An Analysis of Ethereum Workloads from a Key-Value Storage Perspective

Yanjing Ren[†], Jia Zhao[†], Jingwei Li[‡], Patrick P. C. Lee[†]
[†]*The Chinese University of Hong Kong* [‡]*University of Electronic Science and Technology of China*

***Abstract*—Blockchains have revolutionized trust and transparency in distributed systems, yet their heavy reliance on key-value (KV) storage for managing immutable, rapidly growing data leads to performance bottlenecks due to I/O inefficiencies. In this paper, we analyze Ethereum's storage workload traces, with billions of KV operations, across four dimensions: storage overhead, KV operation distributions, read correlations, and update correlations. Our study reveals 11 key findings and provides suggestions on the design and optimization of blockchain storage.**

I. Introduction

Blockchains have been popularized by cryptocurrencies, such as Bitcoin [9] and Ethereum [1], and support applications including smart contracts [21], supply chain tracking [24], and decentralized finance (DeFi) [27]. To achieve high-performance data persistence, existing blockchains mainly use key-value (KV) stores (e.g., based on the log-structured merge tree (LSM) [23]) to organize data in the form of KV pairs for storage management [4], [17]. For example, Ethereum, the leading blockchain platform for smart contracts and DeFi, employs the LSM-based Pebble KV store [19] to manage blockchain data (e.g., accounts, contracts, and transactions) [4]. However, Ethereum's performance is constrained by I/O operations [20]. Due to the immutable, append-only nature of blockchain data and its unbounded data growth (about 200 GiB annually [2]), the I/O costs of KV storage significantly increase as the blockchain size expands over time. Note that LSM-tree compaction, a key background process that can remove stale data from LSM-based storage, cannot fundamentally mitigate this growing I/O burden due to the immutability of historical transactions in blockchains.

Existing studies have proposed optimizations to address KV storage inefficiencies in Ethereum, such as integrating authenticated data structures into storage backends [26], [32] and semantic-aware LSM layouts [12], [13], [20]. However, recent updates of the Ethereum client introduce new storage structures [3], [22], [28] that result in different KV workload characteristics, rendering existing optimizations less effective.

In this paper, we analyze 140-day Ethereum's storage workloads from a KV storage perspective. We characterize KV operations for 29 *classes* of KV pairs, where the classification is defined by Ethereum's storage semantics [5]. We focus on storage overhead, KV operation distribution, read correlations, and update correlations across the classes under different storage mechanisms (e.g., caching and snapshot acceleration). To this end, we present 11 key findings and provide insights into optimizing storage and cache management for blockchains.

Our analysis reveals substantially different workload characteristics across different classes of KV pairs. We highlight some key observations from our analysis: (i) 15 out of 29 classes have only one KV pair for maintaining system state, while five dominant classes account for over 99.2% of total KV pairs, with only an average size of 79.1 bytes; (ii) only three classes perform scans (a.k.a. range queries), while deletions are common; and (iii) read and update correlations exist within the same class and across different classes. Our analysis suggests that LSM KV stores are sub-optimal for Ethereum due to excessive deletions and rare scans. Also, the current caching design in Ethereum underperforms due to its lack of read correlation awareness. This suggests the necessity of a hybrid KV store design that incorporates specialized data structures based on data semantics and access patterns, efficient deletion mechanisms, and correlation-aware storage management, so as to achieve high performance.

We have released our scripts for trace collection and analysis at https://github.com/adslabcuhk/geth_analysis. While we do not release the raw traces due to their large volume, the raw traces can be obtained from the public Ethereum blockchain using our provided scripts.

II. Background and Related Work

In this section, we provide an overview of data management in blockchain systems, using Ethereum as a primary case study. We also review related work and distinguish our study from existing KV storage benchmarking efforts.

A. Blockchain Basics

Blockchain overview. Figure 1 shows the logical data organization based on Ethereum's blockchain. A blockchain is a decentralized, immutable ledger maintained by a distributed network of nodes that synchronize states via consensus protocols, such as Proof of Work (PoW) [7] or Proof of Stake (PoS) [6]. It supports smart contracts (i.e., self-executing programs that automate agreement terms) and maintains a dynamically updated *world state*, which maps accounts to account information such as account balances and contract data. It operates on *transactions*, which encode asset transfers or contract executions. Transactions are validated by nodes, grouped into *blocks*, and cryptographically linked into a verifiable chain starting from the *genesis block* (i.e., the first block of the blockchain). Each block also includes transaction receipts that record execution outcomes.

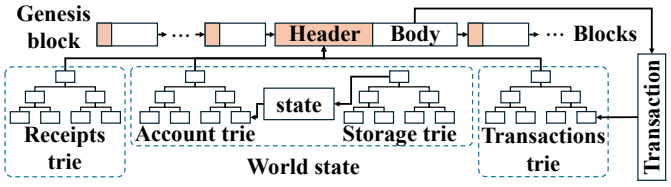


Figure 1: Logical data organization of Ethereum's blockchain.

Each blockchain node employs *authenticated data structures (ADSs)* to ensure data integrity in local storage. For example, Ethereum uses a *Merkle Patricia Tree (MPT)* [30], which combines Merkle Tree's cryptographic proofs with efficient Patricia Trie storage. It organizes accounts, contract storage, transactions, and receipts in different MPTs. However, frequent MPT modifications (e.g., world state updates) and deep traversals (e.g., for account lookup and proof generation) significantly increase I/O overhead and raise scalability concerns.

Data management in Ethereum. Ethereum's official client (i.e., an application for transaction processing and blockchain data management) is *go-ethereum (Geth)* [4]. Geth runs on every node and organizes blockchain data as KV pairs in a high-performance database (Pebble [19] by default). The header and body of each block are separately stored as different KV pairs. The world state (including accounts and contract storage) is organized in multiple MPTs, and each MPT node is also represented as a KV pair. Geth further keeps auxiliary metadata (e.g., chain configurations and journaling) in the same KV store for system maintenance. Keys are prefixed with type-specific identifiers (e.g., 'h' for block headers and 'b' for block bodies) followed by contextual metadata (e.g., block heights, block hashes, or transaction hashes), while values are compactly serialized via Recursive Length Prefix encoding for storage efficiency.

Evolution of Geth. Geth's implementation has been evolving over the past years to enhance performance. Earlier Geth versions store MPT nodes as hash-keyed KV pairs, but this introduces redundant entries and frequent recomputations during trie updates. Thus, Geth moves to a path-based storage model [22], which encodes trie traversal paths directly into keys via a prefix-aware scheme. This significantly reduces redundant entries and recomputations, thereby improving retrieval performance and storage efficiency.

Geth caches KV pairs using the least-recently used (LRU) policy in multiple caches, each for a specific class of KV pairs. In addition to LRU caching, Geth introduces *snapshot acceleration* [28], which maintains a real-time, flat snapshot of the current world state to avoid frequent MPT traversals during account lookups (e.g., from up to 64 requests per lookup [12] to a single request). It also introduces the *freezer database* [3], which offloads stale blockchain segments (e.g., blocks beyond finality thresholds) into immutable flat files, so as to reduce KV pairs in the KV store. However, moving data from the KV store to the freezer database introduces lifecycle management complexity and pruning overhead. To further reduce performance and storage overhead, one proposal (not yet implemented at the time of writing) is to fully prune historical data older than one year [18].

Synchronization and KV operations. Geth's synchronization process generates significant KV operations. It follows a two-step approach. First, each node downloads unprocessed blocks and associated metadata from peer nodes and stores them in its local KV store. Second, each node reads KV pairs (including the world state, blocks, and other metadata) from the KV store to verify downloaded blocks by processing their transactions. The verification process involves frequent updates of the world state and metadata, thereby incurring substantial KV operations.

Geth supports three types of nodes: (i) *full nodes*, (ii) *archive nodes*, and (iii) *light nodes*. Full nodes are the backbone of Ethereum's network and maintain a complete blockchain copy and the latest world state. They also play a crucial role in ensuring the integrity and availability of the blockchain network by validating and propagating transactions. Archive nodes store the entire historical state of the blockchain, but are rarely used in practice [14], while light nodes download and verify minimal data on demand.

Geth supports two synchronization modes for full nodes: (i) *full synchronization* and (ii) *snap synchronization*. The synchronization mode determines how a node processes blocks and maintains the world state, thereby directly impacting KV storage workload. Full synchronization processes every block from the genesis block, while snap synchronization starts from a relatively recent block and performs synchronization to the head of the chain. A node that starts with snap synchronization will switch to block-by-block full synchronization upon reaching the head of the chain. In this paper, we focus on full synchronization in full nodes, so as to comprehensively analyze Ethereum's KV storage workload during transaction processing.

