https://crad.ict.ac.cn
63(3): 668−684, 2026

AppendChain: 一种面向区块链的高效键值存储系统

王锦江[1] 张 晓[1,2,3] 苏琳萱[1] 于锦阳[1] 黄志杰[1] 甘一伟[1] 何赫烺[1] Michael Ngulube[1]

[1](西北工业大学计算机学院 西安 710072)

[2](大数据存储与管理工业与信息化部重点实验室(西北工业大学) 西安 710072)

[3](空天地海一体化大数据应用技术国家工程实验室(西北工业大学) 西安 710072)

(jinjiangwang@mail.nwpu.edu.cn)

AppendChain: An Efficient Key-Value Store System for Blockchain

Wang Jinjiang[1], Zhang Xiao[1,2,3], Su Linxuan[1], Yu Jinyang[1], Huang Zhijie[1], Gan Yiwei[1], He Helang[1], and Michael Ngulube[1]

[1](*School of Computer Science*, *Northwestern Polytechnical University*, *Xi'an* 710072)

[2](*Key Laboratory of Big Data Storage and Management* (*Northwestern Polytechnical University*), *Ministry of Industry and Information Technology*, *Xi'an* 710072)

[3](*National Engineering Laboratory for Integrated Aero-Space-Ground-Ocean Big Data Application Technology* (*Northwestern Polytechnical University*), *Xi'an* 710072)

Abstract Modern blockchain systems face increasing storage pressure due to the continuous growth of on-chain data volume. Most systems adopt LSM-tree-based key-value storage engines as their persistence layer, which maintains intra-level key order through compaction. However, blockchain data are highly heterogeneous, characterized by structurally diverse and unordered keys. This results in frequent compaction operations that cause write amplification and degrade I/O efficiency. To address these issues, we propose AppendChain, a storage optimization framework that reorganizes the write path and key-value layout in the storage engine without altering Ethereum's execution semantics. AppendChain introduces three techniques: 1) Type Separation: partitioning heterogeneous key types into dedicated column families to minimize keyspace overlap; 2) Big Value Separation: redirecting large values to external blob storage to avoid redundant rewrites during compaction; 3) Sorted Merkle Patricia Trie (MPT): applying structure-aware and order-aligned key encoding to enhance the locality of state keys. We implement AppendChain on top of Go-Ethereum (Geth) and integrate it with RocksDB, modifying the storage stack without changing Ethereum's consensus and provenance logic. Evaluations on mainnet workloads show that, compared with Geth, AppendChain improves storage engine throughput (OPS) by up to 28 times, reduces write amplification rate nearly 94.24%, and reduces compaction frequency and time 98.75% and 72.19% respectively, significantly enhancing read/write efficiency under high-throughput transactional workloads. This framework effectively improves the storage performance of blockchain full nodes.

Key words blockchain; Ethereum; MPT; LSM-Tree; key-value storage engine; storage optimization

摘 要 现代区块链数据规模持续增长,节点存储压力不断上升。主流区块链系统通常采用基于LSM-Tree的键值存储引擎作为持久化层,其特性是通过合并操作维护键值在同一层级的有序性。然而,区块链数据具有显著的异构特征,包含大量结构不一与无序的键类型,使得存储引擎频繁触发合并操作,带来

收稿日期: 2025-06-17; **修回日期**: 2026-01-14

基金项目: 国家重点研发计划项目 (2022YFB2702101); 自然基金重大集成项目 (92152301); 国家自然科学基金项目 (62272394)

This work was supported by the National Key Research and Development Program of China (2022YFB2702101), and the Major Integrated Project of the Natural Science Foundation (92152301), and the National Natural Science Foundation of China (62272394).

读写放大现象和 I/O 效率下降等问题。针对上述问题，提出了 AppendChain，在不改变以太坊执行语义的前提下，重新组织数据在存储引擎中的写入路径与存储布局，实现 3 项优化：1）类型分离。将异构键类型划分至独立的列族，最小化键空间重叠。2）大值分离。将大值剥离到外部 Blob 文件存储，避免在合并过程中的重写。3）排序 MPT。通过引入树结构感知、顺序对齐的键编码，增强状态键的局部性。AppendChain 在 Go-Ethereum（Geth）的基础上实现，并集成了 RocksDB，修改了存储栈而未改变以太坊的共识与溯源功能。在主网工作负载上的实验评估表明，与 Geth 相比，AppendChain 将存储引擎吞吐量（OPS）提升了 28 倍，将写放大率降低约为 94.24%，将合并操作次数与耗时分别降低了 98.75% 与 72.19%，在高吞吐量交易工作负载下显著提高了读写效率。该方案显著提升了区块链全节点的存储性能。

关键词 区块链；以太坊；MPT；LSM-Tree；键值存储引擎；存储优化

中图法分类号 TP311

DOI: 10.7544/issn1000-1239.202550482 **CSTR:** 32373.14.issn1000-1239.202550482

区块链作为一种去中心化、不可篡改的分布式账本技术，通过共识机制保证多方的数据一致与信任协同，已在金融[1]、供应链[2-3]、物联网[4-5]等领域得到广泛应用。作为新质生产力之一，区块链推动数据存储向更安全、更高效方向演进。以太坊（Ethereum）[6]作为全球最活跃的区块链平台之一，不仅支持数字资产的转账，还具备图灵完备的智能合约执行能力，推动了去中心化应用[7]的蓬勃发展。

然而，随着交易规模持续增长，以太坊全节点的存储压力愈发严重[8-9]。当前全节点需存储全部区块数据、交易信息、状态快照等内容[10]，截至 2025 年 6 月，数据体量已超过 22 TB[11]，其数据量随时间的增长如图 1 所示。新节点加入时需要同步所有区块，重新执行所有交易并更新状态树，执行完毕后将区块数据、状态数据等持久化到存储层[12]。我们观察到同步超 200 万个区块后，节点实际引发的磁盘读写已超过 400 GB 和 425 GB，而存储的数据只有 25.6 GB，读写放大率分别达到 15.6 和 16.6。因此，区块同步过程给数据存储带来了极大的压力。

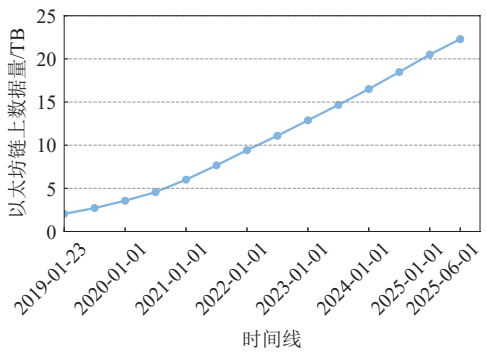


Fig. 1 Ethereum full node on-chain data size

图 1 以太坊全节点链上数据大小

以太坊节点运行时的主要工作包括交易打包、状态更新、数据持久化等[6,13]。网络中的节点根据在共识中的角色进行分工，包括提议节点（proposer）和验证节点（validator）等[6]。提议者需要按照时间顺序从内存池中选择一批交易依次执行。执行过程通过读取和修改状态树中的账户信息，验证交易合法性，完成状态更新。全部交易处理完毕后，系统重新计算状态根哈希并写入区块头，形成完整区块后，将其广播至全网。验证节点在收到区块后，同样执行区块中的所有交易，更新状态树；重新计算根哈希，和区块头中的哈希进行比对。比对一致则认为区块有效，并将区块数据、新增的状态数据、元数据等转化为键值对，持久化到底层存储引擎中，完成一次共识过程。

