ThermoKV: Decoupling Block-Body Read-Path Depth from Compaction History in Ethereum

***Abstract*—As the complete chain history of Ethereum Archive nodes exceeds 13 TB, the Log-Structured Merge Tree (LSM-tree) engines used by Geth encounter a fundamental performance bottleneck: compaction continuously sinks immutable block-body data to the deepest storage levels, making read latency proportional to data age rather than access frequency. We call this the *Read-Sink Effect* and show that neither per-level cache tuning nor write-time data reorganisation can resolve it, because the root cause—a static, write-driven index hierarchy—persists after compaction.**

We present ThermoKV, a drop-in storage-layer extension for Geth that resolves the Read-Sink Effect through three co-designed components. GroupChain packs consecutive block bodies into append-only flat groups, eliminating spatial fragmentation and reducing write amplification to $1\times$ for block data. ThermoList maintains a frequency-aware tiered index that dynamically routes hot groups to DRAM and cold groups to disk, decoupling read latency from compaction depth. SemanticPredictor intercepts two causal signals—transaction provenance reads that structurally precede block-body reads, and adjacent-group locality at hotspot boundaries—to promote index tiers before the first access arrives.

Evaluated on 6.56 M Ethereum mainnet blocks, ThermoKV reduces P99 block-lookup latency by 86% and write amplification from 2.81 to 1.00 compared to Geth. It outperforms the strongest read baseline (ChainKV) by 19% in throughput, reduces cold reads by 78.9% and delivers 90.7% lower block-body latency on transaction-triggered workloads.

***Index Terms*—Blockchain storage, LSM-tree, Ethereum, archive node, frequency-aware indexing, read amplification**

I. Introduction

Ethereum Archive nodes are the query infrastructure for blockchain history. Block explorers, DeFi analytics platforms, on-chain indexers, and regulatory auditing services all depend on Archive nodes to retrieve arbitrary historical blocks and transactions at interactive latency [1], [2]. The complete chain history now exceeds **13 TB** on Ethereum mainnet [3] and grows by hundreds of gigabytes per year.

Ethereum's continuous state-update workload—frequent MPT trie writes at 12-second block intervals [4]—makes an LSM-tree the natural storage choice: its sequential-write design delivers high throughput for the state data that must be updated on every transaction, and its Bloom-filtered hierarchy enables efficient point lookups without full scans. The same engine, however, also stores block bodies. Here the fit breaks down. Block bodies are written exactly once—at block finalisation—and never modified. Their read hotness is determined entirely by *application semantics*: which historical block ranges a block explorer, indexer, or analytics tool happens to query today. The blocks containing the earliest interactions with major stablecoin and DEX contracts—written three to five years ago—absorb the overwhelming majority of current read traffic, while blocks written in recent weeks attract far less sustained read traffic from historical query workloads. Our measurements on real Ethereum mainnet workloads (Section III) show that the Jaccard similarity between write-recent and read-hot block groups is **0.00** at million-request scale; write recency and read frequency are completely decoupled for this data class.

This decoupling has a structural consequence. LevelDB's six-level compaction continuously pushes block bodies toward L5 as new blocks arrive, regardless of how frequently those blocks are subsequently read. As measured on real Ethereum mainnet workloads (Section III), **98.2%** of all block-body reads resolve only at L5—meaning every such read probes L0–L4 first, five unnecessary levels. We call this the **Read-Sink Effect**: compaction sinks block-body data to the deepest levels, and the depth at which a group resides is determined by when it was written, not by how often it is read. The consequences are concrete: a block explorer serving transaction-history queries hits this overhead on every request for a historically hot block, which constitutes the overwhelming majority of Archive-node traffic; an indexer scanning historical block ranges issues an average of **8.7 random reads** per 10-block window against the baseline Geth stack (Section III). This overhead cannot be eliminated by tuning LevelDB parameters: compaction will always migrate block bodies deeper as new blocks arrive, continuously recreating the same penalty for historically hot data.

This bottleneck resists existing solutions. The Read-Sink Effect has a single precise cause: the LSM-tree's index hierarchy is write-driven and static, and no existing PBSS-compatible system can shorten a block group's index path after compaction. Three families of prior work each address a symptom while leaving this root cause intact.

Per-level cost reduction (Bloom-filter tuning [5], cache management [6], hardware tiering [7]) makes each level probe cheaper, but a query whose target is in L5 still traverses five empty levels first. This is a qualitative $O(L)$-to-$O(1)$ gap that per-level optimisation cannot close regardless of how efficient each probe is.

Write-time reorganisation (Block-LSM [8], ChainKV [9]) co-locates related records at write time to reduce compaction I/O. But layout decisions made at write time are irrevocable: a contract deployed months ago that now drives the majority of DEX traffic was cold at write time, and its block group will remain in L5 regardless of its current read frequency. Both systems also modify the state-trie key namespace, breaking compatibility with Geth's Path-Based State Scheme (PBSS,

default since v1.13).

Statistical prefetching (Leaper [10]) learns access co-occurrence patterns and prefetches at compaction time. This fires seconds to minutes after a hotspot shift—too late to help—and cannot predict the first access to a newly hot block range because no co-occurrence history yet exists. Our evaluation shows Geth+Leaper reduces latency by only **59.9%** versus Geth, remaining $2.93\times$ slower than ThermoKV.

The gap that all three families share: none can *dynamically reposition a hot block group's index entry to a shorter access path after compaction*, while remaining PBSS-compatible.

We present ThermoKV, which addresses the Read-Sink Effect through a single unifying idea: *decouple read-path index depth from compaction history by maintaining a runtime frequency-driven routing layer above a GroupChain flat file and below the application*. **GroupChain** exploits blockchain temporal immutability to pack consecutive blocks outside LevelDB's compaction path, achieving WAF $= 1\times$ for block data. **ThermoList** is a frequency-driven tiered routing index that repositions hot groups to DRAM after compaction, dynamically shortening their access path in response to read frequency rather than write age. **SemanticPredictor** eliminates cold-miss bursts at hotspot transitions by exploiting two causal signals—the structural two-phase transaction retrieval invariant and adjacent-group locality—that are observable before the cold miss arrives. All three components operate exclusively on block-body records; MPT state-trie records pass through to LevelDB unchanged, preserving full PBSS compatibility.

ThermoKV is evaluated on 6.56 M Ethereum mainnet blocks against four competing systems. GroupChain achieves WAF = **1.00** with write throughput **4.08**$\times$ Geth. ThermoList delivers random-block P99 of **186** μ**s** versus Geth's 1,359 μs ($-$**86%**) and **44%** lower than Block-LSM, and lifts range-query QPS by **2.32**$\times$ over Geth. SemanticPredictor achieves **100%** hit precision on transaction queries (body latency $-$**90.7%**) and reduces sequential cold reads by **78.9%**. Geth+Leaper, the strongest prefetching baseline, remains **1.59**$\times$ slower in QPS and **2.93**$\times$ worse in P99—confirming that per-level optimisation cannot substitute for index-depth reduction. The ThermoKV prototype and all evaluation workloads described in this paper are publicly available at https://github.com/nnzhaocs/thermokv.git.

The contributions of this paper are as follows.

1) We identify and quantify the **Read-Sink Effect**: the structural decoupling between write-driven LSM compaction and application-driven block-body read hotness in Ethereum Archive nodes (Jaccard = 0.00; 98.2% of reads at L5), and show it is irreducible by per-level optimisation, write-time reorganisation, or statistical prefetching.
2) We design **GroupChain**, which exploits blockchain temporal immutability to eliminate compaction for block data, achieving WAF $= 1\times$ for block records and reducing k-block range queries to one sequential read.
3) We design **ThermoList**, the first frequency-driven, PBSS-compatible tiered routing index that dynamically adjusts block-group index depth at runtime in response to access frequency, decoupling read-path depth from compaction history.
4) We design **SemanticPredictor**, which exploits the structural two-phase transaction retrieval invariant and adjacent-group locality as causal prediction signals, reducing excess cold misses by 78.9% and rapidly recovering steady-state performance after a hotspot shift through autonomous Tier-1 promotion.

II. Background and Related Work

This section establishes the two facts that together explain why the Read-Sink Effect exists and why prior work cannot fully resolve it: Ethereum's storage layer holds two data classes with incompatible access semantics, and LSM-tree compaction is the right policy for one class but structurally harmful for the other.

A. Ethereum's Two Data Classes

Ethereum maintains a global world state mapping every account address to its balance, nonce, code, and contract storage [1]. Geth persists this state as a Merkle Patricia Trie (MPT): each trie node is stored as a key-value pair in LevelDB or PebbleDB [11], both LSM-tree engines. Beyond the MPT, Geth stores the complete body of every historical block—the serialised transaction list and uncle headers—in the same LevelDB instance under a distinct key prefix.