Applicability beyond Geth. Our study focuses on Geth, which is the most widely used Ethereum client and constitutes over 50% of all deployed clients [15]. Other clients (e.g., Besu [17] and Erigon [29]) are also implemented based on the Ethereum Yellow Paper [30] and exhibit similar KV storage semantics. Thus, we expect that these clients share comparable workload characteristics with Geth. A detailed comparative analysis across clients is posed as future work.

B. Related Work

KV storage optimization for Ethereum. Prior studies have proposed various optimizations for Ethereum's storage management. mLSM [26] combines Merkle trees and LSM trees to reduce read and write amplifications while maintaining authentication efficiency. Block-LSM [13] optimizes LSM layouts for blockchain workloads with semantic-aware data grouping and block-aligned compaction. MoltDB [20] notes that 98% of transaction processing time is spent on I/O operations, primarily due to ancient state data, and proposes segregation for ancient state data. ChainKV [12] separates the storage of state and non-state data and applies prefix-based MPT-to-KV transformation. COLE [32] introduces a new ADS with column-based learned storage to improve performance

and storage efficiency. SlimArchive [16] reduces the storage requirements for Ethereum archive nodes by removing duplicate data based on a hash-based storage model.

However, Geth's recent updates [4] have reshaped its storage management, rendering existing optimizations [12], [13], [20], [26], [32] less effective. For example, the path-based storage model [22] is incompatible with mLSM [26] and Block-LSM [13]. Specifically, mLSM combines both MPT and LSM structures and alters state data management and MPT proof generation, and such a combination is incompatible with the path-based model's traversal path encoding and MPT-based proof generation. Block-LSM [13] adds block-based prefixes to the keys of KV pairs, but this prevents the path-based storage model from eliminating redundant KV pairs. Also, the path-based model can achieve comparable storage savings to COLE [32], but without requiring ADS modifications as in COLE.

Despite various advancements, existing approaches often rely on coarse-grained workload characteristics and do not address blockchain-specific access patterns of different data types in their designs. Their dependence on LSM KV stores also introduces high I/O overhead from compaction and deletions. We will evaluate the impact of various optimization strategies on I/O characteristics, including caching and snapshot acceleration, via trace analysis, so as to provide insights into the future design of storage management for blockchains.

Characterization of KV workloads. Prior measurement studies have characterized KV workloads using real-world traces from deployed KV stores. For instance, Atikoglu et al. [8] analyze Facebook's Memcached workloads at massive scale. Rabl et al. [25] evaluate six modern KV stores for application performance monitoring. Chen et al. [11] analyze the performance of interactive cloud services, including Memcached and NGINX. Yang et al. [31] analyze production environment traces from Twitter's memory caches. Cao et al. [10] analyze the KV workloads of RocksDB across three production use cases at Facebook. These workload studies highlight that the characteristics of real-world KV workloads vary significantly across different applications (e.g., social graphs, AI, caching), thereby necessitating tailored KV store designs instead of relying on a single solution.

Our work provides the first quantitative characterization of Ethereum's KV workloads. It is distinct from the existing KV workload studies due to its Ethereum-specific nature. In particular, our analysis reveals some unique findings. For example, scans are rare in Ethereum workloads and occur in only three of 29 classes with a negligible frequency (§IV-B). Deletions are significant, driven by application requirements and Geth's storage evolution (e.g., moving KV pairs to the freezer database) (§IV-B), and such significant deletions are uncommon in practical KV workloads. In addition, read and update operations exhibit correlations within and across classes, and such correlation patterns are not found in existing KV workload studies. These findings provide insights for redesigning KV stores and caches specifically for Ethereum and general blockchain systems.

III. Traces

A. Trace Collection

We have collected KV operation traces from Ethereum's storage backend by modifying a Geth client to log information during synchronization. We capture traces from the KV store interface, so as to include all KV operations issued by Geth. The traces cover KV workloads for 1 M blocks (blocks 20.5 M to 21.5 M) from August 10, 2024 to December 28, 2024 (140 days in total). The traces provide a representative and up-to-date view of Ethereum's workloads (Ethereum has 21.7 M blocks as of February 1, 2025). Our trace collection follows Geth's default configuration, including the Pebble KV store [19] and its path-based storage model [22], and is independent of the underlying storage hardware used for trace collection. We analyze two traces: *CacheTrace* (with a total size of 1.3 TiB), captured with caching and snapshot acceleration enabled, and *BareTrace* (with a total size of 4.1 TiB), captured without these features. In the current Geth implementation, snapshot acceleration is a dependent feature of caching and is enabled or disabled with caching. The total cache size is 1 GiB by default, shared by multiple caches in Geth.

B. Basic Statistics

The traces capture KV operations from a *full node* (i.e., a node that stores and maintains all blocks and the latest world state) that processes 1 M blocks in full synchronization mode. BareTrace contains 9.16 B KV operations with 601.3 M unique keys, accounting for 24.6% of all 2.44 B keys in the KV store after synchronizing 21.5 M blocks. On the other hand, CacheTrace contains 2.86 B KV operations with 755.2 M unique keys, accounting for 19.2% of all 3.94 B keys in the KV store. It contains significantly more unique keys and KV pairs than BareTrace due to snapshot acceleration, which introduces additional KV pairs by snapshotting the latest world state. On the other hand, CacheTrace has significantly fewer KV operations than BareTrace, as its caching mechanism reduces the number of KV operations issued to the KV store interface.

Based on Geth's storage semantics [5], each KV pair is assigned a distinct prefix that corresponds to a specific class. We identify 29 classes, as shown in Table I, where the classification is determined by type-specific prefixes. We focus on five operation types: reads, writes, updates, deletes, and scans. Note that Geth does not distinguish between writes and updates; we classify a write as an update if it is issued to an existing key in the KV store.

Geth supports a broad range of KV classes, but only 29 KV classes are actually used by Geth from our traces, which represent a recent view of Ethereum's workloads. As Geth evolves, our analysis can be readily extended to accommodate more classes. We do not expect significant workload changes in future Ethereum as it mainly performs transaction processing and account updates.

IV. Findings

In this section, we present 11 key findings derived from our trace analysis. We group the findings into four categories: (i)

Class	Description	# KV pairs (%)	Key size	Value size
TrieNodeStorage	Storage nodes in the storage trie	1656.6 M (42.1%)	37.6±0.0001	70.3±0.003
SnapshotStorage	Flat storage data for the smart contract of the current world state	1222.3 M (31.1%)	65	12.5±0.0006
TxLookup	Transaction and receipt lookup metadata	386.2 M (9.81%)	33	4
TrieNodeAccount	Account nodes in the state trie	367.0 M (9.32%)	18.5±0.0001	115.7±0.006
SnapshotAccount	Store flat account nodes of current world state	269.4 M (6.84%)	33	15.9±0.002
HeaderNumber	Block hash to block ID mappings	21.5 M (0.55%)	33	8
BloomBits	Bloom filter bits for log search	10.7 M (0.27%)	43	398.0±0.11
Code	Smart contract bytecode storage	1.47 M (0.04%)	33	6732.7±10.0
SkeletonHeader	Block headers for skeleton synchronization, which downloads block headers and fills in block data as needed	0.55 M (0.01%)	9	609.7±0.02
BlackHeader	Block header data (e.g., parent block, timestamp, gas limit)	0.27 M (0.007%)	31.0±0.06	217.7±1.05
BlockReceipts	Transaction receipts for each block	0.09 M (0.002%)	41	75910.7±346.5
BlockBody	Block body data (e.g., transactions and uncle blocks)	0.09 M (0.002%)	41	79348.1±340.2
StateID	World state version identifier	0.09 M (0.002%)	33	8
BloomBitsIndex	Data table of a chain indexer for progress tracking	0.005 M (0.0001%)	15.0±0.003	32.0±0.009
Ethereum-genesis	Genesis state for the database	1 (-)	49	710909
SnapshotJournal	In-memory differential layers across system restarts within a snapshot journal	1 (-)	15	8369153
Ethereum-config	Ethereum network configuration	1 (-)	48	603
LastStateID	StateID of the latest stored world state	1 (-)	11	8
Unclean-shutdown	List of local crashes	1 (-)	16	33
SnapshotGenerator	Snapshot generation marker across restarts	1 (-)	17	7
TrieJournal	In-memory trie node layers across restarts	1 (-)	11	352749130
DatabaseVersion	Database schema version	1 (-)	15	1
LastBlock	Latest known full block's hash	1 (-)	9	32
SnapshotRoot	Last snapshot's hash	1 (-)	12	32
SkeletonSyncStatus	Skeleton synchronization status across restarts	1 (-)	18	146
LastHeader	Latest known block header's hash	1 (-)	10	32
SnapshotRecovery	Snapshot recovery marker across restarts	1 (-)	16	8
TransactionIndexTail	Oldest block whose transactions are indexed	1 (-)	20	8
LastFast	Latest incomplete block for fast synchronization (now replaced by snap synchronization)	1 (-)	8	32

Table I: Overview of classes identified based on Geth's storage semantics. For each class, we report the number of KV pairs in units of one million (M) and its percentage over all KV pairs in CacheTrace, except for those with only one KV pair. We also report the average key and value sizes in bytes. For classes with variable key and value sizes, we include 95% confidence intervals (under normal distribution).

