and legitimacy of the results. This mechanism effectively prevents Byzantine behaviors such as tampering and omissions of service nodes, and enhances the credibility of queries.

BFT During Index Construction. The index construction process also faces the problem of malicious nodes forging index structures, such as tampering with data ranges or inserting wrong paths, leading to misleading queries. To prevent such attacks, the system introduces a consensus mechanism in the index proposal stage. Algorithms represented by PBFT require more than 2/3 of the nodes to reach a consensus before accepting the index submitted by a node. This mechanism ensures that even if there are Byzantine nodes, incorrect or maliciously constructed indexes cannot be adopted by the system, ensuring the correctness of the index structure and network consistency.

Limitations. The dual-chain architecture introduces additional consensus instances and chain structure maintenance costs, but in the context of the Internet of Things, which emphasizes real-time applications, this overhead is acceptable, and compared to the improvement in the overall system throughput, its benefits significantly outweigh the costs.

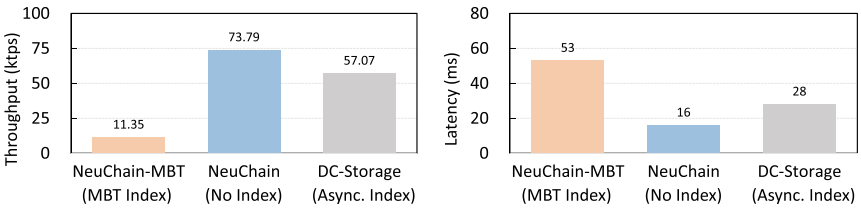
4 Experimental Evaluation

In this section, we evaluate the performance of DC-Storage.

Implementation. We implemented a blockchain prototype in C++ based on NeuChain [21]. We employ MBTs as authenticated indices. We utilize PBFT as the consensus protocol. To tolerate at most f Byzantine nodes, the total number of nodes must exceed $3f+1$. We use ED25519 for digital signatures and SHA256 for hash-based data integrity verification. To optimize system performance, we incorporate pipelining and batching techniques. Each node participates in two concurrent PBFT consensus processes, one for the data chain and the other for the index chain. For optimal performance, blocks are stored in memory.

Experimental Setup. We built a cluster with three nodes and a client on one virtual machine equipped with a 2.10GHz, 8-core Intel Xeon Silver 4110 CPU and 24GB of RAM. The nodes are connected through a local network to simulate a distributed environment. To evaluate the performance benefits of our approach, we compare our system against the following baselines:

- NeuChain: Serves as *an upper bound* for performance as it does not require authenticated index construction.
- NeuChain-MBT: A baseline implementation that only uses MBTs for index authentication.



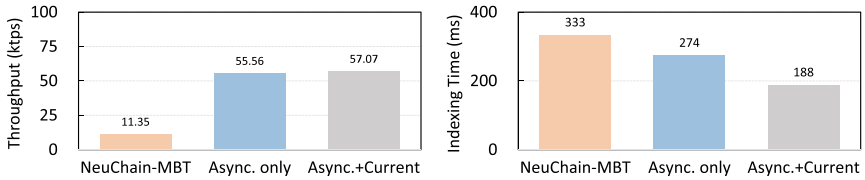
(a) Throughput (b) Latency

Fig. 7. Overall performance comparison.

Workload. Smallbank [10] workload simulates a simple banking application with basic account operations such as balance checking, deposits, and transfers. To simulate IoT workloads, we extend the ledger schema with a timestamp column and build an MBT index on it. The timestamp is updated whenever a transaction modifies the corresponding row. If not explicitly stated, the block size of all competitors is 1000.

4.1 Overall Performance

As shown in Fig. 7, DC-Storage achieves performance comparable to NeuChain while providing the same authenticated query as NeuChain-MBT. Specifically, DC-Storage achieves a 5.02× performance improvement compared to the baseline NeuChain-MBT. This is primarily attributed to the asynchronous index construction mechanism. By decoupling index construction from block creation, DC-Storage approaches the high performance of NeuChain while providing MBT-authenticated queries. The slightly lower performance compared to NeuChain is due to the additional computational overhead of maintaining a separate consensus instance for the index chain. Moreover, transaction execution incurs higher latency as it requires querying both the hash table and the MBT index, instead of directly accessing the state database in NeuChain.