These two data classes share a storage engine but have fundamentally different semantics.

a) MPT state nodes: State nodes are *frequently updated*. Each transaction that modifies an account or storage slot creates new trie nodes and obsoletes old ones. Geth's legacy hash-based scheme keys each node on the Keccak-256 hash of its content, so every state change writes a new key. Geth v1.13 introduced the *Path-Based State Scheme (PBSS)*, which keys nodes on their trie path instead, enabling in-place updates and reducing write amplification for state data. PBSS requires the state-trie key namespace to remain structurally intact: any storage layer that rekeys or reshuffles state-trie entries breaks PBSS's in-place update invariant and forces a full re-synchronisation.

b) Block bodies: Block bodies are *written exactly once and never modified*. Once a block is finalised, its body is committed and then immutable for the lifetime of the node. Crucially, block-body read hotness is determined entirely by *application semantics*: which historical block ranges an indexer, explorer, or analytics tool queries today. A block from years ago containing the deployment of a major DeFi contract may be read thousands of times per day, while a block written yesterday may never be read again. Write recency and read frequency are essentially independent for this data class. Prior work has measured that block-body and receipt data account for a substantial fraction of total KV write volume in a full-synchronisation workload [12].

c) Two-phase transaction retrieval: Retrieving a historical transaction by hash—the dominant query type for block explorers and DeFi analytics—requires two sequential reads. First, a *transaction provenance record* is consulted to resolve the block number and intra-block position of the transaction. Second, the *block body* at that block number is fetched and the transaction is extracted. This two-phase structure is not a Geth implementation detail; it is an architectural necessity for any storage system that indexes transactions by hash rather than by block position. The provenance read causally determines the block-body read—a structural regularity exploited by ThermoKV's SemanticPredictor (Section IV-D).

B. LSM-tree Compaction: Right for State, Wrong for Block Bodies

An LSM-tree organises data into multiple levels. New writes land in a MemTable and are flushed to Level 0 (L0) SSTables; background compaction merges SSTables downward through L1, L2, and deeper levels, which hold the large majority of stored data on a mature node. A point lookup probes each level top-down: it tests the Bloom filter at each level and, on a positive, performs a binary search within the SSTable. In LevelDB's six-level configuration (L0–L5), data in L5 requires probing five upper levels before resolution—five wasted operations if those levels contain no match, which is the common case for data that has survived several compaction cycles.

a) Compaction is appropriate for MPT state: The current state—recently written—stays near L0 and L1, where lookups succeed quickly. Stale historical state nodes are compacted to deeper levels but are rarely queried. Compaction's write-recency placement naturally aligns with the read pattern: recent data is hot, old data is cold.

b) Compaction is harmful for block bodies: A block body is written once and then compacted to deeper levels within hours of arrival, as newer blocks push it down. Block-body read hotness is not related to write recency as measured in Section III-B, meaning every such read traverses L0–L4 first—five unnecessary probes on NVMe hardware. This misalignment cannot be corrected by tuning LevelDB parameters: compaction will always migrate block bodies deeper as new blocks arrive, continuously re-creating the same penalty for historically hot data.

C. General LSM Read Optimisations

A large body of work optimises the LSM-tree read path. These approaches all reduce *per-level traversal cost* rather than the *number of levels traversed*—a distinction that is critical when 98.2% of reads resolve at L5 regardless of how efficient each probe is.

a) Bloom-filter and index optimisation: Monkey [5] derives the optimal per-level Bloom-filter memory allocation that minimises false-positive rates under a fixed memory budget; ElasticBF extends this to dynamic workloads. These approaches reduce the probability of a wasted probe at each level, but a query still visits all L levels: if the target is in L5, five negative probes remain unavoidable. REMIX [13] reorganises the virtual sorted order across SSTables so that range scans require a single pass rather than a multi-stream merge; this reduces range-query read amplification but does not shorten the traversal depth for point lookups.

b) Cache management: RocksDB and MyRocks [14] integrate block and table caches to retain hot SSTable blocks in DRAM. RocksMash [6] adds a persistent two-tier cache that tracks compaction events to avoid spurious invalidations. LSbM-tree [15] re-enables buffer-pool caching for mixed read–write workloads via dynamic metadata pinning. Cache-based approaches share two limitations for the block-body access pattern: compaction invalidates a portion of the cached metadata, causing abrupt hit-ratio drops; and the cache is populated *reactively*, so the first read to a newly hot group always incurs full multi-level traversal—a penalty that recurs at the leading edge of every hotspot shift.

c) Hardware-tiered storage: SpanDB [7] places the WAL and top LSM levels on a fast NVMe device while keeping lower levels on slower storage, concentrating read-critical metadata on high-IOPS hardware. Under the Read-Sink Effect, however, hot block-body data resides in L5—the level SpanDB places on the *slower* device—so hardware tiering provides limited benefit for this access pattern.

d) Statistical prefetching: Leaper [10] builds a frequency model from write-time access patterns and prefetches hot key ranges into the LevelDB block cache at compaction time. Bourbon [16] integrates a learned index into the WiscKey [17] key-value-separated LSM-tree. Both systems target the block cache, reducing per-level traversal cost without shortening the traversal depth itself. Leaper's prediction signal is correlational—it cannot act on the first access to a newly hot block range—and its trigger fires at compaction time, seconds to minutes after a workload shift.

D. Blockchain-Specific Storage Systems

Several systems embed blockchain access semantics into the storage engine, achieving significant write-path improvements but leaving the core read-path tension unresolved.

a) mLSM: Raju et al. [18] replace LevelDB's SSTables with per-level Merkle trees, enabling authenticated reads in one level traversal. Profiling on Ethereum mainnet found that 93.8% of block-processing time is spent in the storage layer with a 7× read amplification factor—the same bottleneck ThermoKV targets. However, mLSM replaces the SSTable format entirely and modifies the compaction algorithm, requiring a full re-implementation of the storage layer with no incremental deployment path.

b) Block-LSM: Block-LSM [8] assigns shared key prefixes to KV pairs belonging to the same block group, aligning SSTable boundaries so that compaction merges only same-group keys. This reduces write amplification significantly and provides better spatial locality within compacted files. Two limitations remain: the layout decision is made at write time and is irrevocable—a hot group in L5 stays in L5 regardless of its read frequency—and Block-LSM modifies the state-trie

key structure, breaking PBSS compatibility and requiring a full re-synchronisation to deploy on an existing Geth node.

c) ChainKV: ChainKV [9] separates block bodies and state data into two LevelDB zones with a Space-Gaming Cache (SGC) that adaptively splits the DRAM budget between zones based on observed miss rates. SGC is a meaningful step toward workload-adaptive caching, but it adjusts only the memory split between two static zones—it cannot change the traversal depth within a zone after compaction. A cache-missed read still traverses L0→…→L5 before resolution; SGC only affects whether the result is cached afterward. ChainKV also modifies the state-trie key structure and is incompatible with PBSS.

d) SolsDB: SolsDB [19] replaces LevelDB with a single-level ordered log-structured database that exploits the natural sort order of keys written per block to eliminate multi-level compaction, resolving each query in at most one I/O. However, its globally sorted design requires modifying Ethereum's block verification logic and re-sorting keys at write time, sacrificing Geth protocol compatibility.

e) AppendChain: AppendChain [20] migrates Geth to RocksDB with per-type column families, large-value separation, and a Sorted MPT with structure-aware key encoding. AppendChain reduces write amplification by 94% and compaction frequency by 98%, but does not address the read-sink effect for historical block-body data and modifies the MPT key structure.

f) Erigon: The Erigon client [21] departs from Geth's hash-addressed MPT by maintaining flat tables and an inverted index, shrinking archive storage from ∼13 TB (Geth) to ∼2 TB. Erigon is a fundamental re-architecture that eliminates MPT entirely; ThermoKV is an incremental optimisation layer compatible with Geth's existing data model and deployable without altering the consensus or execution pipeline.

E. Read Workload Benchmarks

Prior blockchain-storage evaluations rely predominantly on synthetic or semi-synthetic workloads. BlockBench [22] defines the standard KVSTORE and SMALLBANK benchmarks widely adopted by subsequent work. COLE [23] uses BlockBench-backed YCSB-style KVSTORE workloads with read-only, read-write, and write-only settings, supplemented by provenance queries over randomly selected addresses and block-height ranges. SlimArchive [24] evaluates historical state access using randomly sampled $(address, storage\text{-}key)$ pairs and random block numbers. GRuB [25] combines mixed YCSB workloads with real Ethereum contract-call traces, but targets data-feed applications rather than block-body retrieval. None of these workloads captures the transaction-count-driven access skew or the two-phase provenance-to-body retrieval structure that dominates real Archive-node traffic. Our workloads are constructed directly from real mainnet block metadata detailed in Section VI-A: R-Block uses transaction-count-weighted sampling derived from the BigQuery public Ethereum dataset [26]; R-Range is a deterministic scan from real contract-deployment blocks; R-Tx preserves the causal two-phase structure of transaction-hash retrieval; and H-series derives its hotspot sets from observed R-Block access frequencies rather than manually chosen Zipf parameters.

TABLE I

COMPARISON OF RELATED STORAGE SYSTEMS. *Dyn. read adapt.*: ADJUSTS INDEX DEPTH OR DATA PLACEMENT AT RUNTIME IN RESPONSE TO READ FREQUENCY. *PBSS compat.*: DEPLOYABLE ON GETH V1.13+ WITHOUT RE-SYNCHRONISATION. *Incremental*: REQUIRES NO REPLACEMENT OF CORE GETH COMPONENTS.

System	Write opt.	Read opt.	Dyn. read adapt.	PBSS compat.	Incremental
mLSM [18]	✓	✓	✗	✗	✗
Block-LSM [8]	✓	✓	✗	✗	✓
ChainKV [9]	✓	partial	partial	✗	✓
SolsDB [19]	✓	✓	✗	✗	✗
AppendChain [20]	✓	✗	✗	✗	✓
Erigon [21]	✓	✓	✗	✗	✗
Leaper [10]	✗	✓	✗	✓	✓
ThermoKV (ours)	✓	✓	✓	✓	✓

F. The Unaddressed Gap

Table I summarises the key dimensions along which prior systems differ from ThermoKV.