在上述流程中，交易执行是节点运行和数据同步的核心过程，而对状态树的查询、更新是 I/O 开销的主要来源[14]。以太坊使用默克尔哈希前缀树（Merkle patricia trie, MPT）[6]来维护账户状态信息。MPT 在叶子节点存储账户状态，通过中间分支节点压缩账户地址，并提供哈希校验功能。每次账户状态查询需沿着一条从根到叶的路径解析多个树节点。而 MPT 的平均深度达 8.6[9]，在缓存未命中的情况下，读取一个账户状态平均需触发近 9 次数据库访问。加之，截至 2025 年 6 月，以太坊主网账户数量超 3 亿[15]，远超内存缓存能力。据统计，处理一个包含 100 项交易的区块需要执行超过 1×10^4 次的随机 I/O（高 100 倍），这在企业级的 NVMe SSD 上也需要数百毫秒[16]。执行完交易后，系统需持久化 MPT 节点。而树节点采用哈希值作为键、节点内容作为值，这会导致 MPT 节点间的树结构语义在持久化时被抹去，存储引擎只能将其视为独立的键值项处理[10,17]。

以太坊在持久化层采用日志合并树（log-structured merge tree, LSM-Tree）[17-20]架构的键值存储引擎。

LSM-Tree 在内存表(MemTable)暂存数据,随后批量刷盘为有序字符串表(sorted string table, SST)文件。系统在后台定期执行合并操作(Compaction),将低层级 SST 合并、排序后写入高层级 SST,维护层内键值的有序性。

深入分析以太坊的数据类型和交易执行流程,我们将以太坊低吞吐、高读写放大率归因为:区块链键值无序性与 LSM-Tree 的排序机制之间的不匹配[10]。具体来说,现有存储机制存在 3 类主要问题:1)链上各类键值混合写入同一列族,导致键空间重叠,触发大量 Compaction 操作。2)大值在合并时需要读出并重写,拉高 I/O 带宽压力和磁盘写入延迟[21-23]。3)状态数据的哈希键导致存储引擎频繁触发合并操作,进行全范围的键排序,形成指数级增长的排序成本与读写放大效应[16]。

已有研究从多方面尝试优化区块链存储:1)通过在单节点存储部分数据(如只存储区块头[24-26]、用哈希替代区块体中的完整交易[27]等);或通过多节点协作分布存储数据(如利用纠删码分布[28-30]、提出分片机制[31-35]等)。但这类方法通常会牺牲节点数据的完整性,削弱系统的去中心化程度和信任保证能力,同时还会引来更加复杂的数据划分问题。2)优化状态认证结构,如 LVMT[36], COLE[37], SlimArchive[9] 等方案。但这类方法都会破坏了原有的 MPT 结构,与当前以太坊状态验证机制不兼容。3)结合以太坊的数据语义进行优化。如 Block-LSM[10] 提出了对状态数据加共享前缀,ChainKV[16] 提出了状态分离与空间竞争缓存等。但是它们并没有对数据做细粒度的划分,并且状态前缀仅包含区块号,没有完全消除存储引擎的排序过程。

本文提出了 AppendChain,一种面向以太坊全节点的存储优化框架,在不改变以太坊共识逻辑与状态语义的前提下,重构链上数据在存储引擎中的存储布局与写入流程。AppendChain 引入 3 个机制:1)类型分离。根据语义类别,将不同类型键值划分至多个独立列族,消除键空间重叠,提升合并效率。2)大值分离。将大值剥离至 Blob 文件存储,避免在合并时反复重写。3)排序 MPT。设计树结构感知的键编码机制,使同一棵 MPT 中的节点在键空间中具有聚集性,优化合并行为并提升读效率。

我们基于 Go-Ethereum(Geth)[38] 实现了 AppendChain,集成了 RocksDB[19] 替代默认的 LevelDB[20] 作为底层存储引擎,按照键值类型构建了多列族[39]存储模型,实现了 Blob 文件[23]的写支持以及状态树的排序键生成与解析逻辑。我们在主网历史区块数据[40]与模拟高频交易负载[16,21]下对方案进行了评估。与 Geth 相比,AppendChain 显著减少了存储引擎的合并开销与写放大效应,并将吞吐量(单位为 OPS(operations per second))提升 28 倍。AppendChain 从根本上解决了区块链写负载无序性与存储引擎维持有序特性的不匹配。本文的主要贡献总结如下:

1)系统地分析了当前以太坊全节点在数据存储中存在的瓶颈,归因于异构键值混合存储与 LSM-Tree 维持有序的合并机制间的不匹配;

2)提出了 AppendChain,一种面向区块链负载的类型感知与顺序保持的存储优化框架,通过 3 项机制(类型分离、大值分离、排序 MPT)优化数据的写入路径与空间布局;

3)基于 Geth 实现 AppendChain,集成了 RocksDB,完整实现了多列族写入、Blob 存储及树结构感知的 MPT 编解码方案,保持对以太坊的兼容;

4)在主网数据与模拟负载下验证了本文方案效果,结果显示在吞吐量、写放大率、合并操作等核心指标上均显著优于当前实现的方案。

1 背 景

1.1 区块链

区块链是一种去中心化、不可篡改的分布式账本技术,通过共识机制使多个不可信节点在无中心化协调的情况下达成对数据状态的一致认知[41-44]。区块链的核心数据结构是一条由区块组成的链,每个区块包含一组有序的交易,并在区块头中记录前一个区块的哈希值,实现可验证的链式连接。每当矿工或验证者构建出新区块并将其广播到网络中,全网节点将对该区块进行验证,并通过共识机制达成一致[6,13]。区块链系统通常包含共识层、网络层与存储层[43,45],分别对应区块生成的共识过程、点对点通信与数据持久化。其中,存储层的设计直接影响系统的可扩展性与查询效率。

以太坊[6]是一个开源的区块链平台,自 2015 年上线以来,已成为最具影响力的区块链基础设施之一,为去中心化金融[1]、非同质化代币(non-fungible token, NFT)[46]、去中心化自治(decentralized autonomous organization, DAO)[47]等众多应用提供平台支持。以太坊不同类型节点根据资源能力差别承担不同的数据维护职责[40,48]。全节点(full node)保存完整的区块链数据,包括所有交易记录、账户状态及合约信息,

能够独立完成交易验证与智能合约执行；轻节点(light node)仅保存区块头信息，无法独立验证状态，需依赖全节点提供服务[48]。本文主要围绕全节点展开研究与优化分析。

以太坊全节点执行工作[6,48]流程如图 2 所示。全节点(验证节点)在接收到新区块后，首先解析并验证区块头信息，然后依序处理区块中的每笔交易[6,13]。交易处理由以太坊虚拟机(Ethereum virtual machine, EVM)[6]负责，EVM 在执行过程中通过读写状态树 MPT 来完成状态访问与更新。具体地，EVM 会在每笔交易开始时，从状态树中读取交易发起方的账户状态，包括余额、nonce 等信息，判断其是否满足执行条件。验证通过后，EVM 执行交易，根据结果更新状态树，例如转账引发的余额增减等[6]。当前区块所有交易执行完毕后，节点对更新后的状态树重新计算根哈希，并与区块头中声明的状态根进行比对，一致则认为区块合法。最终，节点将区块数据、更新后的状态数据、元数据等转化为键值对，存入底层键值存储引擎中[6]，形成长期可查询、可验证的链上数据。