(a) Throughput Breakdown (b) Index Construction Time

Fig. 8. Performance breakdown.

4.2 Performance Breakdown

We present a performance breakdown to analyze the individual impact of index decoupling and concurrent index construction. Figure 8a compares the performance of the baseline (NeuChain-MBT), DC-Storage with only decoupled indexing (Async. only), and DC-Storage with both optimizations (Async.+Concurrent). The results show that decoupling indexing significantly improves throughput, as index construction was the primary bottleneck (as discussed in Sect. 1). Concurrent index construction, however, does not impact throughput since transaction execution proceeds independently of index chain construction.

Figure 8b shows the time consumed to execute and index 15,000 transactions. Async. only is faster compared to NeuChain-MBT. DC-Storage further reduces the construction overhead through parallel MBT construction. When the number of accounts is 1,000,000 and the size is 44 bytes, the delta hash table consumes about 42 MB of DRAM.

4.3 Performance Varying Block Size

In this experiment, we vary the block size from 100 to 1000 transactions to observe the system performance under different workloads. As shown in Fig. 9,

the performance of all systems increases as the block size increases, due to different reasons. For NeuChain-MBT, the performance bottleneck lies in MBT construction. Larger batch sizes improve performance because batch insertion can avoid constructing unnecessary intermediate nodes during tree reorganization. NeuChain's performance is primarily limited by network round-trips required for PBFT consensus. therefore, larger block sizes reduce the coordination, leading to better performance. DC-Storage combines the advantages of both approaches and shows consistent performance improvements with increased block sizes.

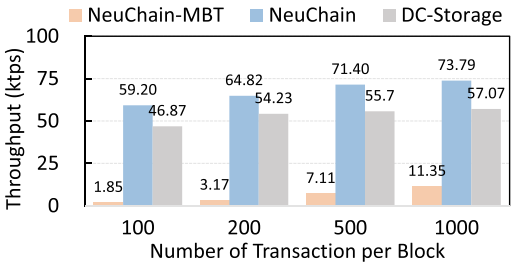


Fig. 9. Throughput varying block size.

4.4 Overhead of Authenticated Index Construction

To show indexing performance, we compare DC-Storage index chain construction performance with vChain [31] and vChain+ [26]. The results are shown in Table 2, where the baseline results are obtained from [26].

Table 2. ADS Construction Cost

DataSet	ETH		4SQ	
	T	S	T	S
vChain-acc1	260	125.6	390	28.1
vChain-acc2	40	126.1	40	29.3
vChain+	110	451.5	250	1209.1
DC-Storage	2	8.5	2	7.1

T: Index construction time(ms/block)
S: Index size(KB/block)

In the ETH [27] and 4SQ [33] workloads, DC-Storage achieves significantly faster index construction, requiring only 2 ms per block, while other solutions are substantially slower (e.g., vChain-acc1 takes 260-390 ms per block). This

superior performance is attributed to the concurrent index construction design of DC-Storage. Moreover, the MBT index consumes only 8.5 KB per block for ETH and 7.1 KB per block for 4SQ. This represents a 99.41% reduction in storage overhead compared to vChain+, which is about 1209.1 KB per block.

5 Future Work

Many IoT applications (e.g., temperature, humidity, and pressure) involve processing complex, high-dimensional data that spans multiple attributes (e.g., time, location, device ID, and sensor readings). While this article focuses on the design and implementation of single-column indexes, future work should address more advanced indexing strategies capable of supporting multi-attribute and multidimensional queries, which are increasingly common in real-world IoT scenarios.