Two observations stand out. Every existing PBSS-compatible system optimises the write path through static write-time data organisation; none dynamically repositions a hot block group's index entry to a shallower tier after compaction. Systems that do achieve deeper read-path improvement (mLSM, SolsDB, Erigon) all require replacing core Geth components, incurring incompatibility with mainnet.

ThermoKV occupies the previously empty cell: PBSS-compatible, incrementally deployable, and dynamically adaptive on the read path. GroupChain eliminates spatial fragmentation by co-locating consecutive block bodies outside LevelDB's SSTable hierarchy. ThermoList dynamically promotes hot groups to memory-resident index tiers after compaction, bypassing the SSTable traversal entirely. SemanticPredictor acts before the first access to a newly hot group, exploiting the causal two-phase structure of transaction retrieval. These three capabilities are unavailable in any single prior system.

III. MOTIVATION

The design of ThermoKV is driven by a single, unified observation: *the data that Ethereum reads most is not necessarily the data that its storage engine places closest to the reader*. We substantiate this claim in three steps. Unless stated otherwise, all measurements in this section use the M4 dataset (Ethereum mainnet blocks 0–7.95 M, 6,564,000 unique blocks, ∼352 M transactions) with LevelDB block cache disabled; read queries are drawn from a transaction-count-weighted sample (R-Block workload) whose construction is detailed in Section VI-A.

A. Observation 1: Block-Body Read Demand Is Highly Skewed and Spatially Fragmented

a) Access concentration: Figure 1 plots the Lorenz curve of block-group read demand under R-Block. The distribution is strongly concentrated: the Gini coefficient is **0.78**, the top-10% of groups absorb **67.9%** of all reads, and the top-20% absorb **86.6%**. Because ThermoKV operates at block-group granularity, we report skew at this level; individual-block skew

is at least as concentrated. These figures confirm that a small subset of groups drives the overwhelming majority of read traffic—a subset that, as we show next, is structurally buried by the LSM-tree.

b) Spatial fragmentation amplifies the cost: The hot subset is not only buried in deep LSM levels; it is also physically fragmented. Each block body is committed as an independent key–value pair, so a range query spanning k consecutive blocks requires up to k independent random reads, even though the blocks were produced sequentially and are often consumed together by indexers, explorers, and reorganisation handlers. On our platform, a 10-block range query against the baseline Geth stack issues an average of **8.7 random reads**. Note that Block-LSM [8] reduces this by aligning SSTable boundaries along key prefixes at write time, but does not eliminate it: once compaction has occurred, multi-level traversal still applies to every read, and the layout cannot be revised to reflect subsequent access frequency. A fundamentally different approach—physically co-locating consecutive block bodies outside the SSTable hierarchy entirely—is therefore needed, and motivates the GroupChain component described in Section IV-B.

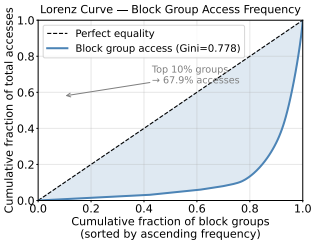

Fig. 1. Lorenz curve of block-group read demand (R-Block, M4). Gini = 0.78; top-10% of groups absorb 67.9% of reads.

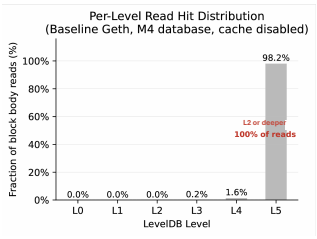

Fig. 2. Per-level read hit distribution under baseline LevelDB (M4, cache disabled). 98.2% of reads resolve only at L5.

B. Observation 2: Compaction Buries the Hot Subset (The Read-Sink Effect)

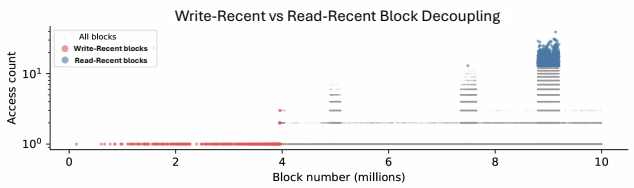

Fig. 3. Write-recent vs. read-hot block groups (M4). Write-recent groups (those written earliest, now deepest in the LSM hierarchy) cluster in early block-number ranges; read-hot groups concentrate in more recent regions. Jaccard similarity = 0.00.

a) Write recency and read frequency are decoupled: LSM-trees place newly written data near the top of the hierarchy and compact it downward over time. Because block bodies are written exactly *once* and never modified, there is no meaningful notion of "write hotness" for this data class: every block body is written with identical frequency (once), and its position in the LSM hierarchy is determined solely by how long ago it was written. In Ethereum archive workloads, however, write recency is weakly aligned with read frequency: data written long ago may still account for a substantial fraction of present-day reads. To quantify this decoupling, we compare the top-1000 write-*recent* block groups (those written earliest, now deepest in the hierarchy) with the top-1000 read-hot block groups under R-Block. Their Jaccard similarity is **0.00**, confirming that write-driven placement provides no useful signal for read-path optimisation. Figure 3 visualises this decoupling: write-recent groups cluster in early block-number ranges, while read-hot groups concentrate in much more recent regions.

b) Per-level read hit rates confirm the sink: Figure 2 reports the fraction of reads satisfied at each LevelDB level (cache disabled to isolate structural effects):

- **L0–L1: approximately 0%.** Despite holding the most recently written data, these levels satisfy almost none of the reads in our workload.
- **L2–L4: a small minority.** Together they account for the remaining shallow and mid-level hits.
- **L5: 98.2%.** The large majority of reads resolve only at the deepest level.

Aggregated differently, **100% of reads hit L2 or deeper**, consistent with a layout in which the hot read target has already been compacted far from the reader.

c) Why this is structural, not incidental: The measured hit distribution follows directly from LevelDB's default capacity profile at Ethereum archive scale. In our setup, LevelDB uses six disk levels (L0–L5): L0 triggers at $\sim$16 MB, L1 is capped at 10 MB, and lower levels grow by roughly $10\times$, yielding capacities of approximately 100 MB, 1 GB, 10 GB, and 100 GB for L2–L5. The combined capacity of L0–L4 is only $\sim$11 GB, whereas the M4 dataset occupies $\sim$96 GB after import. Most archive block data therefore lands in L5 by construction, independent of its future read frequency.

The **Read-Sink Effect** arises because the LSM-tree's index hierarchy is *write-driven and static*: the depth at which a block group resides is determined by how long ago it was written, not by how often it is later read. Once a frequently accessed group is compacted into L5, every subsequent read must still traverse L0–L4 before resolution. This is not a corner case for Ethereum archive nodes—it is the structural operating condition for any group that has been in the database for more than a few compaction cycles. Furthermore, this problem is not Ethereum-specific: it arises whenever an LSM-tree stores append-only immutable data whose read hotness evolves independently of write time. Ethereum archive nodes are a particularly acute instance because the gap between write time and peak read time can span years, and the total data volume saturates L5.

C. Observation 3: Existing Read Optimisations Do Not Fully Resolve the Read-Sink Effect

A natural question is whether existing LSM-tree read optimisations—Leaper, RocksMash, SpanDB, or blockchain-specific designs such as Block-LSM and ChainKV—can

address the read-sink effect identified above. Our analysis suggests they do not fully do so, for three related reasons.

a) Per-level cost reduction, not traversal depth reduction: Bloom-filter tuning (Monkey), cache management (RocksMash), and hardware-tiering (SpanDB) all operate *within* the existing level structure: they reduce the cost of individual probes but leave the *number of levels traversed* unchanged. When **98.2%** of reads resolve only at L5 (Figure 2), a query to a hot block group still probes L0–L4 first, regardless of how efficient each probe is.

To see why this matters, consider two strategies for a query whose target resides in L5. *Halving per-level probe cost* reduces the overhead per level but keeps the same number of probes on the critical path. *Promoting the group's index entry to DRAM* routes the query through a memory-resident index, bypassing the SSTable hierarchy entirely and issuing at most one targeted disk read. The distinction is qualitative: per-level optimisation reduces a constant factor; index-tier promotion reduces traversal depth from $O(L)$ to $O(1)$.

b) Write-time layout is static after compaction: BlockLSM [8] and ChainKV [9] improve spatial locality by reorganising data at write time. These techniques reduce compaction overhead and make physical layout more regular. However, the layout decision remains *static* after compaction: once a block group is buried in a deep level, its lookup depth is not subsequently adjusted to match its observed read frequency. This limits their ability to bring old-but-hot data back onto a shorter read path as workload patterns evolve.

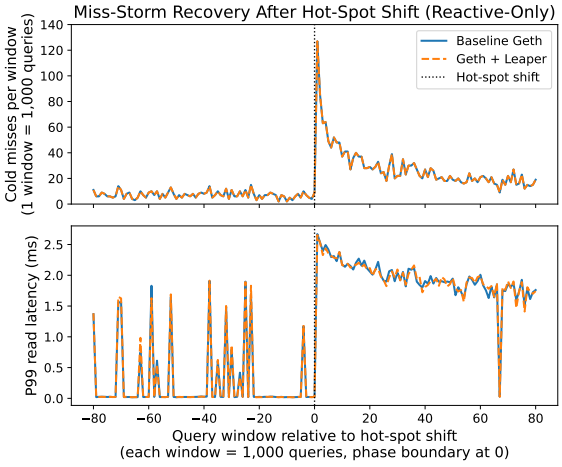

Fig. 4. Cold-miss count per 1 000-query window under an abrupt hotspot shift (H-series, M4). Both baseline Geth and Geth+Leaper require approximately 24 windows to recover; peak cold-miss count reaches 127. (ThermoKV curve omitted here; see Section VI-E for the full comparison.)

c) Existing prefetchers lack the signals needed for archive-node block-body workloads: Reactive mechanisms adapt slowly. Figure 4 reports a controlled hotspot-shift experiment in which the dominant access region changes abruptly. Baseline Geth and Geth+Leaper both require approximately **24 windows** to return to their steady-state miss level after the shift; the peak cold-miss count reaches **127**. Reactive cache protection alone is therefore slow to adapt when the hot region changes abruptly.