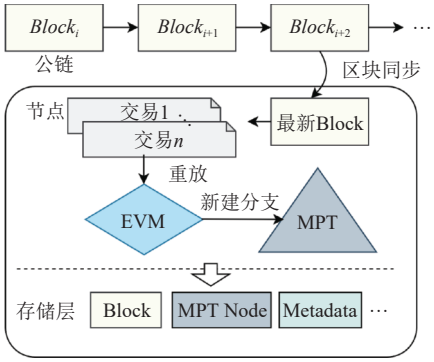


Fig. 2 The Ethereum workflow

图 2 以太坊运行工作流程

1.2 以太坊存储概述

以太坊存储体系承载着完整的链上历史，包括交易记录、账户状态与系统元信息等，所有链上数据均以键值对的形式存储于底层存储引擎中[10,12]。如图 3 和表 1 所示，按语义功能划分，数据主要分为 4 类：区块数据、状态数据、收据与索引数据以及区块与区块链上元数据[10]。

1)区块数据。每个区块包含 2 部分：区块头和区块体[6]。区块头记录相关的元信息，如父区块哈希、状态树根哈希、交易树根哈希、收据树根哈希、时间戳和难度等；区块体则保存本区块内所有交易的列表。以太坊将交易与收据组织为默克尔树结构，可用于单个交易/收据级别的完整性验证。

2)状态数据。以太坊状态管理的核心数据结构是 MPT[6]。MPT 结合了默克尔树的可验证性与前缀树的路径压缩特性，将所有账户状态信息存储于叶子节点，利用分支节点压缩账户地址。一个 MPT 示例如图 4 所示。MPT 节点分为 3 类：①扩展节点(extension node)。用于路径压缩，包含路径片段和指针。②分支节点(branch node)。含 16 个子指针(对应十六进制 0-f)和可选值。③叶子节点(leaf node)。记录账户剩余地址和账户状态。

所有节点内容经 Keccak-256 哈希[6]，结果作为父节点的引用，构成自顶向下的哈希验证路径。当状态变更时，MPT 自底向上新建路径上的节点，并依次重新计算哈希，最终生成新的状态根哈希，写入区块头，作为当前区块状态的唯一标识。

3)收据与交易索引。为支持交易结果溯源与事件查询，以太坊记录每笔交易的执行状态、Gas 消耗及日志信息，形成交易收据[38]，并构建收据树来进行哈希聚合与验证。同时系统引入交易索引机制[10,38]，支持通过交易哈希定位其所在区块。其键由“l”+ *hash*(*tx*)组成，值为该交易所在的区块号。

4)区块与区块链元数据。区块元数据包括区块号到哈希、哈希到区块号的映射和区块总难度，便于

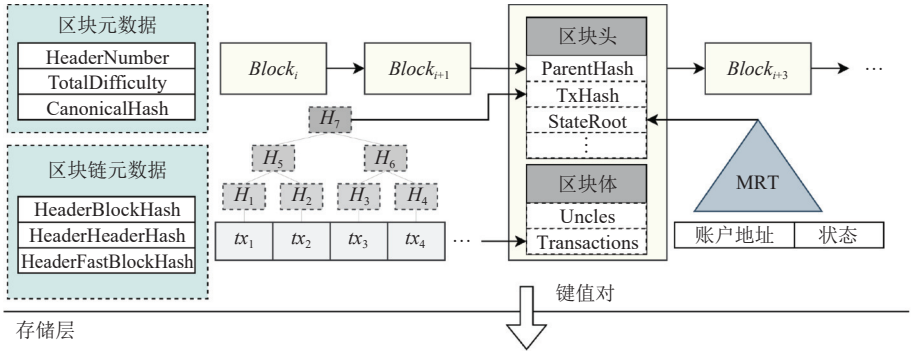


Fig. 3 Overview of blockchain data organization format

图 3 区块链数据组织格式概览

Table 1 Key-Value Structure and Functional Classification of Ethereum Storage Layer

表 1 以太坊存储层的键值结构与功能分类

类别	内容	键	值
区块数据	区块体	“b”+Num+Hash	body
	区块头	“h”+Num+Hash	header
状态数据	状态原镜像	“secure-key-”+Hash	preimages
	状态树节点	Hash	node
收据与交易索引	交易收据	“r”+Num+Hash	receipt
	交易索引	“l”+Hash	Num
区块元数据	哈希→区块号	“H”+Hash	Num
	区块总难度	“h”+Num+Hash+“t”	TotalDiffculty
	区块号→哈希	“h”+Num+“n”	Hash
区块链元数据	最新区块体哈希	“LastBlock”	Hash
	最新区块头哈希	“LastHeader”	Hash
	快速同步最终区块	“LastFast”	Hash

查找区块。区块链元数据维护链的全局状态，包括当前区块高度、链总难度、当前状态树根等。这些元数据在区块同步、链重构与分叉判断中起到重要作用[6,38]。

1.3 LSM-Tree 架构

主流区块链网络节点在存储层均采用 LSM-Tree[17] 架构的键值存储引擎，例如比特币[24]和以太坊[6]均默认使用 LevelDB[20]。LSM-Tree 的优势在于密集写，特别适用于以追加写为主的负载，其架构如图 5 所示。

LSM-Tree 的基本机制[17]为：新写入的键值首先缓存在内存表中，并追加记录于写前日志（write-ahead log, WAL）用于崩溃恢复。内存表达到阈值后被刷写为有序磁盘文件 SST 并存储于第 0 层（L_0），各层级（L_0~L_n）以容量和时间顺序组织。为维护数据顺序并避免数据冗余，LSM-Tree 定期在后台执行合并操作，将多个低层级 SST 文件合并、去重并排序后写入高层级 SST 文件，提升查询效率并控制空间占用[18]。

在理想追加写场景下，若新生成的 SST 文件的键区间与高层文件无重叠，RocksDB 等引擎支持将该文件直接迁移至高层级而无需合并排序，即轻量迁移合并（trivial move compaction）[19]。该操作仅更新元数据指针，几乎无 I/O 成本，从而显著减少写放大率并提高写入吞吐。在该场景下，存储引擎仅需以最小代价维护其多层级数据的完整性与读取效率。但在数据无序的情况下，存储引擎就需要频繁地执行后台合并操作对数据进行排序。

因此，LSM-Tree 架构天然适用于具有顺序追加语义的键值，如区块数据和区块元数据等。但当键值分布高度无序时（如状态数据），频繁的合并操作将带来显著的性能开销。

2 动 机

随着以太坊数据量的持续增长，底层 LSM-Tree 存储引擎在处理异构链上数据时暴露出显著的性能瓶颈。在同步过程中，我们观察到了极大的存储引擎读写开销。图 6 展示了同步到 25 万、50 万、100 万、200 万区块时，存储引擎刷盘、SST 删除、SST 创建数据量的变化。根据 LSM-Tree 的合并逻辑，删除表示合并前被读取的数据量，创建则是合并后重新写入的数据量。可以明显看出，随着数据量的增长，存储引擎出现了指数级增长的读写开销。在同步到 200 万区块时，系统的读写放大率已经分别达到 15.6 和 16.6，反映出高频合并所带来的显著性能开销。