To this end, a variety of multidimensional index structures can be considered. R-tree [16] and their variants (e.g., R*-trees [5] and R+-tree [22]) are widely used for spatial data indexing and support efficient range and nearest-neighbor queries. K-D Tree [6] is suitable for low-dimensional point data and is effective for applications involving small to moderate dimensions. For higher-dimensional data, X-tree [7] can be employed to mitigate the performance degradation caused by the "curse of dimensionality" by avoiding excessive node overlap. Similarly, UB-tree [4], which uses space-filling curves to linearize multidimensional data into a one-dimensional space, is another viable option for building multidimensional indexes atop traditional B+-trees.

Beyond performance, future research should also consider verifiable indexing mechanisms to ensure the integrity and trustworthiness of query results in distributed and potentially untrusted environments. By integrating Merkle trees into multidimensional index structures, it is possible to construct verifiable indexes that allow clients to verify the correctness and completeness of query results without trusting the data source. This is especially relevant in edge computing, cloud-assisted IoT systems, and blockchain-integrated platforms.

6 Conclusion

This paper presents DC-Storage, a fast permissioned IoT blockchain with a novel dual-chain architecture. By employing a double indexing technique, DC-Storage ensures transaction execution consistency and query accuracy. Additionally, DC-Storage introduces concurrent index chain construction, significantly accelerating index building. Experimental results show that DC-Storage achieves a 5.03× higher throughput than its baseline. In the future, we will continue to study how to generalize the proposed dual-chain approach to handle more advanced query types and more complex data models.

Acknowledgement. This work is supported by the National Natural Science Foundation of China (62372097), the National Social Science Foundation of China (21&ZD124), and the Fundamental Research Funds for the Central Universities (N2416003). Zeshun Peng is the corresponding author.

References

1. Leveldb (2024). http://code.google.com/p/Leveldb/
2. Tdengine user case (2024). https://www.taosdata.com/tdengine-user-cases
3. Androulaki, E., et al.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference, pp. 1–15 (2018)
4. Bayer, R., Markl, V.: The ub-tree: Performance of multidimensional range queries (1998)
5. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The r*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, pp. 322–331 (1990)
6. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975)
7. Berchtold, S., Keim, D.A., Kriegel, H.P.: The x-tree: An index structure for high-dimensional data (1996)
8. Bertino, E.: Data security and privacy in the iot. In: EDBT, vol. 2016, pp. 1–3 (2016)
9. Buterin, V., et al.: Ethereum white paper. GitHub Repository **1**(22–23), 5–7 (2013)
10. Cahill, M.J., Röhm, U., Fekete, A.D.: Serializable isolation for snapshot databases. ACM Trans. Database Syst. (TODS) **34**(4), 1–42 (2009)
11. Castro, M., Liskov, B., et al.: Practical byzantine fault tolerance. OsDI **99**, 173–186 (1999)
12. Ding, H., et al.: Efficient and verifiable skyline computation on blockchain system with merkle b+ tree index. In: 2024 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), pp. 2113–2120. IEEE (2024)
13. Du, P., Liu, Y., Li, Y., Yin, H.: Ethmb+: a tamper-proof data query model based on b+ tree and merkle tree. In: CCF China Blockchain Conference, pp. 49–59. Springer (2022). https://doi.org/10.1007/978-981-19-8877-6_4
14. Gao, J., et al.: Chaindb: ensuring integrity of querying off-chain data on blockchain. In: Proceedings of the 2022 5th International Conference on Blockchain Technology and Applications, pp. 175–181 (2022)
15. Goyat, R., et al.: Blockchain-based data storage with privacy and authentication in internet of things. IEEE Internet Things J. **9**(16), 14203–14215 (2020)
16. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, pp. 47–57 (1984)
17. Haque, E.U., Shah, A., Iqbal, J., Ullah, S.S., Alroobaea, R., Hussain, S.: A scalable blockchain based framework for efficient iot data management using lightweight consensus. Sci. Rep. **14**(1), 7841 (2024)
18. Jing, S., Zheng, X., Chen, Z.: Review and investigation of merkle tree's technical principles and related application fields. In: 2021 International Conference on Artificial Intelligence, Big Data and Algorithms (CAIBDA), pp. 86–90. IEEE (2021)
19. Khan, A.A., Laghari, A.A., Shaikh, Z.A., Dacko-Pikiewicz, Z., Kot, S.: Internet of things (iot) security with blockchain technology: a state-of-the-art review. IEEE Access **10**, 122679–122695 (2022)
20. Palankar, M.R., Iamnitchi, A., Ripeanu, M., Garfinkel, S.: Amazon s3 for science grids: a viable solution? In: Proceedings of the 2008 International Workshop on Data-Aware Distributed Computing, pp. 55–64 (2008)