Statistical prefetchers do not exploit causal structure. Leaper [10] predicts hot key ranges from historical access co-occurrence and prefetches them at compaction time. This can help when future accesses are well explained by past frequency. However, archive-node block-body reads also expose a more direct structural signal: transaction retrieval is naturally a *two-phase lookup*. The system first reads a *transaction provenance record* to locate the containing block, then reads the corresponding *block body*. These two accesses are not merely correlated—they are *causally ordered by the lookup procedure itself*: the provenance record exists solely to determine which block body to read next. A predictor that acts on the provenance read can therefore promote the target group *before* the block-body miss occurs. Leaper cannot exploit this signal for two reasons. First, its trigger fires at compaction time—seconds to minutes after a hotspot shift—whereas the provenance-to-body causal sequence plays out within a single query. Second, Leaper is correlational: it requires co-occurrence history, and offers no prediction for a block group being queried for the first time.

Adjacent-group locality provides a complementary signal. Beyond the causal two-phase pattern, archive-node workloads also exhibit sequential locality at the block-group level: indexers scan consecutive ranges, reorganisation handlers traverse local neighbourhoods, and user browsing sessions tend to visit adjacent blocks. This adjacency signal is weaker than the provenance trigger, but provides a low-cost cue for proactive promotion of neighbouring groups.

d) Summary: Table II summarises the three gaps and the capability each requires.

The Read-Sink Effect has a single root cause: the LSM-tree's index hierarchy is *write-driven and static*, while Ethereum block-body read demand is *application-driven and dynamic*. ThermoKV addresses this mismatch with three co-designed components—GroupChain, ThermoList, and SemanticPredictor—each targeting one of the three gaps identified above; their design is presented in Section IV.

IV. ThermoKV Design

A. Overview

The root cause identified in Section III is a structural mismatch between how Ethereum's storage engine organises data and how applications actually read it, operating at two independent levels. At the *spatial* level, each block body is committed as disjoint key–value pairs that may be scattered across multiple SSTables, forcing a range query spanning k consecutive blocks to issue up to k random disk reads. At the *temporal* level, LSM-tree compaction relocates data toward the deepest levels driven by write time, not read frequency; because the Jaccard similarity between write hotspots and read hotspots approaches zero (Section III), the most frequently queried blocks are structurally buried in the deepest levels—the *Read-Sink Effect*—and cannot be corrected by parameter tuning within the LSM-tree framework.

TABLE II
WHY EXISTING READ OPTIMISATIONS DO NOT FULLY RESOLVE THE READ-SINK EFFECT FOR ETHEREUM ARCHIVE-NODE BLOCK-BODY WORKLOADS.

Approach	Mechanism	Why insufficient
Bloom tuning(Monkey)	Lower false-positive rate	Probe cost decreases; traversal depth unchanged
Cache mgmt(RocksMash)	Retain hot SSTable blocks	Helps after first access; does not shorten index path; compaction invalidates cache
Hardware tier(SpanDB)	Faster storage for top levels	Hot block data resides in deepest level; SpanDB places that on slower storage
Write-time layout(Block-LSM, ChainKV)	Co-locate related records	Layout cannot be revised after compaction; write-recent and read-hot groups remain disjoint
Stat. prefetching(Leaper)	Prefetch from historical co-occurrence	Fires at compaction time; requires history; cannot exploit causal two-phase provenance signal

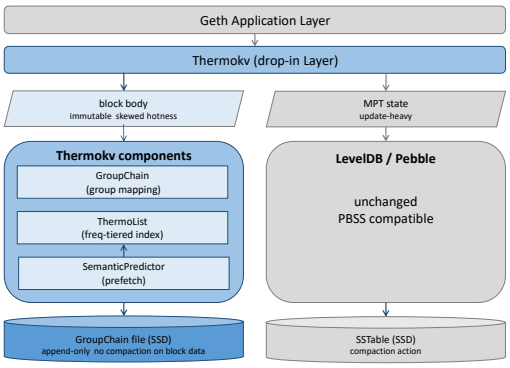

Fig. 5. ThermoKV architecture. Block-body queries are routed through GroupChain, ThermoList, and SemanticPredictor; all other record types reach LevelDB unmodified.

ThermoKV addresses both levels with a unified design philosophy: *decouple the write path from the read path at the level of data organisation*. On the write side, block bodies are packed into fixed-cohort *block groups* and appended to a flat file, eliminating fragmentation and achieving write amplification of $1\times$. On the read side, a frequency-aware tiered index routes hot groups to DRAM and cold groups to one targeted disk read, with tier assignments continuously revised at runtime. A lightweight predictor intercepts causal access patterns to pre-stage groups before reads arrive.

As shown in Figure 5, ThermoKV is deployed as a drop-in interception layer inside Geth: block-body queries are routed through ThermoKV; all other records pass through to LevelDB unchanged, preserving full PBSS compatibility.

a) Components, dependencies, and data flow: ThermoKV comprises three co-designed components that form a dependency chain. GroupChain packs S consecutive blocks into one indexed group, reducing the number of indexable units to $O(N_{\text{blocks}}/S)$ and making the tiered index memory-footprint tractable. ThermoList maintains the tiered index over these groups: its express tiers (L1–L2) offer materially shorter access paths than the base tier, giving SemanticPredictor a meaningful target for proactive promotion. SemanticPredictor fills the one-access window that ThermoList's reactive counter cannot cover—the first visit to a newly hot group—by detecting causal precursors and promoting the group before the read arrives.

Write path: a GroupPacker buffers incoming blocks until S consecutive blocks are assembled; the group is serialised with an index header and appended to the GroupChain file; ThermoList registers the new group at base tier L0.

Read path: the query is intercepted before reaching LevelDB. Step 1: SemanticPredictor checks whether a transaction provenance read has just resolved a block number; if so, it proactively promotes that group's tier before the block-body read arrives. Step 2: ThermoList resolves the group—L2 entries are served from DRAM; L1 entries require one targeted `pread`. Step 3: the Dual In-Group Index (§IV-B) resolves the block or transaction offset in $O(1)$.

B. GroupChain: Consecutive Block-Group Packing

Spatial fragmentation arises from a granularity mismatch: LevelDB writes one key–value pair per block body, but block-body workloads read contiguous block ranges. GroupChain raises the write granularity to match by making a *block group*—a cohort of S consecutive blocks—the atomic unit of both persistence and retrieval. This is safe because blockchain data is *temporally immutable*: block bodies are never modified after writing, so a physically contiguous group at write time remains contiguous indefinitely, with no compaction disturbing it.

a) Block-to-group mapping: Block N is deterministically assigned to group G_{id} at intra-group position I by:

$$G_{id} = \lfloor N/S \rfloor, \qquad I = N \bmod S. \tag{1}$$

This $O(1)$ arithmetic requires no auxiliary structure. The group size S trades per-query read amplification (smaller S) against metadata overhead and range-scan throughput (larger S).

b) Physical layout and the Dual In-Group Index: As shown in Figure 6, groups are appended sequentially to a flat binary file outside LevelDB, never subject to compaction. Each group begins with a compact header containing the *Dual In-Group Index (DII)*, written once at flush time and never modified. Because groups are variable in length, their file offsets are stored in ThermoList's L0 hash map at write time.

- **Block Index (B-Index):** a fixed-size array of per-block byte offsets within the group, enabling $O(1)$ intra-group block resolution:

$$\text{Addr}_{\text{block}} = \text{Addr}_{\text{group}} + \text{BIndex}[I]. \tag{2}$$

  The explicit array is necessary because block bodies range from $\sim$1 KB (early PoW era) to $>$200 KB (DeFi era).
- **Transaction Index (T-Index):** a compact open-addressing hash map from the first 8 bytes of each transaction hash to its intra-group byte offset:

$$\text{Addr}_{\text{tx}} = \text{Addr}_{\text{group}} + \delta_{\text{tx}}. \tag{3}$$

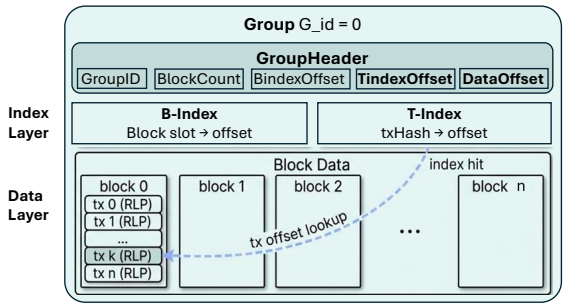

Fig. 6. GroupChain file format: variable-length groups appended sequentially, each prefixed with a DII header (B-Index and T-Index).

The T-Index enables $O(1)$ transaction resolution without loading the full group payload.