以太坊不同类型数据在写入语义、访问模式、键分布与数据体积上差异明显，而当前统一混合存储策略无法适应 LSM-Tree 的特性，导致 Compaction

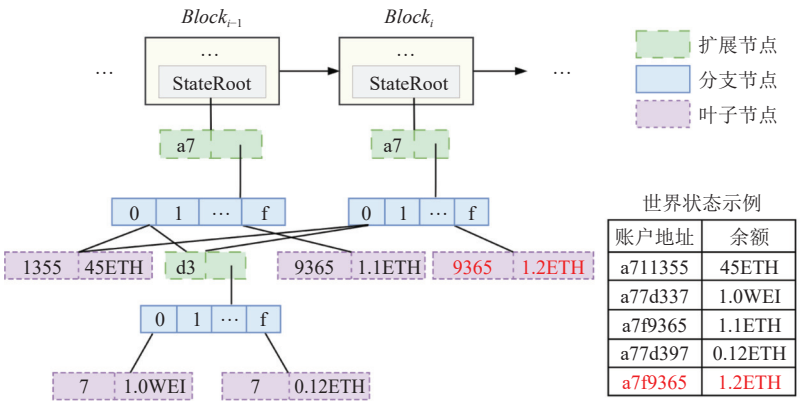


Fig. 4 Example of a MPT

图 4 1 个 MPT 示例

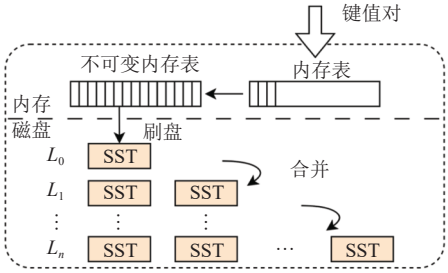


Fig. 5 Storage engine based on LSM-Tree architecture

图 5 基于 LSM-Tree 架构的存储引擎

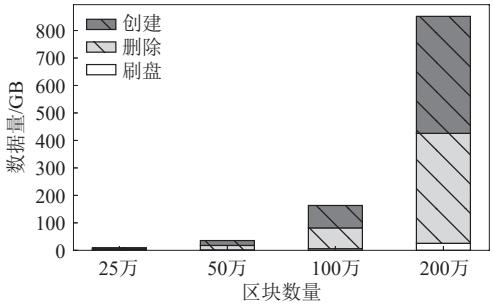


Fig. 6 Data volume of flushing, SST remove, and SST create of storage engine during block synchronization

图 6 区块同步期间存储引擎的刷盘、SST 删除和 SST 创建数据量

发生频繁、读写放大率和延迟显著增加[10,16,21]。本节围绕区块链数据特征，从键类型分布、哈希无序性与大值更新代价 3 个方面分析当前架构的问题来源。

2.1 异构数据混合存储引发排序开销

以太坊全节点在处理新区块时，会集中写入多种数据类型，包括区块头、区块体、交易收据、区块元数据等。这些数据通常采用“前缀+区块号+哈希”的格式构造键值，呈现出明显的顺序追加特性，即同一类键值在键空间中具有递增性。理论上，这类数据非常适合 LSM-Tree 的写入机制，若能保证写入顺序与键顺序一致，则可避免键区间重叠，从而触发存储引擎的轻量迁移合并，实现低成本的文件跨层迁移，无需排序与重写。

然而，当前以太坊节点采用统一列族将区块数据、状态数据等混合存储。这种存储策略打破了顺序键的连续性。由于新区块的多类数据会在同一时刻被批量写入，导致几乎每次刷盘的 SST 都存在键范围重叠。该模式迫使存储引擎执行标准合并操作，对所有相关文件进行排序、合并和重写。同时该合并过程还在区块链持续同步的过程中反复触发，加剧合并开销。

此外，区块链中还包含一些键固定但值频繁变化的数据（如最新区块体哈希、最新区块头哈希等区块链元数据项）。其键不变，但每个新区块同步都需重写这些项，导致旧值残留在多个 SST 文件中。在后续的合并过程中，这类键值需要被去重，带来合并成本。

更进一步地，混合存储还引入了来自状态数据（MPT 哈希键）的干扰。这些键分布无序，可能插入到已排好序的任意键区间，迫使原本排序好的数据又需要重新排序，并且这个过程还在不断进行，I/O 压力持续增加，形成指数级的资源消耗。

综上所述，统一列族的混合写策略破坏了顺序键的追加语义，使原本可优化的写路径退化为标准合并流程。系统不仅无法触发轻量迁移合并，还需面对因键重叠与键重复带来的额外 I/O 成本。若能将各类顺序追加数据独立存储，可显著减少合并频率，释放 LSM-Tree 架构的顺序写性能潜力。

2.2 大值重写引发读写放大

在 LSM-Tree 架构中，系统写入性能不仅受键的顺序影响，也受限于数据的大小。以太坊全节点在运行时频繁写入大值数据项，如区块体、状态树分支节点等，这些数据的平均大小超过 500B。

由于 LSM-Tree 的合并机制要求对多个 SST 文件进行读出、排序并重新写入新的文件，即便大部分键值未发生修改，只要处于同一 SST 中，所有大值也必须被重复读取与写入[22-23]。

在包含大值的场景下，这种全量排序带来新的代价。1）I/O 占用：大对象在合并时被重复搬移，即便内容不变也需完整读写，拉高磁盘带宽与延迟。2）内存消耗：大值在排序与合并阶段对内存的需求较大，但是存储引擎的缓存空间是固定的，当大值占用缓存后，系统在内存中能处理的数据量变小，吞吐量下降。3）写入阻塞：存储引擎后台合并被大值拖慢时，会进入写停顿状态（write stall），前台写入也会被同步阻塞，影响以太坊节点的区块产出与同步速度。

2.3 哈希键无序性引发高频合并

以太坊采用 MPT 存储状态数据，每个节点的键由其内容通过 Keccak-256 哈希生成。这种哈希键能提供一致性与抗篡改能力，但其结果在全局键空间中近似随机分布，抹除了原本的树结构语义，缺乏任何节点关联特征。这种无序性与底层存储引擎基于字典序排序的合并策略之间存在根本性冲突，导致一系列性能瓶颈。

以太坊在交易执行过程中，EVM 需对 MPT 完成一系列状态变更操作。首先遍历 MPT 路径，定位需更新的账户节点；对该节点创建 1 个副本，修改节点

数据(如余额、nonce 等);自底向上对相关节点创建副本,更新每层节点的哈希值;生成新的状态根,写入区块头。在内存中,MPT 保持着严格的树状结构,状态更新过程也遵循树路径的链式修改机制。但在刷盘阶段,所有更新的节点都会被转换为一个独立的键值项写入存储引擎。

在写入时,这些哈希键仍被批量写入同一个内存表,尚具一定的聚集性;但一旦触发合并,由于引擎需维护全局键序,这些逻辑相关的键会按字典序分散至多个 SST 文件中。随着数据量不断增长,新写入的数据与旧数据之间的键空间重叠逐步扩大,引擎需要频繁地全范围扫描、排序和去重,形成指数级增长的排序成本与读写放大效应。

更为严重的是,这种结构不仅影响写入,还削弱了读效率。尽管 MPT 的访问具备明显的路径局部性,但是存储引擎重排序后,读取时却要跨多个 SST 文件、多次随机 I/O 来查找节点内容,并且读入内存的 SST 也只有极少数键值被使用,显著增加磁盘 I/O 压力。

综上所述,当前以太坊全节点的键值存储体系中,顺序键未被有效利用,哈希键带来高排序成本,大值对象造成严重写放大效应。而存储引擎默认的合并策略无法根据键特性进行动态优化,进而形成系统性的性能瓶颈。