KV storage management, (ii) KV operation distribution, (iii) read correlation, and (iv) update correlation.

A. KV Storage Management

We first analyze the KV size distribution across different classes to evaluate blockchain storage overhead. We extract all KV pairs, including those from snapshot acceleration, from the KV store after CacheTrace is captured (recall that BareTrace has snapshot acceleration disabled). Note that some KV pairs in the KV store are not accessed; that is, they are written during the synchronization of the first 20.5 M blocks but are then never accessed during trace collection.

Finding 1. *Five classes of KV pairs dominate KV storage.*

Table I shows the statistics of each class. Across the 29 classes, five classes dominate KV storage and altogether contribute to over 99.2% of all KV pairs. The five classes are grouped into two categories: (i) snapshot acceleration, including SnapshotAccount and SnapshotStorage, which store flat account and contract data synchronized with the latest world-state, respectively; and (ii) Ethereum data, including TrieNodeAccount, TrieNodeStorage, and TxLookup, which represent account MPT nodes, contract storage MPT nodes, and transaction lookup indexes, respectively. All of them, except TxLookup, are the world state data. Since Geth migrates old block data (e.g., BlockHeader, BlockBody, and BlockReceipts) to the freezer database (§II-A), the KV store only stores a

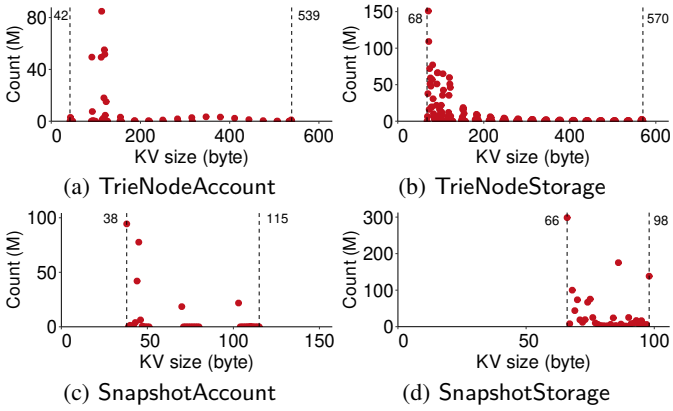


Figure 2: KV size distribution. Since the distributions of KV size are not continuous, we use scatter points to represent each distribution. The x-axis represents the size of KV pairs in bytes. We also mark the actual range of the KV sizes for clarity.

small number of recent blocks, while all world state data is kept in the KV store. Thus, the world state dominates KV storage. In addition, with a large number of transactions in each block, TxLookup also accounts for 9.8% of all KV pairs.

For the remaining 24 classes, 15 of them have only one KV pair each, primarily for Geth's system maintenance (e.g., system configurations and journals). The other nine classes manage blockchain-related data (e.g., blocks, receipts, smart contract code, and chain state information) to facilitate efficient block validation and smart contract execution.

Finding 2. *KV sizes (per KV pair) vary across classes.*

Table I shows that the five dominant classes (see Finding 1) exhibit small KV sizes, averaging 79.1 bytes. Figure 2 shows the KV size distributions for four of the dominant classes (TxLookup is excluded due to its fixed size of 33 bytes). TrieNodeAccount (Figure 2(a)) and TrieNodeStorage (Figure 2(b)) peak at small sizes (i.e., 113 bytes and 71 bytes, respectively), but have long tails extending to 539 bytes and 570 bytes, respectively. In contrast, SnapshotAccount (Figure 2(c)) and SnapshotStorage (Figure 2(d)) show more uniform distributions, with peaks at three distinct sizes (38 bytes, 70 bytes, 103 bytes for SnapshotAccount, and 66 bytes, 86 bytes, 98 bytes for SnapshotStorage). Their maximum sizes are smaller than those of TrieNodeAccount and TrieNodeStorage.

For the nine blockchain-related classes, Code (smart contract code), BlockBody (block bodies), and BlockReceipts (transaction execution receipts) have significantly larger KV sizes, averaging 6.61 KiB, 77.5 KiB, and 74.2 KiB, respectively. The remaining six classes have an average KV size of 239.6 bytes. For the 15 system maintenance classes, most have small KV sizes (averaging 95.2 bytes), except for TrieJournal (world state journaling), SnapshotJournal (snapshot journaling), and Ethereum-genesis (Ethereum genesis data), which have large average KV sizes of 336.4 MiB, 7.98 MiB, and 0.68 MiB, respectively. In summary, small KV pairs dominate KV storage, while large KV pairs (over 1 KiB) account for only 0.04% of all KV pairs.

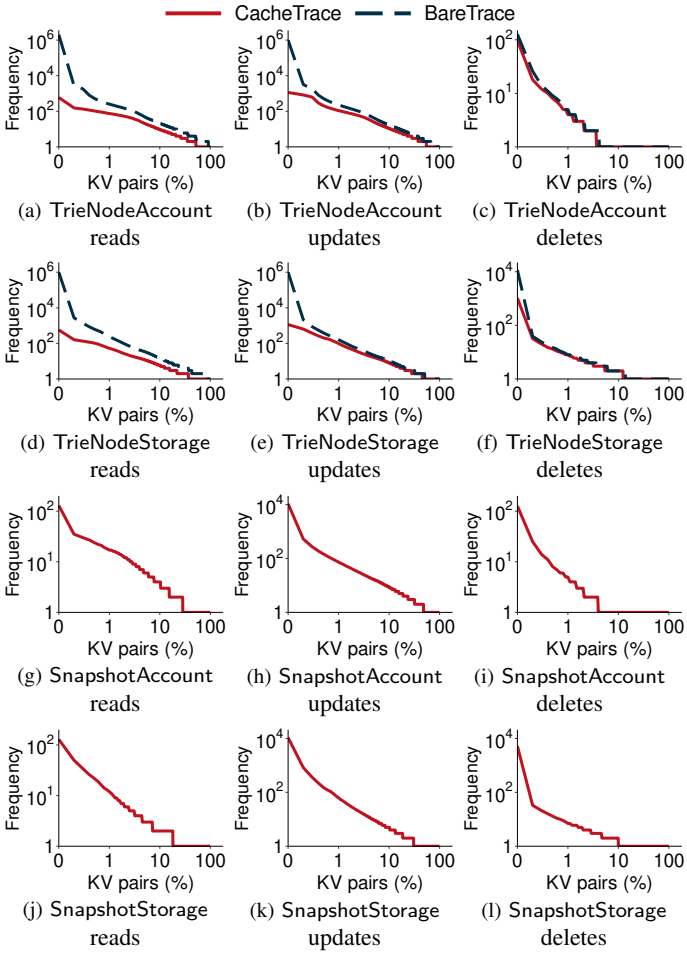


Figure 3: Frequency distribution of KV operations on world state. Both axes are in log scale.

B. KV Operation Distribution

We analyze the distribution of KV operations across each class of KV pairs in different traces, as shown in Table II and Table III for CacheTrace and BareTrace, respectively.