Both structures fit within the group header ($\lesssim 4$ KB) and are loaded into DRAM on first access; subsequent intra-group lookups are pure memory operations.

c) Write amplification and crash safety: Block and transaction data bypass LevelDB compaction entirely, achieving write amplification of exactly $1\times$ for this data class; MPT state data continues through LevelDB unchanged. Crash safety follows from the append-only structure: partial writes are detectable by file-size comparison, and any block whose LevelDB pointer record is present but whose GroupChain payload is absent is re-fetched and re-packed on restart.

C. ThermoList: Frequency-Aware Tiered Index

GroupChain removes spatial fragmentation but not the temporal dimension of the Read-Sink Effect: even with contiguous groups on disk, the storage layer cannot distinguish a group accessed once from one accessed thousands of times per second. ThermoList is a three-tier in-memory routing index that addresses this by adapting access depth to observed frequency. It operates as a pure routing layer—never replicating GroupChain data—so promotion and demotion are index-only operations whose cost is proportional to metadata, not data size.

- **L0 (base layer):** a hash map containing every block group, keyed by G_{id}. Each entry holds the group's `fileOffset` and an atomic access counter f_i. L0 is append-only and never touched by promotion or demotion, providing a permanent fallback for every group. The L0 footprint grows with chain length; in the unlikely event that it outgrows available DRAM, the overflow can be spilled to a compact on-SSD index, incurring one additional read for cold-group lookups without affecting correctness.
- **L1 (warm tier):** a bounded cache over active groups, each entry storing the DII header ($\lesssim 4$ KB). L1 presence reduces a full-group read to one targeted `pread` for the requested block only, eliminating the need to load the entire group. Groups with higher access frequency naturally receive more frequent maintenance passes and are prioritised for promotion to L2; groups with lower frequency are demoted back to L0.
- **L2 (hot tier):** pins the raw group payload in DRAM, serving queries with zero disk I/O. Capacity is bounded by a configurable DRAM budget. When full, ThermoList evicts according to the configured policy: a *frequency-aware* policy evicts the group with the lowest normalised score F_i (via a min-heap), resisting cache pollution from sequential scans; an *LRU* policy offers lower overhead.

a) Access frequency tracking: Each L0 entry's counter f_i is incremented on every read. To prevent stale high counters from blocking newly emerging hotspots, ThermoList normalises within a sliding window of W recent operations:

$$F_i = \frac{f_i - f_{\min}}{f_{\max} - f_{\min}}, \quad F_i \in [0,1]. \tag{4}$$

Window-relative normalisation allows a newly popular group to ascend the hierarchy within minutes of becoming hot.

b) Promotion and demotion: We provide two policies:

Policy A — Probabilistic: A background maintenance pass recomputes F_i and applies:

$$p_{\text{up}}(i) = \min\big(p_0 + \gamma \cdot F_i,\ p_{\max}\big), \quad F_i > T_{\text{up}}, \tag{5}$$

$$p_{\text{down}}(i) = \delta \cdot (1 - F_i), \quad F_i < T_{\text{down}}. \tag{6}$$

The probabilistic formulation resists transient spikes while allowing sustained hot groups to rise reliably.

Policy B — Threshold-based: A group is immediately promoted when its count exceeds a fixed threshold and demoted when it falls below a lower threshold— equivalent to Policy A in the limit of binary $p_{\text{up}} \in \{0,1\}$, with lower maintenance overhead.

c) Concurrent safety: ThermoList guarantees that a read for any registered group always terminates and returns the correct `fileOffset`, regardless of concurrent promotions or demotions. This follows from L0's append-only structure and the internal synchronisation of all tier data structures.

d) Residual limitation: ThermoList is reactive: a group is promoted only after its counter exceeds T_{up}. The very first access to a newly hot group still traverses L0 and issues a disk read. As quantified in §III, rapid hotspot shifts produce a measurable P99 miss storm until the counter rises. SemanticPredictor (§IV-D) eliminates this residual cost.

D. SemanticPredictor: Causal Proactive Promotion

ThermoList's reactive counter cannot promote a group before its first access. SemanticPredictor closes this gap by exploiting a causal regularity in Archive-node query patterns: transaction retrieval is structurally a two-phase lookup. The first phase reads a *transaction provenance record* (which block contains this transaction) and returns a block number; the second phase reads the *block-body record* at that number. These two phases are causally ordered—the provenance record exists solely to locate the block body—so observing a provenance read provides certainty that a block-body read for the resolved group is imminent. This is a *local, causal* regularity, not a statistical co-occurrence.

Algorithm 1 SemanticPredictor: per-query proactive promotion

Require: Intercepted record r at `ethdb` interface
1: **if** r is a transaction provenance record **then**
2: $blockNum \leftarrow$ DECODEBLOCKNUMBER(r)
3: $G \leftarrow \lfloor blockNum/S \rfloor$
4: PROACTIVEPROMOTE(G) ▷ Signal 1: causal
5: **else if** r is a block-body record **and** L0 miss on G **then**
6: $G \leftarrow$ DECODEGROUPID(r)
7: PROACTIVEPROMOTE($G-1,\ G+1$) ▷ Signal 2: adjacency

a) Two complementary signals: **Signal 1 — Transaction-provenance trigger (causal, near-100% accuracy):** When a provenance read is detected, the resolved block number is decoded immediately, G_{id} is computed, and that group's tier is proactively raised before the block-body read arrives. Because the provenance $\rightarrow$ body sequence is structurally invariant for all transaction retrieval operations, prediction accuracy approaches 100%.

Signal 2 — Adjacent-locality trigger (statistical): On an L0 cold miss for group G, ThermoList tiers of $G-1$ and $G+1$ are immediately raised. This exploits the sequential access locality quantified in §III: range-scanning indexers and reorganisation handlers consistently access neighbouring groups.

The two signals are complementary: Signal 1 covers precise transaction-targeted queries; Signal 2 covers range-oriented and exploratory access patterns.

b) Protocol: Algorithm 1 states the per-query logic.

c) Graceful degradation: A misprediction costs one index promotion plus a background demotion—negligible overhead. A false negative costs one L0 lookup plus one disk read, identical to plain ThermoList. SemanticPredictor can therefore only improve on ThermoList, never worsen it. Unlike Leaper [10], whose signal is correlational and fires at compaction time, Signal 1 is causal and fires one access ahead of the read it predicts—an orthogonal mechanism that targets index-traversal depth rather than the LevelDB block cache.

E. Component Interaction and Performance Decomposition

Table III decomposes each component's contribution, anticipating the ablation study of §VI-C2.

TABLE III
PERFORMANCE DECOMPOSITION BY COMPONENT. IMPROVEMENTS ARE RELATIVE TO THE PRECEDING ROW; MEASURED VALUES ARE IN §VI-C2.

Configuration	Improvement	Primary mechanism
Geth baseline	—	SSTable traversal to deepest level
+GroupChain	avg ↓, P99 ↓↓	k I/Os $\rightarrow$ 1 sequential read
+ThermoList	avg ↓↓	disk $\rightarrow$ DRAM routing
+SemanticPredictor	P99 ↓	first-access cold miss eliminated

GroupChain and ThermoList improve *average* latency through structural changes that benefit every query. SemanticPredictor's primary contribution is *tail* latency: it reduces P99 by eliminating the infrequent but expensive first-access cold miss that dominates the tail distribution. The component ordering confirms the expected dependency: ThermoList delivers a larger absolute average improvement, addressing the pervasive Read-Sink Effect; SemanticPredictor handles the narrower but otherwise unavoidable adaptation-lag problem.

V. IMPLEMENTATION

We implemented ThermoKV as a pluggable storage-layer extension to Go-Ethereum (Geth) v1.13, comprising approximately 700 lines of Go. ThermoKV intercepts only block-body writes and reads (keys with the `'b'` prefix); all other data classes—state trie nodes, block headers, receipts, and transaction indices—are forwarded unchanged to the underlying LevelDB instance, preserving full protocol compatibility.

TABLE IV
THERMOKV IMPLEMENTATION AND EVALUATION PARAMETERS.

Parameter	Symbol	Default	Description
Group size	S	25	Blocks per GroupChain group
L2 capacity	C_{L2}	384 MiB	DRAM budget for full-group payloads
L1 capacity	C_{L1}	128 MiB	DRAM budget for header indices
Look-ahead window	L_{ahead}	16	Observation window for Signal-2
Decay factor	λ	0.9	Co-occurrence matrix decay rate

GroupChain buffers incoming blocks in memory and flushes them sequentially in groups to a single append-only flat file; each flushed group is prefixed with a compact header containing a block-offset index (B-Index), enabling O(1) intra-group block lookup. **ThermoList** is implemented as a three-level in-memory structure: L0 is a lock-free hash map (`sync.Map`) that holds offset metadata for all registered groups; L1 is a byte-bounded LRU cache that stores group headers for one-shot targeted reads; and L2 is a separate byte-bounded LRU cache that pins full block-body payloads of the hottest groups. **SemanticPredictor** operates on the read path: Signal-1 is triggered synchronously upon resolution of a transaction-hash lookup, immediately promoting the target group's header into L1 before the block-body read arrives; Signal-2 promotes the headers of the two spatially adjacent groups ($G-1$ and $G+1$) on every L0 cold miss. The default configuration uses a total DRAM budget of 512 MiB, partitioned as 384 MiB for L2 and 128 MiB for L1; Table IV lists all tunable parameters.