3 AppendChain 设计

针对上述问题,本文提出一种优化方案 AppendChain,在不改变以太坊执行语义的前提下,重新组织数据在 LSM-Tree 引擎中的写入路径与存储布局,通过类型分离、大值分离和排序 MPT 方案,减少了合并的次数和开销,降低写入放大,提升写入效率,并提升整体系统在数据同步和交易处理时的存储性能。

3.1 总体设计

AppendChain 基于现有 LSM-Tree 存储引擎,通过引入分离模块,在数据进入内存表之前完成分类处理。其整体设计如图 7 所示,包含三大模块:1)类型分离。根据写入数据的键语义,将不同类型数据分别划分至不同列族。具体而言,将顺序写入类键值划分至各自的列族,无序写入类键值独立维护为无序列族,实现顺序数据的纯追加写入,无序数据的自合并,减少合并开销。2)大值分离。将大尺寸的值剥离至独立 Blob 文件存储,通过指针的方式在 SST 中存储引用,避免其在合并过程中频繁参与数据重写,降低 I/O 压力。3)排序 MPT。对传统 MPT 引入树结构感知的键编码机制,增强键的局部顺序性,消除 SST 文件间的键空间重叠,增强系统的读写性能。

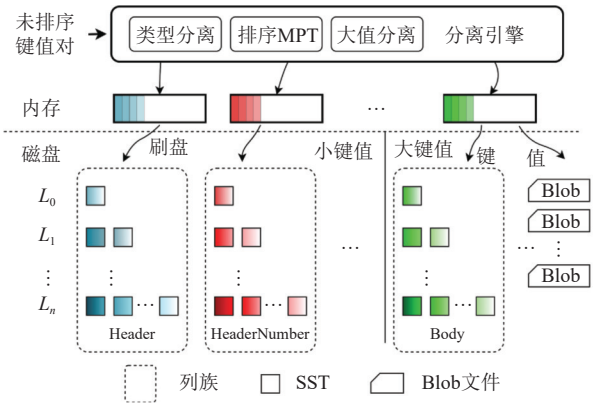


Fig. 7 Architecture of AppendChain

图 7 AppendChain 架构

通过上述模块协同工作,AppendChain 在不修改区块链原有一致性验证机制的前提下,实现了对键值的语义感知与写路径分离,缓解了 LSM-Tree 引擎在区块链场景下合并开销高的问题。

3.2 类型分离

在传统以太坊节点的存储流程中,所有顺序数据与无序数据共用一个列族,导致键空间重叠,引发排序开销与写放大等问题,降低系统的写入效率。

为此,AppendChain 引入了基于键值语义的类型分离机制,从源头上优化数据布局。该机制表现为在数据进入内存表前进行分类,将链上数据按其类型划分至多个独立的列族。首先,对于区块头、区块体、收据、区块元数据等具有明确顺序追加语义的数据,分别写入各自的列族。这类数据键空间不重叠,写入路径稳定,可使合并操作退化为轻量迁移合并,极大减少 I/O 压力,消除重写成本。其次,对于状态树节点等无序哈希键值则划入专属的状态列族,避免影响其他键空间,并结合后续的顺序追加写优化,降低了全局排序的复杂度。此外,对于无法进行排序优化的交易索引数据,也单独划分一个列族,确保了无序数据在相对较小范围内完成排序合并,避免参与全局排序。对于其他数据,则被存储在默认列族中。

3.3 大值分离

以太坊链上存在海量大尺寸键值对,如区块体数据(含交易列表)、状态树节点等。在传统 LSM-Tree 架构中,键值对被批量存入 SST 文件,合并时即便不涉及相关数据,仅新旧数据键空间发生重叠,就需将 SST 整体读出与重写,带来极高的 I/O 成本。

为此，AppendChain 引入了大值分离策略，将超过设定阈值的大值脱离主 LSM-Tree 路径，单独存储于 Blob 文件中。原键值对所在的 SST 中只保留一个指向 Blob 文件的引用指针，记录大值的位置。当合并操作发生时，仅合并键及值的指针。大值仅在内容更新时才被重写，小值继续参与排序与合并。这一机制有效减轻了大值在非必要情况下的频繁重写带来的 I/O 开销。

传统的键值分离策略在刷盘时分离[22-23]，而在写前日志（WAL）中已经预先刷盘过 1 次，带来了 2 次写开销。参考 BVLSM[49] 的观点，我们可以将 WAL 的 1 次写入有效利用，将 WAL 设计为不再循环删除，而是将其视为 Blob 文件维护，设计文件创建、写入等过程。对于该文件我们采用顺序追加方式写入，维护独立的刷盘过程，无需合并与垃圾回收过程，提升存储的灵活性。我们将 BVLSM 应用到 AppendChain 中，进一步带来了性能优化。

通过对大值的剥离，存储引擎在合并时不再强制读取与重写大值内容，合并操作更加轻量，显著降低了写放大率和磁盘写入延迟。

3.4 排序 MPT

以太坊状态数据通过 MPT 结构组织与存储，每个状态树节点的键均为哈希值（keccak256(Node)）。当状态数据持久化时，这些哈希键应分布在多个 SST 文件中，每次合并都需进行高开销的排序与重写操作，导致写放大效应加剧、合并延迟上升、系统吞吐下降。

为解决上述问题，我们提出一种键排序编码方案——排序 MPT，如图 8 所示。在保持原有哈希校验功能的同时，方案引入基于树结构顺序的键编码方案，消除哈希键的无序性，降低 SST 文件之间的键空间重叠。

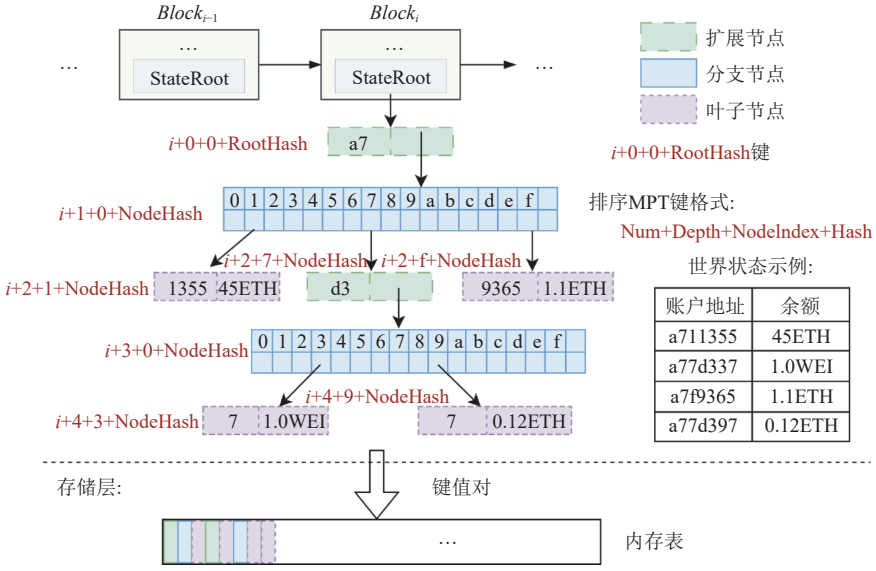


Fig. 8 Schematic diagram of sorted MPT

图 8 排序 MPT 示意图

键排序编码方案描述的是排序 MPT 并不改变 MPT 的树结构与哈希验证逻辑，而是在键生成过程中引入语义辅助信息，将原始哈希键扩展为包含树结构顺序的复合键。具体编码格式如下：Key=BlockNumber+Depth +NodeIndex+Hash。其中，BlockNumber 为当前区块高度，Depth 为当前节点在 MPT 中的深度，NodeIndex 为节点在当前深度层的横向索引号，Hash 为节点的哈希值。排序 MPT 将键按结构有序排列，形成键段聚集效应，使得同一区块中相关状态树节点的键在字典序上具备相邻性，LSM-Tree 更容易聚集相近键至同一或相邻 SST 文件。这种写入模式完全符合顺序追加特性，消除了合并开销。在状态访问过程中，系统需要从 MPT 根开始自顶向下解析路径。排序 MPT 使得读取一个完整的状态路径所涉及的节点通常已经聚合在同一或相邻 SST 文件中，大幅减少了读取时对多层级、多 SST 文件的访问需求。