21. Peng, Z., et al.: Neuchain: a fast permissioned blockchain system with deterministic ordering. Proc. VLDB Endowment **15**(11), 2585–2598 (2022)
22. Sellis, T., Roussopoulos, N., Faloutsos, C.: The r+-tree: A dynamic index for multi-dimensional objects (1987)
23. Shi, R., Cheng, R., Han, B., Cheng, Y., Chen, S.: A closer look into ipfs: accessibility, content, and performance. Proc. ACM Measurem. Anal. Comput. Syst. **8**(2), 1–31 (2024)
24. Statista, R.: Internet of things-number of connected devices worldwide 2015-2025. Statista Research Department. statista. com/statistics/471264/iot-number-of-connected-devices-worldwide (2019)
25. Tamassia, R.: authenticated data structures. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 2–5. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39658-1_2
26. Wang, H., Xu, C., Zhang, C., Xu, J., Peng, Z., Pei, J.: vchain+: optimizing verifiable blockchain boolean range queries. In: 2022 IEEE 38th International Conference on Data Engineering (ICDE), pp. 1927–1940. IEEE (2022)
27. Wood, G., et al.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper **151**(2014), 1–32 (2014)
28. Wu, H., Peng, Z., Guo, S., Yang, Y., Xiao, B.: Vql: efficient and verifiable cloud query services for blockchain systems. IEEE Trans. Parallel Distrib. Syst. **33**(6), 1393–1406 (2021)
29. XiaoJu, H., XueQing, G., ZhiGang, H., LiMei, Z., Kun, G.: Ebtree: a b-plus tree based index for ethereum blockchain data. In: Proceedings of the 2020 Asia Service Sciences and Software Engineering Conference, pp. 83–90 (2020)
30. Xiong, Z., Zhang, Y., Luong, N.C., Niyato, D., Wang, P., Guizani, N.: The best of both worlds: a general architecture for data management in blockchain-enabled internet-of-things. IEEE Network **34**(1), 166–173 (2020)
31. Xu, C., Zhang, C., Xu, J.: vchain: enabling verifiable boolean range queries over blockchain databases. In: Proceedings of the 2019 International Conference on Management of Data, pp. 141–158 (2019)
32. Yaga, D., Mell, P., Roby, N., Scarfone, K.: Blockchain technology overview. arXiv preprint arXiv:1906.11078 (2019)
33. Yang, D., Zhang, D., Qu, B.: Participatory cultural mapping based on collective behavior data in location-based social networks. ACM Trans. Intell. Syst. Technol. (TIST) **7**(3), 1–23 (2016)
34. Zheng, Q., Li, Y., Chen, P., Dong, X.: An innovative ipfs-based storage model for blockchain. In: 2018 IEEE/WIC/ACM International Conference on Web Intelligence (WI), pp. 704–708. IEEE (2018)
35. Zhou, E., et al.: Mstdb: qhybrid storage-empowered scalable semantic blockchain database. IEEE Trans. Knowl. Data Eng. **35**(8), 8228–8244 (2022)
36. Zhu, Y., Zhang, Z., Jin, C., Zhou, A.: Enabling generic verifiable aggregate query on blockchain systems. In: 2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS), pp. 456–465. IEEE (2020)