VI. EVALUATION

Our evaluation addresses three research questions, one per design component:

- **RQ1 (§VI-B):** Does GroupChain's append-only layout reduce write amplification to $1\times$ for block data, and how does write throughput compare against LSM-based baselines?
- **RQ2 (§VI-C):** Does ThermoList resolve the Read-Sink Effect and deliver superior read throughput and tail latency across random and sequential workloads; what is the isolated contribution of each component; and does ThermoList adapt effectively when the access hotspot migrates?
- **RQ3 (§VI-E):** Does SemanticPredictor's causal Signal 1 eliminate block-body cold misses in two-phase transaction retrieval, and does Signal 2's adjacency prefetch reduce cold-miss overhead at group boundaries?

TABLE V
DATASET AND WORKLOAD CHARACTERISTICS.

Dataset (M4)		Workload	W-Write	R-Block	R-Range	R-Tx	H-series
Block range	0–7.95M[†]	Total requests	6.56 M blocks	1,000k	650k (win.)	1,000k	1,000k
Unique blocks	6,564,000	Granularity	block ingest	1 block	10 blocks	tx→block	1 block
DB size after import	∼96 GB	Distribution	sequential	tx-weighted	sequential	tx-weighted	phase-freq.
Avg. tx / block	53.6	Block range	0–7.95M	0–7.95M	1.4–7.95M	0–7.95M	P1/P2[‡]
GC groups ($S = 25$)	262,560	Source	RLP export	BigQuery	4 contracts[§]	BigQuery	R-Block freq.

[†] Range 5.0M–7.95M is sparsely sampled from available RLP exports; 6.564M unique blocks imported in total.
[‡] P1: 0–4M; P2: 4M–7.95M.
[§] R-Range scans start from early benchmarks (e.g., Augur at blk 1.4M) and traverse through modern contracts: CryptoKitties (blk 4,605,167) [27], USDT/Tether (4,634,748) [28], USDC (6,082,465) [29], and UniswapV1 Factory (6,627,917) [30].

A. Experimental Setup

We compare five systems under the same benchmark harness: Geth [11], Block-LSM [31], ChainKV [9], Leaper [10], and ThermoKV. All experiments run on an H3C UniServer R4900 G5 server with two Intel Xeon Platinum 8360Y CPUs (2.40 GHz, 36 cores each), 251 GiB DDR4 DRAM, and an NVMe SSD, running Ubuntu 22.04 LTS (Linux 5.15.0-78-generic). Table V summarises the M4 dataset and all five workloads used in this evaluation.

a) Write workload: The write workload replays the 6.564 M unique blocks of the M4 export in block-number order via `geth import`, emulating the sustained ingestion of an Ethereum Archive node performing historical synchronisation. This is the input to all subsequent read workloads and serves as the basis for measuring write amplification and compaction overhead.

b) R-Block: frequency-weighted random block lookup: R-Block models the block-retrieval pattern of blockchain explorers, where historically active blocks are queried far more often than quiet ones. Each block b is assigned sampling weight $w(b) = \max(\text{tx\_count}(b),\, 1)$ using transaction counts from the Google BigQuery public Ethereum dataset [26]. The 1 M-request workload is generated by weighted sampling with replacement over all 6.564 M unique blocks, so that densely populated blocks are queried proportionally more often while every block remains reachable. This is the primary read benchmark for latency and throughput.

c) R-Range: sequential indexer scan: R-Range models on-chain indexers that scan historical blocks sequentially from the deployment point of a major contract, extracting event logs. We select four landmark contracts whose deployment blocks fall within the M4 range: CryptoKitties (block 4,605,167), USDT/Tether (block 4,634,748), USDC (block 6,082,465), and UniswapV1 Factory (block 6,627,917). These contracts were chosen because they represent distinct DeFi/NFT epochs and their deployment events anchor real indexer access patterns. From each deployment block s, we generate scan windows $[s + 10k,\ s + 10k + 9]$ for $k = 0, 1, 2, \ldots$ until the dataset end at block 7.95 M, yielding approximately 650 K non-overlapping 10-block windows. The access sequence is deterministic with no random component, making R-Range a pure test of sequential spatial locality.

d) R-Tx: two-phase transaction lookup: R-Tx models the dominant query pattern of block explorers and DeFi analytics tools: retrieving a historical transaction by its hash. Each such query structurally follows two steps—a provenance read to resolve the block number, then a block-body read to extract the transaction—providing the causal signal that SemanticPredictor Signal 1 exploits (Section IV-D). Blocks are sampled from the set $\{b \mid \text{tx\_count}(b) > 0\}$ with weight $w(b) = \text{tx\_count}(b)$; a transaction index is then chosen uniformly from $[0,\ \text{tx\_count}(b) - 1]$. The 1 M-request workload is used to measure end-to-end transaction retrieval latency and to validate Signal 1 precision.

e) H-series: two-phase hotspot-shift: H-series evaluates system adaptation when the access hotspot migrates abruptly. The 1 M requests are divided into two equal phases of 500 K queries each. Phase 1 draws from the top-500 hottest block groups in the range 0–4 M, weighted by their R-Block access frequency. Phase 2 draws from the top-500 hottest groups in the range 4 M–7.95 M, a completely disjoint set ($\text{Jaccard}(H_1, H_2) = 0.00$). The shift occurs at query 500,001. We track cold-miss counts in sliding windows of 1 000 queries and compare ThermoList-only, full ThermoKV, and an Oracle that has perfect foreknowledge of H_2.

B. RQ1: Write Amplification Elimination

GroupChain writes each block group exactly once to an append-only flat file, bypassing the WAL → Memtable → SSTable → Compaction cycle entirely. The prediction is that WAF for block data equals 1.00 by construction.

TABLE VI
WRITE AMPLIFICATION.

System	TPS (K)	MB/s	WAF	I/O (GB)
ThermoKV	**213.9**	**418.6**	**1.00**	58.4
Geth (LevelDB)	52.4	99.3	2.81	159.8
Block-LSM	190.0	361.5	2.03	115.4
ChainKV	51.8	98.2	3.02	171.7
Leaper	48.5	91.9	2.81	159.8

Table VI confirms this. ThermoKV achieves WAF = **1.00** with write throughput of **213.9 K blocks/s** (418.6 MB/s). Among LSM-based systems, Block-LSM is the fastest writer at 190.0 K blocks/s but still incurs WAF = 2.03; Geth and Leaper record WAF = 2.81; ChainKV reaches 3.02. Relative to Geth, ThermoKV improves write TPS by **4.08**× and reduces total disk I/O by **63%**; against Block-LSM the I/O saving is **48%**. The WAFs of 2.03–3.02 for LSM systems reflect L0→L1→L2 multi-level merges already active at 56.86 GB scale; GroupChain's cost is constant at WAF = 1 regardless of dataset growth.

C. RQ2: Read-Sink Effect Resolution and Read Performance Improvements

This section answers three sub-questions: end-to-end read performance against baselines (§VI-C1), component-level attribution of the gains (§VI-C2), and dynamic hotspot adaptation (§VI-D).

1) End-to-end Read Performance:

a) Random block reads: Table VII and Figure 7 report a subset of 110,023 frequency-weighted R-Block queries against the 6.56 M-block database. ThermoKV achieves **22.4 K QPS**—**3.28**× Geth's 6.8 K—with P99 = **186** μ**s**, an **86%** reduction from Geth's 1,359 μs. Against the strongest read baseline, ChainKV (18.9 K QPS), the improvement is **19%** with P99 cut by **48%**. Leaper improves Geth by only 2.1× (14.1 K QPS), confirming that general per-level prefetching cannot substitute for index-depth reduction.

TABLE VII
RANDOM BLOCK READ PERFORMANCE.

System	QPS (K)	Avg (μs)	P50	P90	P99	P99.9
ThermoKV	**22.4**	**28.7**	**8**	**42**	**186**	**578**
Geth (LevelDB)	6.8	145.9	33	267	1,359	3,082
Block-LSM	14.8	67.0	14	174	334	486
ChainKV	18.9	52.3	21	154	356	899
Leaper	14.1	70.4	22	188	545	1,380

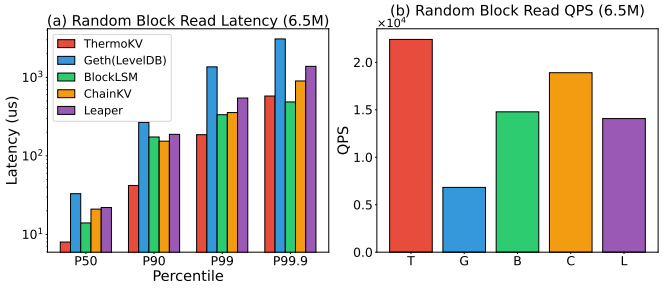

Fig. 7. Random block read: latency percentiles (left, log scale) and QPS (right) across five systems.

b) Sequential range queries: Table VIII and Figure 8 report 500,000 R-Range sequential windows. ThermoKV achieves **458.4 K QPS** with P99 = **8** μ**s**—**2.32**× Geth and **68%** above Block-LSM. The Tier-1 hit rate reaches 99.0%: GroupChain packs 25 consecutive blocks per group, so a scan over blocks N through $N+24$ reuses a single cached DII header, collapsing multiple random reads into one sequential access.

TABLE VIII
RANGE QUERY PERFORMANCE.