我们设计的新键引入了 10B 额外的存储开销，包括 BlockNumber 的 8B 和 Depth, NodeIndex 的各 1B。我们认为，相较于哈希所使用的 32B，以及该方案带来的完全消除排序重写开销，这是可以接受的。

需要强调的是，该优化机制仅作用于存储引擎

的键布局逻辑，不会改变原有的 MPT 树生成逻辑和哈希校验逻辑。所有原始哈希键均可从编码中恢复，状态验证与合约执行流程保持完全兼容，属于透明的底层存储优化策略。

4 AppendChain 实现

我们基于 Geth v1.9.25 实现了 AppendChain 的全部功能，相关代码①已开源。

Geth[38] 的整体架构包括共识层、网络层、存储层等。共识层的 EVM 负责交易执行，并在内存中维护 MPT 树结构；执行后，对 MPT 中新建的节点提交底层数据库。涉及提交区块数据、状态数据等的接口在 core/rawdb 包中。存储层提供对底层键值数据库访问的接口，具体在 ethdb 包中。该包提供对 LevelDB 和 PebbleDB[50] 连接的实现，Geth 默认采用 LevelDB。我们在其中实现了对 RocksDB 连接的封装，遵循 ethdb/database.go 定义的接口标准，实现了打开数据库、增删查改、遍历、批量写入等功能。完成 RocksDB 与 Geth 的集成是我们的核心工作之一。

1)类型分离。我们在 core/rawdb 模块中实现了类型分离机制，将不同类型的链上数据(见 1.2 节)划分至不同的列族。具体来说，我们创建了以下列族：headers, bodies, receipts, tds, metadatas, stateNode headerNumbers, canonicalHashes, statePreimage,default。在上层不同类型数据的写入函数中调用 RocksDB 的 PutCF 接口来指定目标列族。读操作也做了相应适配，确保逻辑上与 Geth 兼容。

2)大值分离。我们集成了 BVLSM[49] 方案实现了大值分离。对以太坊交易的重放日志进行统计，大值(>128B)的条目所占比例为 80%~90%。因此，在 RocksDB 中，配置在写前日志时期写 Blob 文件，将设定分离阈值参数值为 128B。经过分析，该参数值可以将所有元数据、区块头等小值保留在 SST 中，对于大的区块体，状态树节点等划分至 Blob 文件。

3)排序 MPT。为支持 AppendChain 提出的排序 MPT 结构，我们对共识层至存储层之间涉及状态读写的全部路径函数进行了更新。具体涉及的模块包括 core/state, core/rawdb, trie/node 等。

在 trie/node 包中新增了节点类型 NewKey，该类型在功能上与 hashNode 相似，用于存储 MPT 中间节点引用。在状态树构造和遍历过程中，节点序列化函数支持自动判断并生成该类型的编码格式；相应地，解码函数中增加了对 NewKey 类型的识别与重构逻辑。

在 Geth 中，当 1 笔交易被 EVM 执行时，首先会在内存中的 MPT 上应用状态变更，并将所有被修改的节点标记为“脏节点”。随后，系统会进入提交(commit)阶段。集成新键的节点提交如算法 1 所示。该过程从根节点向下递归遍历，对于每个节点，如果它是全节点，则创建 1 个新全节点对象来收集子节点的新键。随后对所有子节点依次调用 commit。对 commit 传递的回调函数的功能是：将子节点生成的新键存入新全节点对象，并通过回调函数继续向上层传递该新键。待所有子节点处理完毕后，按“Key=BlockNumber+Depth +NodeIndex+Hash”生成当前节点的新复合键，并将其与新的节点编码后的值一起提交至存储层。此时，如果是全节点还需要再遍历 1 遍所有子节点，发现子节点是 NewKey 类型时，表示这是本次交易执行没有修改过的前序 MPT 节点，直接将其值赋给新全节点对象，链接前序 MPT 节点。最后，如有回调函数，则向上层返回当前节点的新键。通过上述过程，在节点提交数据库时实现排序 MPT 中新键的生成与存储逻辑。

算法 1. Sorted MPT 节点递归提交算法。

输入：*db*, *hash*, *num*, *depth*, *nodeIndex*, *callback*;

输出：本节点及其子节点以新键写入底层存储。

① *function commit*(*db*, *hash*, *num*, *depth*, *nodeIndex*, *callback*); /*从 MPT 根节点自顶向下递归遍历，将所有被修改的脏节点通过新键提交底层数据库*/

② if(*hash* 不在 *db.dirties* 中)

③ return;

④ end if

⑤ *node*←*db.dirties*[*hash*];

⑥ if(*node* 类型为 *fullNode*)

⑦ *nFNode*←新建 *newFullNode*{};

⑧ end if

⑨ for all (*childHash*,*idx*) in *node.forchilds*():

⑩ *commit*(*db*, *childHash*, *num*, *depth*+1, *idx function*(*newKey*){

⑪ if (*nFNode* 存在)

⑫ *nFnode*[*idx*]=*newKey*;

⑬ end if

① https://github.com/ WatermelonKnife/AppendChain

```
⑭   if (callback 存在)
⑮      callback(newKey);
⑯   end if })
⑰ end for
⑱ newKey← {num, depth, nodeIndex, hash};
⑲ if (nFnode 存在)
⑳   for all (childHash, idx) in node
㉑      if (child 类型是 *NewKey)
㉒         nFnode[idx]=child;
㉓      end if
㉔   end for
㉕   writeTrieNode(newKey.Bytes(),nFnode.rlp());
㉖ else /*nFNode 不存在*/
㉗   WriteTrieNode(newKey.Bytes(),node.rlp());
㉘ end if
㉙ if callback 存在
㉚   callback(newKey);
㉛ end if
```

5 实验评估

为了评估 AppendChain 在以太坊真实主网数据下的实际效果，本节设计并实施了一系列实验，围绕如下问题展开分析：AppendChain 是否能提升节点底层键值存储引擎的操作吞吐量；系统是否有效降低了写放大率与 Compaction 的触发频率和耗时；各个优化机制分别带来哪些性能收益；整体架构在真实数据下是否大幅提升读写性能及稳定性和减轻存储引擎负担。