Finding 3. *Most KV pairs are rarely or never read.*

We first examine the read ratios of KV pairs. Table IV further shows the read ratio of KV pairs in each class (measured by the fraction of KV pairs that are read in each class) in CacheTrace and BareTrace. Here, we exclude TxLookup, which has zero reads (Tables II and III); the reason is that our trace collection captures KV operation traces during Geth's synchronization (§III-A), in which Geth does not serve any application and issue queries to transactions. From Table IV, TrieNodeAccount and TrieNodeStorage have low read ratios in both traces (e.g., up to 14.7% of TrieNodeAccount KV pairs are read in BareTrace); the read ratios of SnapshotAccount and SnapshotStorage are even lower. Figure 3 shows that most KV pairs read in CacheTrace are read only once: among all KV pairs being read, 71.5%, 81.8%, 48.1%, and 63.1% of them are read just once in SnapshotAccount, SnapshotStorage, TrieNodeAccount, and TrieNodeStorage, respectively. In BareTrace, among all KV pairs being read, 8.40% and 15.2% of them are read only

Class	% of all KV operations	Writes (%)	Updates (%)	Reads (%)	Scans (%)	Deletes(%)
TrieNodeStorage	38.5	8.51	50.9	35.7	-	4.87
SnapshotStorage	17.9	14.3	32.6	45.0	0.002	8.09
TxLookup	11.1	52.0	0.0004	-	-	48.0
TrieNodeAccount	23.2	2.32	59.7	38.0	-	0.003
SnapshotAccount	7.48	7.20	64.9	27.9	0.000001	0.006
HeaderNumber	0.05	74.9	0.0007	25.1	-	-
BloomBits	0.02	97.8	-	2.20	-	-
Code	0.41	1.11	11.7	87.2	-	-
SkeletonHeader	0.05	16.4	0.40	83.2	-	-
BlockHeader	0.62	16.9	0.0002	60.6	5.63	16.9
BlockReceipts	0.11	32.1	0.0003	35.8	-	32.1
BlockBody	0.14	24.2	0.0002	51.6	-	24.2
StateID	0.07	50.0	0.0005	-	-	50.0
BloomBitsIndex	0.002	0.55	0.55	98.9	-	-
LastStateID	0.03	-	0.11	99.9	-	-
Unclean-shutdown	0.00004	-	50.0	50.0	-	-
LastBlock	0.04	-	99.7	0.28	-	-
SnapshotGenerator	0.0004	-	100.0	-	-	-
SnapshotRoot	0.0007	-	50.0	-	-	50.0
SkeletonSyncStatus	0.009	-	99.8	0.19	-	-
LastHeader	0.03	-	100.0	-	-	-
TransactionIndexTail	0.00009	-	59.9	40.1	-	-
LastFast	0.03	-	100.0	-	-	-

Table II: Statistics of KV operation distribution in CacheTrace. We show the percentages of writes, updates, reads, scans, and deletes in each class.

Class	% of all KV operations	Writes (%)	Updates (%)	Reads (%)	Scans (%)	Deletes(%)
TrieNodeStorage	57.3	1.96	36.8	60.2	-	1.10
TxLookup	3.46	52.0	0.0004	-	-	48.0
TrieNodeAccount	38.6	0.62	58.1	41.3	-	0.0005
HeaderNumber	0.03	41.3	0.0004	58.7	-	-
BloomBits	0.006	94.3	-	5.75	-	-
Code	0.13	1.11	11.7	87.2	-	-
SkeletonHeader	0.05	4.57	1.45	75.6	-	18.4
BlockHeader	0.20	16.4	0.0002	61.7	5.47	16.4
BlockReceipts	0.03	32.1	0.0003	35.9	-	32.0
BlockBody	0.05	23.2	0.0002	53.5	-	23.2
StateID	0.02	50.0	0.0005	-	-	50.0
BloomBitsIndex	0.002	0.15	0.15	99.7	-	-
LastStateID	0.03	-	33.3	66.7	-	-
Unclean-shutdown	0.00005	-	50.0	50.0	-	-
LastBlock	0.01	-	98.9	1.05	-	-
SkeletonSyncStatus	0.003	1.51	97.7	0.75	-	-
LastHeader	0.01	-	100.0	-	-	-
TransactionIndexTail	0.00003	-	55.3	44.7	-	-
LastFast	0.01	-	100.0	-	-	-

Table III: Statistics of KV operation distribution in BareTrace. We show the percentages of writes, updates, reads, scans, and deletes in each class.

once in TrieNodeAccount and TrieNodeStorage, respectively. As most KV pairs are either never read or read only once, relocating such KV pairs to separate storage can reduce the KV store size and indexing overhead.

Class	BareTrace (%)	CacheTrace (%)
SnapshotAccount	-	11.0
SnapshotStorage	-	12.0
TrieNodeAccount	14.7	13.0
TrieNodeStorage	8.34	6.59

Table IV: Read ratios of KV pairs in both traces.

Finding 4. *Scans are rare in Ethereum.*

Scans for key ranges are very rare in Ethereum. As shown in Table II, scans occur only in SnapshotAccount, SnapshotStorage, and BlockHeader, and account for a negligible percentage of KV operations. SnapshotAccount has only two scans (less than 10^{-6}% of its KV operations), while scans in SnapshotStorage account for only 0.002% of its KV operations. In contrast, BlockHeader shows more scans, with 5.63% in CacheTrace and 5.47% in BareTrace. Scans are rare mainly because most KV operations in Ethereum come from transaction processing (§II-A), which only modifies two accounts instead of scanning a range of neighboring accounts. For smart contracts, scans are also uncommon. This finding suggests that a hybrid KV store design may be useful to avoid order maintenance for all classes of KV pairs.

Finding 5. *Deletions are significant, with some keys repeatedly deleted and reinserted.*

Tables II and III show that deletions are common in TxLookup and BlockHeader, accounting for 48.0% and 16.9% of KV operations in CacheTrace and 48.0% and 16.4% of KV operations in BareTrace, respectively. TxLookup has a high deletion rate since Ethereum indexes only recent transactions, while BlockHeader has a high deletion rate as Ethereum performs data pruning to relocate old blocks to the freezer database. In contrast, SnapshotStorage, TrieNodeStorage, SnapshotAccount, and TrieNodeAccount have low deletion rates, as Ethereum removes only obsolete MPT nodes, which are limited in the path-based storage model.

Furthermore, Figures 3(c), 3(f), 3(i), and 3(l) show that some keys have a deletion frequency greater than one (i.e., they are repeatedly deleted and reinserted) due to frequent modifications of MPTs during transaction processing. In LSM KV stores, deletions write tombstone entries that persist until compaction. Thus, log-based storage with batched deletions can effectively reduce redundant operations and unnecessary compaction; alternatively, hash-based storage with in-place deletions can also improve KV store performance.

Finding 6. *Caching has limited effectiveness for medium-frequency KV pairs.*

Caching and snapshot acceleration in CacheTrace significantly reduce the total number of reads, from 4.65 B in BareTrace to 0.96 B in CacheTrace. However, the read frequency distribution of KV pairs remains largely unchanged, except for the most frequently read KV pairs. Figures 3(a) and 3(d) show that caching and snapshot acceleration effectively reduce reads to the top 0.1% most-read KV pairs by 99.97% for TrieNodeAccount and 99.94% for TrieNodeStorage. For medium-frequency KV pairs (e.g., read 10 to 100 times), the reduction is less pronounced (e.g., 50.0-64.4% for TrieNodeAccount). Such differences indicate that while caching is effective for high-frequency KV pairs, it is less effective for medium-frequency KV pairs, as the simple LRU cache eviction strategy (§II-A) does not take into account the specific workload characteristics of different classes of KV pairs. This suggests that a unified cache design that addresses Ethereum's storage semantics and workload patterns may be useful to further alleviate I/O bottlenecks in Ethereum.

Finding 7. *Snapshot acceleration reduces reads and writes to the world state, but incurs high storage overhead.*

Snapshot acceleration, in conjunction with caching, reduces reads to TrieNodeAccount and TrieNodeStorage by 82.7% and 87.5% in CacheTrace compared to BareTrace, respectively. Although it incurs extra reads to SnapshotAccount and SnapshotStorage, the overall reduction in reads related to the world state (i.e., TrieNodeAccount, TrieNodeStorage, SnapshotAccount, and SnapshotStorage) is 79.7%. Snapshot acceleration also reduces writes (including updates) to TrieNodeAccount and TrieNodeStorage. The total number of writes (including updates) for TrieNodeAccount, TrieNodeStorage, SnapshotAccount, and SnapshotStorage drops by 64.2% from 4.11 B in BareTrace to 1.47 B in CacheTrace. However, snapshot acceleration incurs high storage overhead, with the number of KV pairs in the KV store increasing by 61.5% from 2.44 B in BareTrace to 3.94 B in CacheTrace.