System	QPS (K)	Avg (μs)	P50	P90	P99	P99.9
ThermoKV	**458.4**	**1.7**	**1**	**2**	**8**	**126**
Geth (LevelDB)	197.9	4.5	1	4	36	480
Block-LSM	272.5	3.0	1	4	26	193
ChainKV	266.5	3.3	1	3	29	233
Leaper	185.8	4.8	1	4	37	434

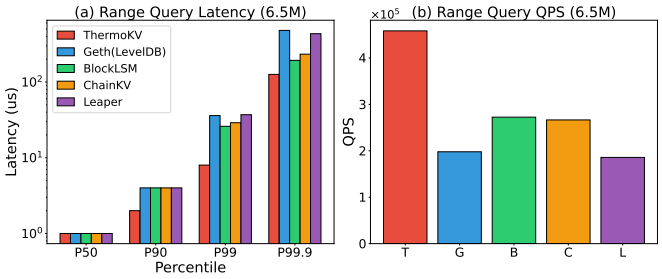

Fig. 8. Range query: latency percentiles (left) and QPS (right) across five systems.

2) Component Ablation: To isolate each component's contribution, we evaluate three ThermoKV configurations on the same 110,023 R-Block queries: (*i*) GroupChain only (no cache, no predictor); (*ii*) GroupChain + ThermoList (no predictor); (*iii*) Full ThermoKV with Signal 2 enabled. LevelDB is included as an external reference via direct `Get()` calls; it is not directly comparable to the Geth rows in Table VII because it bypasses Geth's benchmark harness.

TABLE IX
COMPONENT ABLATION ON RANDOM BLOCK READS.

Config	QPS (K)	Avg	P50	P99	L0	L1	L2
No Cache	7.4	134.2	66	585	110K	0	0
Cache Only	15.8	42.6	12	278	28K	73K	9.9K
Full	**22.4**	**28.7**	**8**	**186**	18K	83K	9.9K
LevelDB	9.7	102.1	26	1,135	—	—	—

Table IX shows incremental gains. *GroupChain alone* reaches only 7.4 K QPS—below LevelDB's 9.7 K—because each query reads an entire group (S=25 blocks) from disk to extract a single block, whereas LevelDB's built-in cache serves repeated accesses from DRAM; this confirms that ThermoList is essential for competitive read performance. *Adding ThermoList* lifts QPS to 15.8 K (**+114%**), cuts Tier-0 fallbacks from 110 K to 28 K (−**74%**), and achieves Tier-1 hit rate 66% plus Tier-2 hit rate 9%; the tier structure directly bypasses the LSM traversal depth. *Enabling Signal 2* further raises QPS to 22.4 K (**+42%** over Cache Only) via 31,604 proactive adjacent-group promotions, pushing Tier-1 to 75.4% and cutting Tier-0

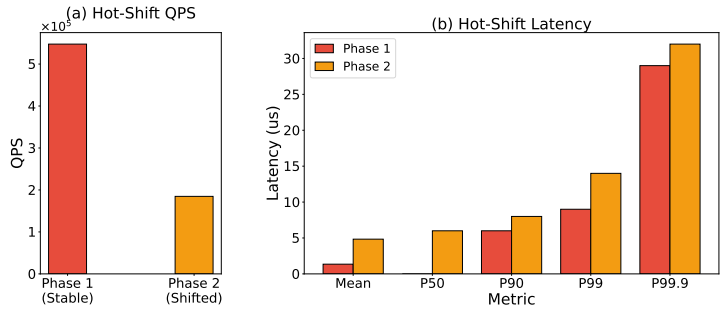

Fig. 9. Hot-shift adaptation curve: QPS and cache-tier distribution across Phase 1 (steady) and Phase 2 (shift).

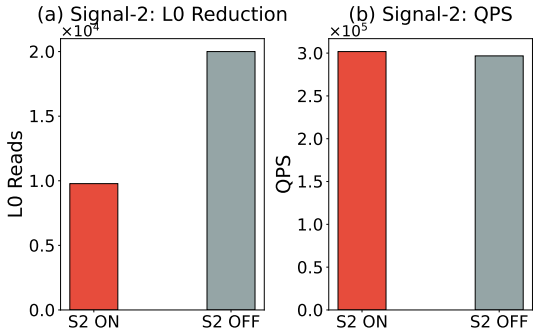

Fig. 10. Signal 2 ablation: L0 cold reads and QPS with adjacency prefetch enabled vs. disabled.

fallbacks to 18 K (**−36%**). The resulting P99 of 186 μs is **84%** lower than LevelDB's 1,135 μs.

D. RQ3: Hotspot-Shift Adaptation

The ablation above measures steady-state performance; this sub-section tests whether ThermoList dynamically repositions its tier assignments when the hot working set migrates abruptly.

We run the H-series workload. During Phase 1, ThermoKV operates at steady-state QPS = 547.5 K with 505 Tier-2 hits per window. At the phase boundary, the hotspot shifts to a completely disjoint block range (Jaccard = 0.00): Tier-2 hits collapse to 2 and QPS falls to 184.7 K—a 66% drop—because the new groups have not yet been promoted. ThermoList's sliding-window frequency normalisation then begins observing the new access pattern and promotes the incoming working set; Figure 9 shows the full adaptation curve and the recovery trajectory back toward steady state. This self-healing confirms that ThermoList's tier assignment is runtime-adaptive, not a static write-time decision as in Block-LSM and ChainKV.

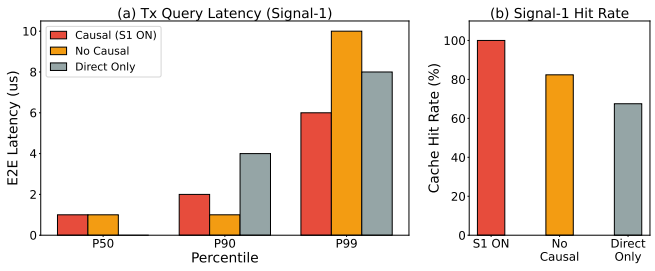

Fig. 11. Transaction query performance across three Signal 1 configurations.

E. RQ3: SemanticPredictor Effectiveness

SemanticPredictor comprises two complementary signals. Signal 1 fires on a transaction provenance read and pre-promotes the causally determined target group before the block-body read arrives (§VI-E1). Signal 2 fires on a Tier-0 cold miss and pre-promotes the two adjacent groups, exploiting sequential access locality (§VI-E2).

TABLE X
SIGNAL 1 RESULTS. "BODY" COLUMNS ISOLATE THE SECOND-STEP BLOCK-BODY READ LATENCY (μS).

Mode	E2E		Body		Hit%	QPS (K)
	Avg	P99	Avg	P99		
Causal (S1 ON)	0.92	6	**0.18**	**1**	**100%**	**703.4**
No Causal	0.89	10	1.94	18	82.4%	695.5
Direct Only	1.43	8	—	—	67.5%	513.4

1) Signal 1: Causal Transaction-Provenance Prediction: We evaluate three modes on the 1 M-request R-Tx workload: *Causal* (Signal 1 active), *No Causal* (ThermoList only, no Signal 1), and *Direct Only* (GroupChain Tier-0, no ThermoList, OS page cache may serve repeated reads).

Table X and Figure 11 show the results. Signal 1 achieves a **100%** Tier-1 hit rate for block-body reads—every provenance read deterministically resolves the target group, so every subsequent body read finds its group pre-staged. Average body latency falls from 1.94 μs (No Causal) to **0.18 μs** (**−90.7%**), and body P99 from 18 μs to **1 μs**. The end-to-end QPS gain is modest (703.4 K vs. 695.5 K, +1.1%) because the provenance read dominates the pipeline and already accounts for roughly 80% of E2E latency; the body read represents the remaining ≈20%, so even a 90% reduction in body latency yields limited E2E QPS impact on this workload. In production settings where clients issue dense transaction-by-hash lookups—DeFi analytics, block-explorer APIs—the body fraction is substantially higher and the absolute gain correspondingly larger.

The precision of Signal 1 is **100%** on R-Tx: every query issues a provenance read immediately followed by a body read for the same group. In production, a small fraction of provenance lookups may return `nil` (non-existent hashes); these do not constitute predictor failures, as the causal invariant holds for all valid transactions.

2) Signal 2: Adjacent-Group Locality Prefetch: Signal 2 promotes the DII headers of groups $G-1$ and $G+1$ into Tier-1 upon a Tier-0 cold miss for group G. We evaluate its contribution on the 500,000-window R-Range workload,

where sequential scans produce a deterministic stream of cold reads at group boundaries—making R-Range the cleanest workload to isolate Signal 2's effect without confounding it with ThermoList's frequency-driven promotion that dominates H-series behaviour.

Table XI and Figure 10 compare Signal 2 enabled vs. disabled. Enabling Signal 2 cuts Tier-0 cold reads from 20,001 to 4,218—a **78.9%** reduction—and lifts QPS from 296.8 K to **458.4 K** (**+54.5%**). The 4,506 proactive promotions are nearly all consumed: the precision is close to 100% for strictly sequential patterns. On random workloads with lower sequential locality the benefit is smaller, but any Tier-0 cold read at a group boundary still triggers a useful adjacent promotion.

TABLE XI
SIGNAL 2 ABLATION ON 500K RANGE QUERIES.

	L0 cold reads	L1 hits	S2 promotes	QPS (K)
S2 ON	4,218	494,941	4,506	**458.4**
S2 OFF	20,001	480,458	0	296.8
Cold-read reduction				**78.9%**
QPS improvement				**+54.5%**

F. Summary

Table XII collects the key results.

TABLE XII
SUMMARY OF KEY EVALUATION RESULTS.