随后 5.1~5.3 节将从实验环境配置、工作负载构建、测试流程及对比基线等方面展开具体介绍，并对不同优化策略的性能影响进行详细分析。

5.1 环境设置

1）环境。我们在一台机器中部署并测试了 AppendChain 原型系统。在所有区块链节点中，交易执行都是完全相同的，对于不包含网络情况的存储性能对比实验，我们在一台机器中重放交易执行过程，能够模拟真实的区块链存储负载。这种测试方法也与文献 [9−10,16,21,37] 类似。该机器配备 Intel 12th Gen Core i7-12700 处理器，包含 12 个核心与 20 个线程，搭配 32 GB DDR4 2 666 MHz 内存。磁盘为 2 块 8 TiB 的 HGST HUS728T8TAL SATA 企业级硬盘，其中一块用于读取原始区块数据，另一块用于测试过程中的数据写入，避免数据读操作对结果造成干扰。操作系统为 Ubuntu Server 20.04.6 LTS，Geth 的版本为 1.9.25。存储引擎采用我们集成的 RocksDB[50]，版本为 7.10.2。

2）工作负载。①写测试。我们将以太坊主网通过 full sync 模式[40] 同步前 200 万个区块，并提取包括区块头、区块体等链上数据作为实际重放负载。为了准确复现全节点写入行为，我们实现了区块重放机制：系统从 LevelDB 原始数据库中依序读取历史区块，通过 EVM 执行交易，更新状态树，重构链上状态，并将所有数据写入新的 RocksDB 后端，形成同等的数据副本。整个过程中记录所有延迟与 RocksDB 内部性能统计信息。该过程模拟了真实交易执行过程中的存储负载，确保了实验对区块链写入逻辑的一致。我们选用第 1 号区块至第 200 万号区块作为输入，具备高度代表性。②读测试。由于以太坊并没有开源的读负载序列，并且长期在主网检测读负载是较难实现的，因此本文采用常规的读测试方法，通过合成负载来评估系统的读性能。

3）评估指标。我们对写吞吐量、合并次数与耗时、写放大率、长尾延迟、交易执行时间、读吞吐等指标进行观测。每项实验独立运行，结果均重复采样 3 次并取均值。

4）基线与对比项。为了分析每一项优化机制的独立贡献与协同效果，我们使用如下对比基线与演化版本：

① Geth (LevelDB)。Geth 默认实现，使用原生 LevelDB 作为键值存储引擎。

② RocksDB。仅将底层存储替换为 RocksDB，其他保持不变。

③ +Big Value Separation。在 RocksDB 基础上集成 BVLSM 大值分离。

④ +Type Separation。在大值分离的基础上引入类型分离。

⑤ +Sorted MPT（AppendChain）。进一步集成排序 MPT 等同于集成所有优化的 AppendChain 完整系统。

5.2 写性能

1）吞吐量。图 9（a）展示了以太坊底层存储引擎在不同优化策略下处理不同规模区块（25 万至 200 万块）时的写入吞吐量。实验结果表明，原生 LevelDB 实现的吞吐量始终处于最低水平，为 21 kOPS~48 kOPS，写入性能随规模明显降低。将存储引擎替换为 RocksDB 后，吞吐量提升至 510 kOPS，带来 10.8~24.3 倍的吞吐提升。引入大值分离后，吞吐量提升至 560 kOPS；引入类型分离后，性能略有降低，这由

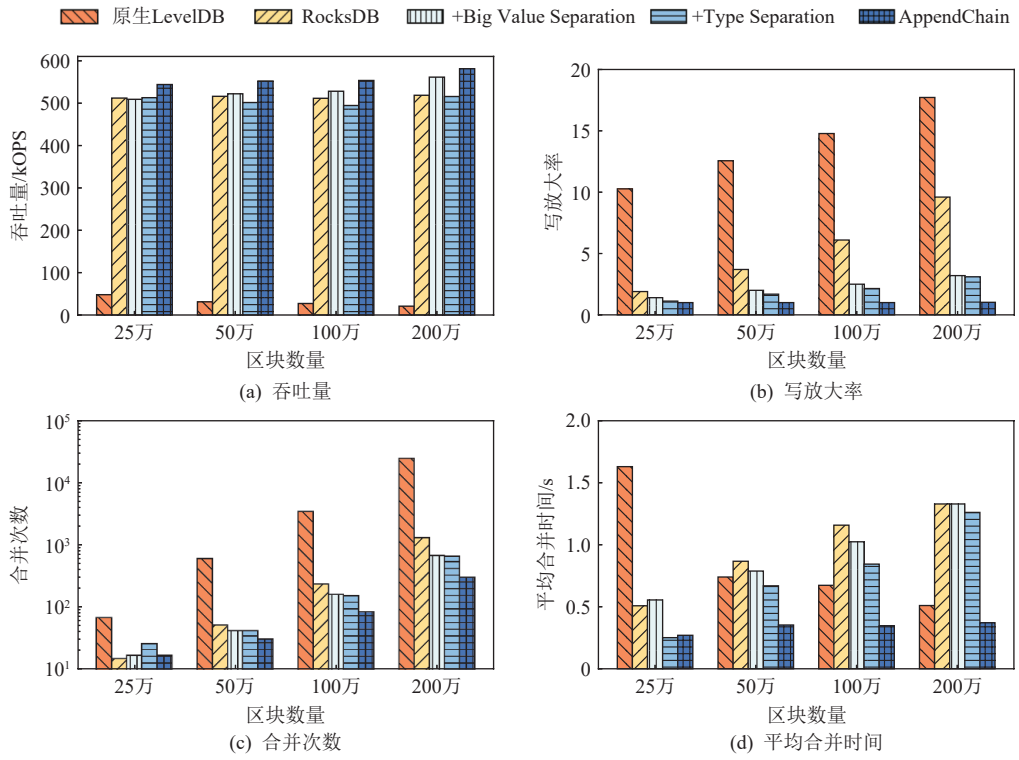


Fig. 9 Performance comparison of Ethereum key-value storage engine under different optimizations

图 9 以太坊键值存储引擎在不同优化下的性能比较

于对每类数据进行精细化管理带来了轻微的开销；引入排序 MPT 后，吞吐量最终提升至 580 kOPS 左右，相较于原生 LevelDB 带来 7.1~28 倍的性能提升。说明结构化键编码带来的键空间局部性得到优化，显著提升了写效率。

吞吐量数量级的提升主要归因于 RocksDB 的引入，而 AppendChain 各优化策略在此基础上对写入路径与键布局进行语义感知重构，实现了额外性能收益。

2)写放大率。图 9(b)展示了写放大率变化的趋势。结果显示，原生 LevelDB 的写放大率随着数据规模的扩大而显著升高，从 10.27 增长至 17.72。将存储引擎替换为 RocksDB 后，写放大率降低了约 45.8%。引入大值分离后，写放大率进一步降低，最大降低幅度达到约 81.9%，表明其成功规避了大值数据在 Compaction 中的重复搬移成本。继续启用类型分离，写放大率略微下降，说明其能避免数据跨类型合并的操作。最终集成排序 MPT 机制后，写放大率稳定在近似 1.0 的理想状态，最大值不超过 1.024。表明排序 MPT 成功保持了键的顺序性与聚集性，显著减少了存储引擎合并过程的重写成本。

AppendChain 的多项优化措施协同作用，显著缓解了 LSM-Tree 写放大问题。即便在 RocksDB 这一优化后的基础上，AppendChain 仍实现了最多约 89.3% 的写放大率降低，并可稳定至近似 1，说明精细化数据管理带来的效果非常显著。