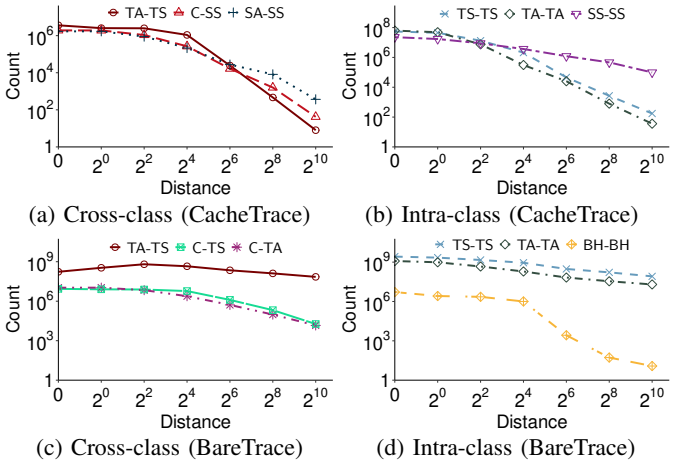
C. Read Correlations

Geth batches and flushes writes (updates) to the KV store at the end of verifying each block, while reads are triggered on-demand during transaction processing and the read performance can be degraded by random I/O patterns. Thus, it is critical to understand read behavior. In BareTrace and CacheTrace, 50.8% and 33.7% of all KV operations are reads, respectively. Here, we focus on studying read correlations.

We study the significance of read correlations using the frequency of *correlated reads*. We define a *distance* metric as the number of reads separating two specific reads (e.g., a zero distance means adjacent reads). Correlated reads are defined as unordered pairs of reads for two KV pairs, each from a specified (same or different) class, and are counted only if they occur at least twice for a given distance across the whole trace. To calculate the number of correlated reads between two classes (say, classes A and B, where A and B can be the same or different) under a specific distance d, we scan the entire trace and count the occurrences of unordered KV pairs (i.e., one from class A and the other from class B) whose reads are separated by exactly d reads; a higher occurrence count indicates stronger read correlations at distance d. We analyze the top three class pairs that contribute to the highest number of correlated reads at distance zero (i.e., the correlated reads come from adjacent reads) across different classes and within the same class.

Finding 8. *Correlated reads are clustered in small regions.*

Figures 4(a) and 4(b) depict the counts of correlated reads for different class pairs across different distances in CacheTrace



(a) Cross-class (CacheTrace) (b) Intra-class (CacheTrace)

(c) Cross-class (BareTrace) (d) Intra-class (BareTrace)

Figure 4: Distance-based read correlation analysis results for CacheTrace and BareTrace. The axes are in log scale. Classes in the legend are abbreviated as: TrieNodeAccount (TA), TrieNodeStorage (TS), SnapshotAccount (SA), SnapshotStorage (SS), BlockHeader (BH), and Code (C).

(a larger count means higher read correlations). The number of correlated reads decreases significantly as the distance increases, indicating that correlations are confined to small regions (e.g., within 64 reads). In particular, intra-class correlations (Figure 4(b)) exhibit significantly more correlated reads (nearly two orders of magnitude higher) at distance zero than cross-class correlations (Figure 4(a)).

Figures 4(c) and 4(d) show the counts of correlated reads for different class pairs across different distances in BareTrace. TrieNodeAccount-TrieNodeStorage (TA-TS) shows a large number of correlated reads across all distances, peaking at 640.9 M at distance four. Despite fewer reads to Code (compared to TrieNodeAccount and TrieNodeStorage), the cross-class correlations Code-TrieNodeAccount (C-TA) and Code-TrieNodeStorage (C-TS) exhibit non-negligible cross-class correlations with high correlated read counts. Intra-class correlations for TrieNodeAccount and TrieNodeStorage are more pronounced and peak at 1.21 B and 2.64 B at distance zero, respectively. Compared to CacheTrace, BareTrace shows significantly higher correlated read counts, implying that caching and snapshot acceleration reduce read correlations.

Finding 9. *Correlated reads are skewed in frequency.*

Figure 5 shows the frequency distribution of correlated reads at distances zero and 1024 (the smallest and largest distances in our analysis, respectively) in CacheTrace and BareTrace. Following Finding 8, we focus on the six most prominent class pairs in each trace. In both traces, the frequency of correlated reads at distance zero is significantly higher than at distance 1024. At distance zero, the highest number of cross-class correlated reads occurs in Code-SnapshotStorage (C-SS) (106 in CacheTrace) and TrieNodeAccount-TrieNodeStorage (TA-TS) (0.79 M in BareTrace). For intra-class correlated reads, TrieNodeAccount-TrieNodeAccount (TA-TA) exhibits the highest frequencies in both CacheTrace (405) and BareTrace (1.95 M). Caching and snapshot acceleration in CacheTrace reduce the skewness of frequency distribution compared to BareTrace, which shows a more skewed frequency distribution at distance zero. The finding suggests that correlated reads tend to be clustered in small regions, consistent with previous findings.

D. Update Correlations

Following the same method in read correlations (§IV-C), we analyze the intra-class and cross-class correlated updates of different classes of KV pairs.

Finding 10. *Correlated updates are clustered in small regions.*

Figure 6 shows the update correlations of the top three class pairs that contribute to the highest number of correlated updates across different classes and within the same class. For cross-class update correlations in CacheTrace (Figure 6(a)), the most frequent correlated updates are in LastFast-LastHeader (LF-LH) and LastBlock-LastFast (LB-LF) class pairs. The correlated updates of the two class pairs all peak at 1 M at a distance of zero, and decrease to zero at a distance of four. The reason is that these classes are used to record the latest state of blockchain synchronization and only have one KV pair each (Table I). These KV pairs are updated whenever a new block is added to the chain in a batch manner, which explains the small distance of the correlated updates.

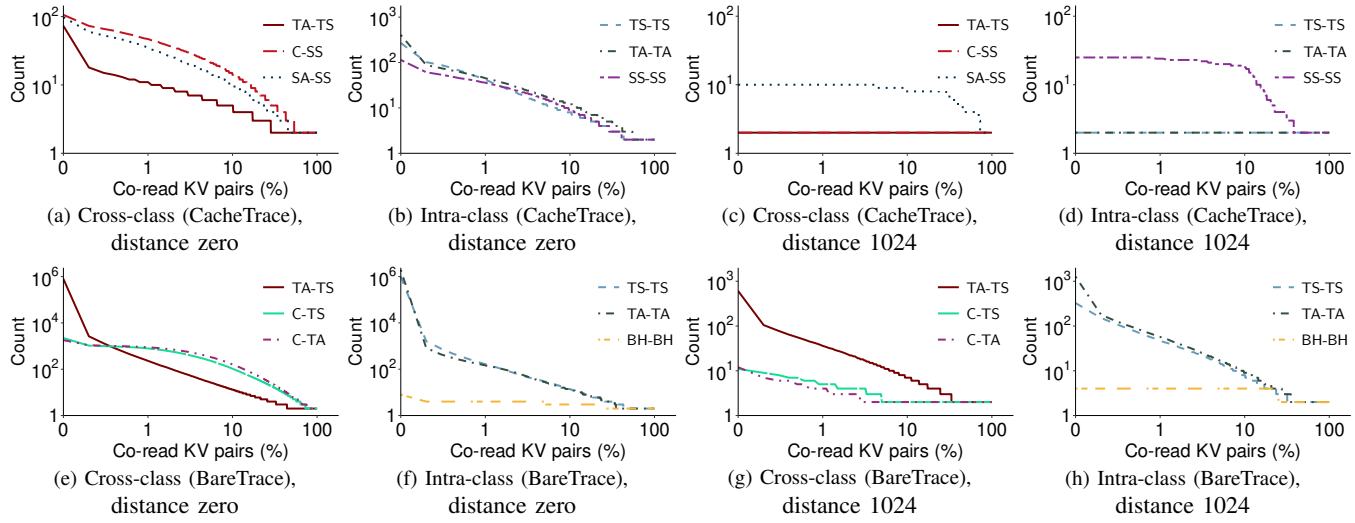
Similarly, in BareTrace (Figure 6(c)), the most frequent correlated updates also occur within these classes. However, due to the absence of snapshot acceleration, the number of updates is much higher (i.e., 3.99 B and 1.27 B in BareTrace and CacheTrace, respectively), which affects the order of updates and leads to differences in ranking. Thus, LastHeader-LastStateID (LH-LS) replaces Code-LastHeader (C-LH) in the top three class pairs. Compared to read correlations, cross-class update correlations are more clustered, since the updates are in a batch manner controlled by Geth and only happen when a new block is added, while the reads mainly depend on the transaction execution order.

For intra-class update correlations in both CacheTrace (Figure 6(b)) and BareTrace (Figure 6(d)), the most frequent correlated updates are within the world state classes and the Code class, which is bound to the smart contracts in the world state. The frequency of correlated updates decreases as the distance increases, with the count of correlated updates becoming zero when the distance is 1024 for Code. This suggests that updates within these classes also have a strong correlation within a small distance.