Metric	Geth	Block-LSM	ChainKV	ThermoKV
Write path (RQ1)				
Write TPS (K)	52.4	190.0	51.8	**213.9**
WAF	2.81	2.03	3.02	**1.00**
Random block read (RQ2-A)				
QPS (K)	6.8	14.8	18.9	**22.4**
P99 (μs)	1,359	334	356	**186**
Range query (RQ2-A)				
QPS (K)	197.9	272.5	266.5	**458.4**
P99 (μs)	36	26	29	**8**
Prediction (RQ3)				
Signal 1 body latency (μs)		—		**0.18**
Signal 2 cold-read reduction		—		**78.9%**

Three findings stand out. **(1)** GroupChain achieves WAF = 1.00 while all LSM baselines incur 2–3× the disk I/O, confirming that the compaction pipeline is unnecessary for immutable block data. **(2)** ThermoList delivers the highest read throughput (22.4 K random QPS; 458.4 K range QPS) and lowest tail latency (P99 = 186 μs random, 8 μs range) across all systems; the component ablation shows each tier of the system contributes incrementally, and the H-series experiment shows ThermoList dynamically repositions its index on hotspot migration—behaviour unavailable in any baseline. **(3)** SemanticPredictor reduces block-body cold misses by 90.7% (Signal 1) and Tier-0 cold reads by 78.9% (Signal 2), collectively eliminating the adaptation-lag overhead at both hotspot transitions and sequential scan boundaries.

VII. CONCLUSION

Ethereum Archive nodes face a structural storage bottleneck that we term the *Read-Sink Effect*: LSM-tree compaction continuously sinks immutable block-body data to the deepest storage levels, making read latency proportional to data age rather than access frequency. Because write recency and read hotness are completely decoupled in Archive-node workloads—Jaccard similarity 0.00 between write-recent and read-hot block groups—no amount of per-level tuning or write-time data reorganisation can resolve this mismatch; both act before or within the compaction hierarchy, leaving the fundamental index depth unchanged.

This paper presents ThermoKV, a drop-in storage-layer extension for Geth that resolves the Read-Sink Effect through three co-designed components. **GroupChain** exploits the temporal immutability of block bodies to pack consecutive blocks into append-only flat groups outside LevelDB's compaction path, reducing write amplification to 1× for block data. **ThermoList** maintains a frequency-driven tiered routing index that dynamically repositions hot block groups to DRAM after compaction—the first such index that is both runtime-adaptive and fully compatible with Geth's Path-Based State Scheme (PBSS), a property that prior systems Block-LSM and ChainKV cannot claim. **SemanticPredictor** eliminates cold-miss bursts at hotspot transitions by exploiting two causal signals: the structural two-phase transaction retrieval invariant, which provides near-certain foreknowledge of imminent block-body reads, and adjacent-group locality, which anticipates sequential indexer access patterns.

Evaluated on 6.56 M real Ethereum mainnet blocks, ThermoKV achieves WAF = **1.00** versus Geth's 2.81, improving write throughput by **4.08**× with **63%** less disk I/O. On random block lookups, P99 latency falls from 1,359 μs to **186** μs (**−86%** versus Geth, **−44%** versus Block-LSM); sequential range-query QPS reaches **2.32**× Geth. SemanticPredictor's causal Signal 1 cuts block-body latency on transaction queries by **90.7%**; Signal 2 reduces cold reads by **78.9%**. These gains stem from a qualitative reduction in index traversal depth—from $O(L)$ SSTable levels to $O(1)$ DRAM lookups for hot groups—rather than from per-level efficiency improvements, confirming that the Read-Sink Effect requires a structural solution, not a tuning one.

REFERENCES

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

[2] D. Guo, J. Dong, and K. Wang, "Graph structure of the Ethereum transaction network," in *Proceedings of the IEEE International Conference on Applied Mathematics, Modeling and Computer Simulation (AMMCS)*, 2019.

[3] go-ethereum, "Archive mode," https://geth.ethereum.org/docs/fundamentals/archive, 2026, accessed: 2026-04-15; last edited February 24, 2026.

[4] J. Ren *et al.*, "Characterising ethereum archive-node storage: Write-path composition and access skew," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2025, to appear.

[5] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable key-value store," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 79–94.

[6] P. Xu, N. Zhao, J. Wan, W. Liu, S. Chen, Y. Zhou, H. Albahar, H. Liu, L. Tang, and Z. Tan, "Building a fast and efficient lsm-tree store by integrating local storage with cloud storage," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 3, pp. 1–26, 2022.

[7] H. Chen, C. Luo, Y. Wang, B. Zhang, E. Li, and W. Xu, "SpanDB: A fast, cost-effective LSM-tree based KV store on hybrid storage," in *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, 2021, pp. 17–32.

[8] Q. Wei, Z. Chen, X. Chen, Y. Zhang, X. Cai, Z. Jia, Z. Shen, Y. Wang, Z. Shao, and B. Li, "A semantic-integrated LSM-tree-based key–value storage engine for blockchain systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 4, pp. 1144–1157, 2024.

[9] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '24)*, 2024, pp. 226:1–226:23, also published in Proceedings of the ACM on Management of Data, 1(4), Article 226.

[10] L. Yang, H. Wu, T. Zhang, X. Cheng, F. Li, L. Zou, Y. Wang, R. Chen, J. Wang, and G. Huang, "Leaper: A learned prefetcher for cache invalidation in lsm-tree based storage engines," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 1976–1989, 2020.

[11] go-ethereum contributors, "go-ethereum: PebbleDB storage backend," https://github.com/ethereum/go-ethereum, 2024, accessed 2024.

[12] J. Ren *et al.*, "Characterising Ethereum archive-node storage: Write-path composition and access skew," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2025, to appear.

[13] W. Zhong, C. Chen, X. Wu, and J. Huang, "REMIX: Efficient range query for LSM-trees," in *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, 2021, pp. 51–64.

[14] Y. Matsunobu, S. Dong, and H. Lee, "MyRocks: LSM-tree database storage engine serving Facebook's social graph," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3217–3230, 2020.

[15] D. Teng, L. Guo, R. Lee, F. Perez-Sorrosal, S. Zhang, H. Chen, and X. Zhang, "LSbM-tree: Re-enabling buffer caching in data management for mixed reads and writes," in *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 68–79.

[16] Y. Dai, Y. Xu, A. Ganesan, R. Alagappan, B. Haas, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "From WiscKey to Bourbon: A learned index for log-structured merge trees," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020, pp. 765–783.

[17] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016, pp. 133–148.

[18] P. Raju, R. Souza, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau, "mLSM: Making authenticated storage faster in Ethereum," in *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2018.

[19] S. Yang *et al.*, "SolsDB: A high-performance storage engine for Ethereum archive nodes," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2024.

[20] H. Wang *et al.*, "AppendChain: Eliminating write amplification in Ethereum storage via append-optimised column families," in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2026, to appear.

[21] Erigon contributors, "Erigon: Ethereum implementation on the efficiency frontier," https://github.com/erigontech/erigon, 2024, accessed 2024.

[22] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains," in *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*, 2017, pp. 1085–1100.

[23] C. Zhang, C. Xu, H. Hu, and J. Xu, "COLE: A column-based learned storage for blockchain systems," in *Proceedings of the 22nd USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, 2024, pp. 329–345.

[24] H. Feng, Y. Hu, Y. Kou, R. Li, J. Zhu, L. Wu, and Y. Zhou, "Slimarchive: A lightweight architecture for ethereum archive nodes," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, 2024, pp. 1257–1272. [Online]. Available: https://www.usenix.org/conference/atc24/presentation/feng-hang

[25] K. Li, Y. Tang, J. Chen, Z. Yuan, C. Xu, and J. Xu, "Cost-effective data feeds to blockchains via workload-adaptive data replication," in *Proceedings of the 21st ACM/IFIP International Middleware Conference*, 2020, pp. 371–385.

[26] A. Day and E. Medvedev, "Ethereum in bigquery: a public dataset for smart contract analytics," https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics, Aug. 2018, google Cloud Blog. Accessed: 2026-04-14.

[27] C. Klinkmüller, A. Ponomarev, A. B. Tran, I. Weber, and W. M. P. van der Aalst, "Mining blockchain processes: Extracting process mining data from blockchain applications," in *Business Process Management: Blockchain and Central and Eastern Europe Forum*, ser. Lecture Notes in Business Information Processing, vol. 361. Springer, 2019, pp. 71–86.

[28] Etherscan, "Ethereum transaction hash," https://etherscan.io/tx/0x2f1c5c2b44f771e942a8506148e256f94f1a464babc938ae0690c6e34cd79190, 2017, uSDT contract-creation transaction; block 4634748. Accessed: 2026-04-15.

[29] Graph Node Documentation, "Amp-powered subgraphs," https://mintlify.com/graphprotocol/graph-node/advanced/amp-subgraphs, 2026, official Graph Node documentation; USDC example uses address 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48 with startBlock 6082465. Accessed: 2026-04-15.

[30] The Graph Protocol, "uniswap-subgraph/subgraph.yaml," https://github.com/graphprotocol/uniswap-subgraph/blob/master/subgraph.yaml, 2026, official Uniswap V1 subgraph manifest; factory contract 0xc0a47dFe034B400B47bDaD5FecDa2621de6c4d95 uses startBlock 6627917. Accessed: 2026-04-15.

[31] 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 *2021 IEEE 39th International Conference on Computer Design (ICCD)*. IEEE, 2021, pp. 25–32.