Finding 11. *Correlated updates have unique frequency distribution.*

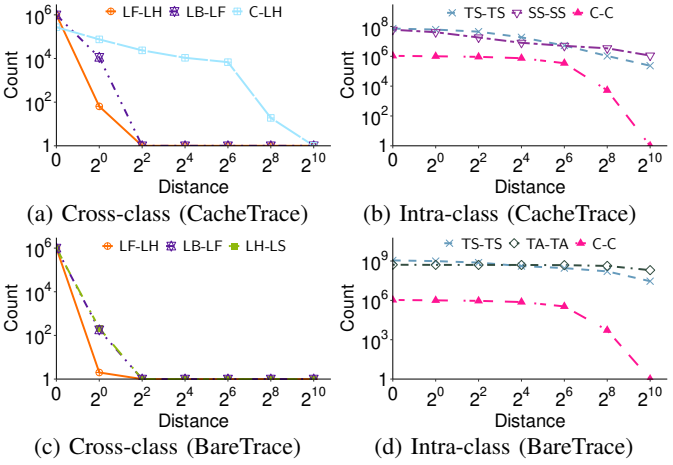
Figure 7 shows the frequency distribution of intra-class correlated updates. Recall that the most frequent correlated updates across classes are within the class pairs between LastFast, LastHeader, and LastBlock, but they only contain one KV pair each. Thus, we do not show the frequency distribution of correlated updates across these classes.

For intra-class correlated updates, the frequency distribution is unique for each class. At distance zero, TrieNodeStorage



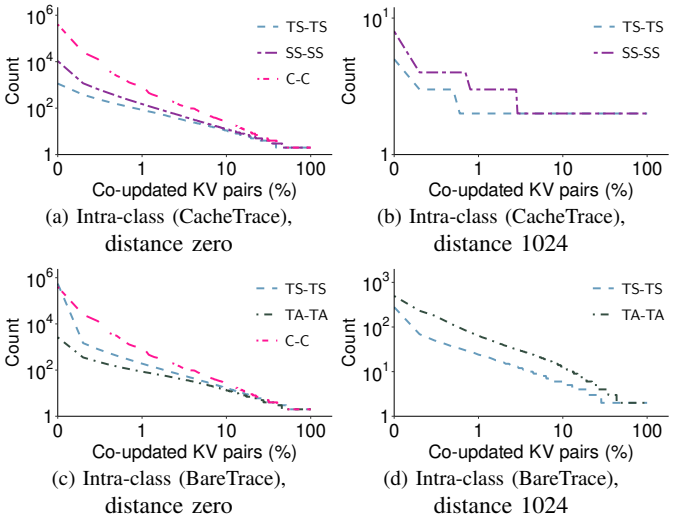
(a) Cross-class (CacheTrace), distance zero
(b) Intra-class (CacheTrace), distance zero
(c) Cross-class (CacheTrace), distance 1024
(d) Intra-class (CacheTrace), distance 1024
(e) Cross-class (BareTrace), distance zero
(f) Intra-class (BareTrace), distance zero
(g) Cross-class (BareTrace), distance 1024
(h) Intra-class (BareTrace), distance 1024

Figure 5: Frequency distribution of correlated reads. The axes are in log scale.



(a) Cross-class (CacheTrace)
(b) Intra-class (CacheTrace)
(c) Cross-class (BareTrace)
(d) Intra-class (BareTrace)

Figure 6: Distance-based update correlation analysis results for CacheTrace and BareTrace. The axes are in log scale. Classes in the legend are abbreviated as: LastFast (LF), LastHeader (LH), LastBlock (LB), LastStateID (LS) ,TrieNodeStorage (TS), TrieNodeAccount (TA), SnapshotStorage (SS), and Code (C).



(a) Intra-class (CacheTrace), distance zero
(b) Intra-class (CacheTrace), distance 1024
(c) Intra-class (BareTrace), distance zero
(d) Intra-class (BareTrace), distance 1024

Figure 7: Frequency distribution of intra-class correlated updates. The axes are in log scale.

exhibits the highest frequencies in both CacheTrace and BareTrace (i.e., 1 M), while the highest frequency of correlated updates in TrieNodeStorage is only 10 when the distance is 1024. Also, there is no intra-class correlated update in Code. This indicates that updates to these classes are more tightly coupled, consistent with the previous finding.

V. Summary of Findings

We discuss the implications of our trace analysis results for blockchain storage design.

KV storage management. Reducing storage and I/O costs is critical for blockchain storage. Based on the storage patterns from Findings 1 and 2, the KV store contains a significantly high number of small-sized KV pairs. To mitigate storage overhead, adding further indexes or metadata to the KV store should be avoided.

In terms of KV operation distributions, Finding 3 reveals that most world-state-related KV pairs are rarely or never read over extended periods. However, the LSM KV store in Ethereum still incurs high indexing and compaction overhead for these KV pairs. One possible optimization strategy is that KV pairs associated with the world state can be initially appended to a log, and are inserted into the KV store only upon being read.

Finding 4 indicates that scans are rare and limited to snapshot acceleration (SnapshotAccount and SnapshotStorage) and block headers (BlockHeader), while Finding 5 shows that deletions are prevalent. The findings suggest that LSM KV stores are unsuitable for blockchain storage, as they incur high deletion overhead due to tombstone writes and unnecessary compaction overhead for supporting scans. We recommend a hybrid KV store design, such that it is tailored to data semantics and access patterns and includes efficient deletion mechanisms

that bypass LSM compaction overhead. Finding 7 shows that snapshot acceleration incurs high storage overhead, especially as the number of KV pairs grows. Combined with Finding 3, this suggests that snapshot acceleration can be refined to target only active KV pairs (i.e., those being read after insertion).

In terms of read correlations of KV pairs, Findings 8-9 suggest that read correlations are overlooked in current blockchain storage. High read correlations exist, particularly among classes such as accounts, contract storage, and contract codes. Correlation-aware designs (e.g., co-locating frequently accessed data) can possibly enhance performance.

Findings 10-11 indicate that update correlations exhibit patterns similar to read correlations, but are more clustered within small regions (i.e., smaller distances). Given the batched writes design of Geth, we observe that Ethereum workloads show a unique read-modify-write pattern, in which reads are performed during transaction processing and then writes (updates) are performed after each block is verified, with strong correlations among both reads and updates. This suggests that correlation-aware co-location (as discussed in read correlations above) can also optimize frequently co-updated KV pairs. For example, the garbage collection overhead for obsolete KV pairs can be reduced by limiting the range of data touched during garbage collection, thereby improving performance.

In summary, the KV store for Ethereum should adopt a hybrid index structure tailored to each class. Only three classes require scans, which can be efficiently managed using an LSM-tree or B+-tree index. The world-state-related data (e.g., TrieNodeAccount, TrieNodeStorage, Code) can be initially stored together in a compact log structure with low write amplification (compared to LSM-tree), and then transitioned to a read-optimized structure (e.g., hash-based KV store) only when being read. Also, deletions are common across Ethereum workloads, so each data structure should be optimized for efficient deletion mechanisms.

Cache management. Effective cache management relies on the understanding of KV pair access distributions and correlations. Finding 6 reveals that the current caching mechanism is sub-par, especially for reads to medium-frequency KV pairs. Combined with Finding 3, which indicates that most KV pairs are not accessed over extended periods, we suggest that the caching mechanism should prioritize active KV pairs and exclude KV pairs that are never read from being admitted to the cache on the write path. In addition, our trace analysis reveals that read correlations exist within and across classes. Findings 8 and 9 show that correlated reads tend to be clustered within small regions and are frequently repeated. These findings suggest that a correlation-aware cache design can enhance hit rates and reduce I/O operations.

In summary, Ethereum can benefit from a correlation-aware cache design that considers both read distributions and correlations to optimize cache performance.

Design principles. Our findings suggest possible design principles for blockchain storage: (i) avoiding unnecessary indexes or metadata that may incur higher storage overhead; (ii) using hybrid KV store designs with tailored index structures (e.g., LSM-/B+-tree for BlockHeader, correlation-aware indexes for TrieNodeAccount and TrieNodeStorage); (iii) optimizing deletions to reduce background I/Os; (iv) providing adaptive indexing with varying granularities based on KV pair access patterns (e.g., utilizing block-level indexes for never-accessed KV pairs and individual indexes for accessed ones); (v) refining snapshot acceleration to target active KV pairs; and (vi) adopting correlation-aware designs for storage layouts and cache policies.

We provide a possible conceptual design for hybrid KV storage and correlation-aware caching as follows.

(i) Hybrid KV storage. For world-state data (e.g., TrieNodeAccount, TrieNodeStorage) and snapshot data (e.g., SnapshotAccount, SnapshotStorage), which exhibit strong read and update correlations (Findings 8-11), we can co-locate them in the same storage region of a log-based structure based on Geth's path-based storage model. This reduces I/Os to retrieve KV pairs from different regions in LSM-tree storage. For classes with high deletion rates (e.g., TxLookup) (Finding 5), we can store their KV pairs in an append-only log to mitigate I/O overhead, so that we can remove old KV pairs in batches and eliminate LSM-tree ordering maintenance. For immutable block data (e.g., BlockHeader, BlockBody, and BlockReceipts), we can also store them in append-only logs. For other KV classes with less distinct patterns and a small number of KV pairs, we keep them in the Pebble KV store as usual. Key design challenges include designing selective index structures and maintaining storage consistency across different KV classes.

(ii) Correlation-aware caching. In correlation-aware caching, we can model temporal access sequences from historical workloads to identify read and update correlations. We can further implement smart prefetching and eviction policies based on the correlation patterns. For example, when a KV pair is read, we can proactively prefetch its correlated KV pairs into the cache and evict correlated KV pairs together, so as to improve cache hits and overall performance.

VI. Conclusion

We present a comprehensive analysis of Ethereum's storage workloads using real-world KV operation traces collected from a Geth client over a 140-day span. We highlight 11 key findings that cover storage overhead, KV operation distribution, read correlations, and update correlations. Based on the findings, we provide recommendations for KV storage and caching optimization to improve Ethereum's storage management.

Acknowledgments

This work was supported by National Key R&D Program of China (2022YFB3103500), National Natural Science Foundation of China (62332004 and 62522204), Sichuan Science and Technology Program (2024NSFTD0031 and 2024YFHZ0339), and Research Grants Council of Hong Kong (GRF 14214622). The corresponding author is Jingwei Li.

REFERENCES

[1] "Ethereum," https://ethereum.org/, 2025.

[2] "Ethereum full node sync (default) chart," https://etherscan.io/chartsync/chaindefault, 2025.

[3] "go-ethereum freezer database," https://geth.ethereum.org/docs/fundamentals/databases, 2025.

[4] "go-ethereum (Geth)," https://geth.ethereum.org, 2025.

[5] "go-ethereum's KV storage schema," https://github.com/ethereum/go-ethereum/blob/master/core/rawdb/schema.go, 2025.

[6] "Proof of stake," https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/, 2025.

[7] "Proof of work," https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/, 2025.

[8] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. of ACM SIGMETRICS*, 2012.

[9] Bitcoin.org, "Bitcoin," https://bitcoin.org/, 2025.

[10] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook," in *Proc. of USENIX FAST*, 2020.

[11] S. Chen, S. GalOn, C. Delimitrou, S. Manne, and J. F. Martinez, "Workload characterization of interactive cloud services on big and small server platforms," in *Proc. of IEEE IISWC*, 2017.

[12] Z. Chen, B. Li, X. Cai, Z. Jia, L. Ju, Z. Shao, and Z. Shen, "ChainKV: A semantics-aware key-value store for ethereum system," in *Proc. of ACM SIGMOD*, 2024.

[13] Z. Chen, B. Li, X. Cai, Z. Jia, Z. Shen, Y. Wang, and Z. Shao, "Block-LSM: An ether-aware block-ordered lsm-tree based key-value storage engine," in *Proc. of IEEE ICCD*, 2021.

[14] Create4Life, "What's the number of ethereum archive nodes?" https://www.reddit.com/r/ethereum/comments/frsffp/whats_the_number_of_ethereum_archive_nodes/, 2020.

[15] ethernodes.org, "Ethereum mainnet statistics," https://www.ethernodes.org/, 2025.

[16] H. Feng, Y. Hu, Y. Kou, R. Li, J. Zhu, L. Wu, and Y. Zhou, "SlimArchive: A lightweight architecture for ethereum archive nodes," in *Proc. of USENIX ATC*, 2024.

[29] E. Technologies, "Erigon: Ethereum implementation on the efficiency frontier," https://github.com/erigontech/erigon, 2025.

[17] hyperledger, "Besu ethereum client," https://besu.hyperledger.org, 2025.

[18] G. Kadianakis, lightclient, and A. Stokes, "EIP-4444: Bound historical data in execution clients prune historical data in clients older than one year," https://eips.ethereum.org/EIPS/eip-4444, 2021.

[19] C. Labs, "Pebble," https://github.com/cockroachdb/pebble, 2025.

[20] J. Liang, W. Chen, Z. Hong, H. Zhu, W. Qiu, and Z. Zheng, "MoltDB: Accelerating blockchain via ancient state segregation," *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[21] B. K. Mohanta, S. S. Panda, and D. Jena, "An overview of smart contract and use cases in blockchain technology," in *Proc. of IEEE ICCCNT*, 2018.

[22] NodeReal, "Geth path-based storage model and newly inline state prune," https://nodereal.io/blog/en/geth-path-based-storage-model-and-newly-inline-state-prune/, 2023.

[23] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, pp. 351–385, 1996.

[24] M. M. Queiroz, R. Telles, and S. H. Bonilla, "Blockchain and supply chain management integration: a systematic review of the literature," *Supply chain management: An international journal*, vol. 25, no. 2, pp. 241–254, 2020.

[25] T. Rabl, M. Sadoghi, H.-A. Jacobsen, S. Gómez-Villamor, V. Muntés-Mulero, and S. Mankowskii, "Solving big data challenges for enterprise application performance management," *arXiv preprint arXiv:1208.4167*, 2012.

[26] P. Raju, S. Ponnapalli, E. Kaminsky, G. Oved, Z. Keener, V. Chidambaram, and I. Abraham, "mLSM: Making authenticated storage faster in ethereum," in *Proc. of USENIX HotStorage*, 2018.

[27] F. Schär, "Decentralized finance: on blockchain and smart contract-based financial markets," *Review of the Federal Reserve Bank of St Louis*, vol. 103, no. 2, pp. 153–174, 2021.

[28] P. Szilágyi, "Ask about geth: Snapshot acceleration," https://blog.ethereum.org/2020/07/17/ask-about-geth-snapshot-acceleration, 2020.

[30] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[31] J. Yang, Y. Yue, and K. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at Twitter," in *Proc. of USENIX OSDI*, 2020.

[32] C. Zhang, C. Xu, H. Hu, and J. Xu, "COLE: A column-based learned storage for blockchain systems," in *Proc. of USENIX FAST*, 2024.

A. Artifact Appendix

A.1 Abstract

Our artifact provides a modified Ethereum execution client (Geth) for trace collection and a suite of Go-based tools to analyze these traces from a key-value (KV) storage perspective. It allows for the reproduction of the 11 key findings in our paper and provides a framework for future research in blockchain storage optimization.

A.2 Artifact check-list (meta-information)

- **Program:** Go programs and shell scripts.
- **Compilation:** Go compiler (1.23), Make.
- **Binary:** Geth and custom analysis tools.
- **Data set:** Ethereum KV operation traces (BareTrace and CacheTrace). We provide sampled traces with KV operations for the processing of blocks from 20500000 to 20501000 (i.e., 1,000 blocks).
- **Run-time environment:** Linux (Ubuntu 22.04 or newer recommended) and Go version 1.23 (or newer).
- **Hardware:** For sampled trace analysis, we recommend a standard PC with at least 16 GB RAM and 20 GiB of free storage space; for full trace analysis, we recommend a powerful server with more than 256 GiB of RAM and 8 TiB SSD storage space.
- **Output:** Text log files with statistical analysis.
- **Experiments:** KV size analysis, access distribution analysis, access correlation analysis.
- **How much disk space required (approximately)?:** >20 GiB for sampled trace analysis, and >8 TiB for full trace collection and analysis.
- **How much time is needed to prepare workflow (approximately)?:** Less than 30 minutes for initial setup, including Go installation, code compilation, and sampled trace downloads.
- **How much time is needed to complete experiments (approximately)?:** Less than 2 hours with our sampled traces. Full trace analysis can take 2 weeks to 1 month, depending on hardware and network conditions.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** GNU Lesser General Public License v3.0.
- **Data licenses (if publicly available)?:** No license are applied to the public blockchain data.
- **Archived (provide DOI)?:** It is archived on Zenodo with DOI: `10.5281/zenodo.16904170`. It can be accessed at `https://zenodo.org/records/16904170`.

A.3 Description

A.3.1 How to access

The artifact is available at: `https://github.com/adslabcuhk/geth_analysis`. There are two folders:

- `go-ethereum-1.14.11/`, which contains the modified Geth client for trace collection.
- `analysis/`, which contains analysis tools and scripts for trace processing.

The repository also includes `README.md` with detailed instructions on how to set up the environment, compile the code, and run the experiments for artifact evaluation.

A.3.2 Hardware dependencies

To run experiments on our sampled traces (§A.3.4), we recommend a machine with a multi-core CPU, at least 16 GiB of RAM, and at least 20 GiB of free storage space. The machine should have a stable, high-speed Internet connection to download the sampled traces from public cloud storage.

For full trace collection and analysis, we recommend a powerful server with at least 256 GiB of RAM and 8 TiB SSD storage space. The server should have a high-speed Internet connection (without proxy) to interact with the Ethereum network and download block data from other peer nodes.

A.3.3 Software dependencies

The software dependencies include:

- `Go pebble library` for KV storage operations.
- `Ethereum rlp library` for Ethereum data serialization.
- `Sentry SDK` for error tracking and monitoring.
- `Unix system call interface` for low-level OS operations.
- `Exp/rand` for experimental random number generation.
- `xxhash` for fast non-cryptographic hash functions.
- `Protocol Buffers` for structured data serialization.
- `Prometheus client model` for metrics collection.
- `Snappy compression library` for fast data compression.

The dependencies are managed using Go modules, and can be installed using the script *./analysis/build.sh install* in the root directory of the repository. The script will automatically download and install all required dependencies.

A.3.4 Data sets

We provide sampled BareTrace (3.9 GiB) and CacheTrace (1.2 GiB) for 1,000 Ethereum blocks from 20500000 to 20501000. The traces can be downloaded from the following SharePoint link:

- `https://gocuhk-my.sharepoint.com/:f:/g/personal/pclee_cuhk_edu_hk/EsP3NcKY5VFOkY5veTWZPoEBK1sL16tL1blkuNYhsXe75w?e=HGoEku`

We do not provide the full traces that are used in our paper due to the large sizes (over 1.3 TiB and 4.1 TiB for CacheTrace and BareTrace, respectively). Nevertheless, the full traces can be obtained by running our modified Geth client on Ethereum.

Considering the large size of the KV store (around 275 GiB), we do not provide the full KV store data in our sampled dataset. Instead, we provide the statistical results for different KV classes, including the size distribution of all KV pairs in the KV store after CacheTrace is collected. The size distribution is generated by the analysis tool `countKVSizeDistribution` and can be used to reproduce the KV size distribution results presented in our paper. The file, called `kvSizeDistribution.tar` can be downloaded from the above Sharepoint link.

A.4 Installation

To install the artifact, set up the compilation environment, install the software dependencies, and build the modified Geth client and analysis tools. The following steps show the installation process. We use Ubuntu 22.04 as an example, but the steps are similar for other Linux distributions.

1. **Install Go.** Ensure that Go version 1.23 or higher is available. Install it using the package manager of the Linux distribution. If an older version of Go is installed, update it to the latest version to avoid compatibility issues.

   ```
   sudo apt install golang-go
   ```

   If the system's package manager provides an older version, install it manually.

   ```
   wget https://go.dev/dl/go1.23.2.linux-amd64.tar.gz
   sudo rm -rf /usr/local/go
   ```

```
sudo tar -C /usr/local -xzf
go1.23.2.linux-amd64.tar.gz
export PATH=$PATH:/usr/local/go/bin
```

2. **Build the modified Geth client.** To collect traces, build the modified Geth client. The binary is located at `go-ethereum-1.14.11/build/bin/geth`.

```
cd go-ethereum-1.14.11 && make
```

3. **Build the analysis tools.** We provide a suite of Go-based tools for trace analysis. To build these tools, navigate to the `analysis` directory in the repository and run the build script. If the dependencies are not yet installed, the script *./build.sh install* will install them automatically. Note that the executables will be placed in the `analysis/bin` directory.

```
cd analysis
./build.sh install # install dependencies
./build.sh build # build the analysis tools
```

A.5 Experiment workflow

The experiment workflow consists of three main stages: setup, data collection, and analysis. For a quick start, we recommend using the provided sampled traces (§A.3.4) to reproduce the key findings from our paper. In this case, start with Step 3 to run the analysis tools directly.

Step#1. Setup. Create directories for execution: `mkdir -p ethereum/consensus ethereum/execution`. Set the target block range to collect traces (e.g., from 20500000 to 20501000) in `go-ethereum-1.14.11/common/globalTraceLog.go` by changing the variables `targetStartBlockNumber` and `targetEndBlockNumber`. Then, compile the `geth` binary and copy it into `ethereum/execution`

Step#2. Trace collection. The trace collection process involves running a consensus client (e.g., Prysm) and an execution client (e.g., Geth) to sync the Ethereum network and collect KV operation traces. First, generate an authentication secret (JWT) for communication between the consensus and execution clients. See instructions below for generating the JWT secret.

```
cd ethereum/consensus
curl https://raw.githubusercontent.com/prysmati
clabs/prysm/master/prysm.sh--outputprysm.sh
chmod +x prysm.sh
./prysm.sh beacon-chain generate-auth-secret
cp jwt.hex ../jwt.hex
```

Then, run the modified Geth client to start syncing the blockchain and collecting traces. The following commands illustrate how to run the Geth client with the appropriate flags for BareTrace (as an example).

```
cd ethereum/execution
./geth --cache 0 --cache.noprefetch --snapshot
--network --datadir ./data --syncmode full
--http --http.api eth,net,engine,admin
--authrpc.jwtsecret ../jwt.hex
```

After Geth starts, run the consensus client in a separate terminal.

```
cd ethereum/consensus
./prysm.sh beacon-chain --datadir ./data
--network --execution-endpoint=http://localhost:8551
--jwt-secret=../jwt.hex --checkpoint-sync-url=http
s://beaconstate.info --genesis-beacon-api-url=http
s://beaconstate.info
```

Finally, the Geth client will start syncing the Ethereum network and collecting traces. The traces will be written to the execution directory (`ethereum/execution`) named as `geth-trace`.

Step#3. Trace analysis. Use the scripts in `analysis` to process the blockchain data and trace files.

- **KV storage management.** Download and extract `kvSizeDistribution.tar`, which contains the size distribution of all KV pairs in the KV store after CacheTrace is collected. Note that it contains the KV size distribution for each class of KV pairs. Each line of a file in the archive includes two fields: `size` and `count`, meaning that there are `count` KV pairs with the given `size`.
- **KV operation distribution.** Run `./kvOpDistributionAnalysis.sh` on a sampled trace log file. The output files will be stored in the `mergedKVOpDistribution` folder and named as `<KV class>_<KV operation type>_with_key_dis.txt`, where `<KV class>` is the name of the KV class and `<KV operation type>` is the type of KV operation (e.g., `read`, `write`, `delete`, etc.).
- **Read correlations.** Use `./readCorrelationAnalysis.sh` to collect read correlation data from the sampled trace files. The analysis tool will generate three types of output files in the `readCorrelationOutput` folder: `freq-category-<distance>.log`, `freq-sorted-<distance>.log`, and `Dist-<distance>-<class 1>-<class 2>-freq.log`, which contain the frequency of read operations between two KV classes at a specific distance, the pairs of correlated reads sorted by their frequency at a specific distance, and the pairs of correlated reads sorted by their frequency for a specific pair of KV classes, respectively.
- **Update correlations.** Use `./updateCorrelationAnalysis.sh` to collect the update correlation data from the sampled trace files. The results will be stored in the `updateCorrelationOutput` folder, with the same file naming and data structure as the read correlation files.

A.6 Evaluation and expected results

To reproduce the results in the paper, please refer to the `README.md` file and follow the instructions provided in the Evaluation section.

KV storage management (Findings 1 and 2). *Expected outcome:* KV storage management analysis produces results in Findings 1 and 2, which show that five dominant classes account for over 99.2% of total KV pairs, with only an average size of 79.1 bytes, while 15 out of 29 classes have only one KV pair for maintaining system state. Also, the size distribution of KV pairs for world state data is skewed. *Approximate runtime:* Zero, as the results are included in the downloaded file (§A.3.4).

KV operation distribution (Findings 3 - 7). *Expected outcome:* KV operation distribution analysis produces results in Findings 3 to 7, which illustrate that only three classes perform scans, while deletions are common. The distribution of KV operations is highly skewed, with a few classes accounting for the majority of operations. *Approximate runtime:* 10 compute minutes.

Read correlations (Findings 8 and 9). *Expected outcome:* Read correlation analysis produces results in Findings 8 and 9, which show that read correlations exist within the same class and across different classes. *Approximate runtime:* 20 compute minutes.

Update correlations (Findings 10 and 11). *Expected outcome:* Update correlation analysis produces results in Findings 10 and 11, which show that update correlations exist within the same class and across different classes. The update correlations are weaker than read correlations since not all read KV pairs will be updated. *Approximate runtime:* 20 compute minutes.