3)合并性能。图 9(c)(d)分别展示了合并操作次数与平均耗时。原生 LevelDB 的合并操作次数远高于其他配置，在区块为 200 万时达 2.4 万次，且呈指数增长趋势。虽然单次合并时间较低(0.51~1.63 s)，但总合并成本极高。将存储引擎换为 RocksDB 后，合并次数显著下降至 15~1 302，平均合并耗时为 0.5~1.3 s，整体开销大幅下降。引入大值分离后，合并次数进一步下降，最大仅 676，合并时间维持在 0.55~1.32 s，变化平稳。说明将大键值脱离主键路径，单个 SST 能容纳更多的键值，能够减少合并次数与开销。引入类型分离后，合并次数在小规模数据上略微上升，其原因为多列族写入路径分离导致各内存表独立刷盘。但平均合并时间下降至 0.25~1.26 s，说明数据间干扰减少，合并更轻量。最终集成排序 MPT 后，合并次数降至最低，最大仅为 301；合并时间始终低于 0.38 s 且极为稳定。

AppendChain 集成所有优化后，存储引擎仍存在合并操作，但绝大多数都是轻量合并，不存在实际数据迁移，大幅减少了合并负担。

4)交易执行。图 10 展示了原生 LevelDB 与 AppendChain 在不同规模数据下的每一个交易平均

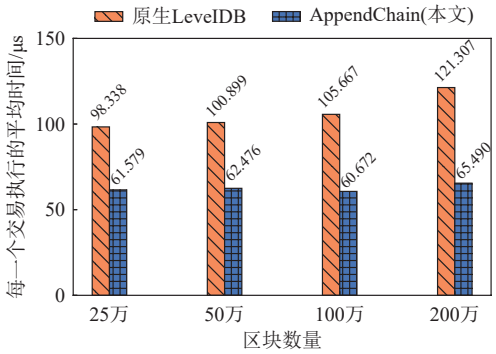


Fig. 10 Comparison of average transaction execution time on Native LevelDB and AppendChain

图 10 原生 LevelDB 与 AppendChain 平均交易执行时间对比

执行时间。实验结果显示，原生 LevelDB 的平均交易执行时间表现出明显的随规模增长而减少的趋势。而 AppendChain 始终保持在较低水平，平均执行时间稳定在 60~65 μs，最高下降幅度达 49%。

交易执行过程包括执行前读取状态树（MPT）存储的账户状态信息，以及执行完成对新增状态树节点的写回数据库。当目标账户节点未命中内存缓存时，系统需从磁盘中读取对应的叶子节点，并在状态更新后写回。AppendChain 通过排序 MPT 优化，增强相关联状态树节点在磁盘中的聚集性，在状态树遍历和更新过程中带来更高的缓存命中率和更少的磁盘访问次数，改善了交易执行过程中因 DB 读写性能带来的延迟。

5）长尾延迟。图 11 展示了原生 LevelDB 与 AppendChain 在不同分位点下的长尾延迟表现。结果表明，AppendChain 在所有分位点均实现显著的写入延迟降低，平均延迟相较原生 LevelDB 降低约 50%。进入高分位区间，原生 LevelDB 延迟陡增，P99 延迟达 1.1 s，表现出显著的长尾延迟膨胀现象。相比之下，

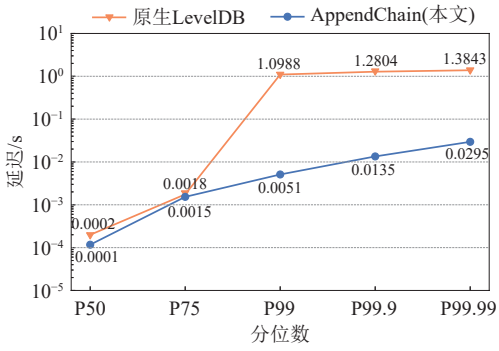


Fig. 11 Comparison of Native LevelDB and AppendChain on long-tail latency percentiles

图 11 原生 Level DB 与 AppendChain 的长尾延迟百分数对比

AppendChain 在高分位点延迟控制良好，P99 延迟仅为 5.1 ms，P99.99 延迟为 29.5 ms。与原生 LevelDB 相比，AppendChain 的 P99 延迟降低了 99.95%，P99.99 延迟降低了 97.87%。AppendChain 通过各项优化，减少了 Compaction 操作时长，进而大幅降低 I/O 拥塞时间。

6）数据分布。图 12 展示了 AppendChain 在 200 万区块数据集上，不同列族的数据体积与写放大率的分布情况。实验表明，具备天然的顺序追加特性的列族，写放大率均为 1.0 的理想状态，表明将其分离存储完全消除了 Compaction 操作引发的数据重写。metadatas 列族存储固定键的键值，写放大率仅为 1.1，表明几乎所有更新操作在内存中完成。状态数据 stateNode 占比最大，原本是写放大的主要来源。引入排序 MPT 后，将节点键编码全追加键，成功将写放大率优化至 1.0。statePreimage 列族用于存储交易索引，映射交易所在的区块，无法直接优化，但通过分离写入路径仍可缓解其与其他数据的混写干扰，使写放大率控制在 2.0 以内。此外，以太坊中其他未归类的数据被归入 default 列族，写放大率为 1.5，但占用空间极小，对系统整体的影响可忽略不计。

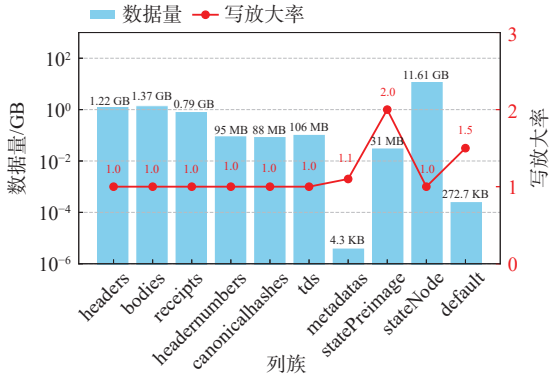


Fig. 12 Data size and write amplification of column families when AppendChain is under the size of 2MB

图 12 AppendChain 在 2MB 块大小下各列族的数据大小和写放大率

总体来看，AppendChain 通过类型感知的分离存储与结构化排序键设计，有效减少了数据重写。同时，所有具有顺序追加特性的列族，即便数据量增长，写放大率依然为 1，可预测 AppendChain 在数据量持续增长下仍能保持高效的存储。

5.3 读性能

由于以太坊主网缺乏公开的真实读负载，本文采用 3 种典型的合成读负载：随机分布（random）、扰乱的 Zipf 分布（scrambled zipfian）以及倾斜的最新 Zipf 分布（skewed latest zipfian）。这些负载分别模拟

了均匀访问、热点位置扰乱访问以及偏向最近写入数据的典型实际使用场景。每种负载的读请求个数占全数据库的约 2%, 并且测试时长超过 10 min, 确保测试压力足够评估系统性能。

图 13 展示了原生 LevelDB 对比 AppendChain 在 3 种负载下存储引擎的读吞吐量。结果表明, AppendChain 在各种负载下均优于原生 LevelDB, 带来 9.56~12.68 倍的吞吐量提升。在倾斜最新 Zipf 分布中, 将全部是区块读负载的吞吐量提升至最大 168 kOPS, 全部是状态读负载的吞吐量提升至 91 kOPS。随着负载的逐渐集中(从随机到倾斜最新 Zpif), 系统读性能逐渐提高, 这由于存储引擎缓存到内存中的 SST 会命中更多的键值, 带来小幅度的性能提升。AppendChain 通过各项优化, 显著带来读性能提升。

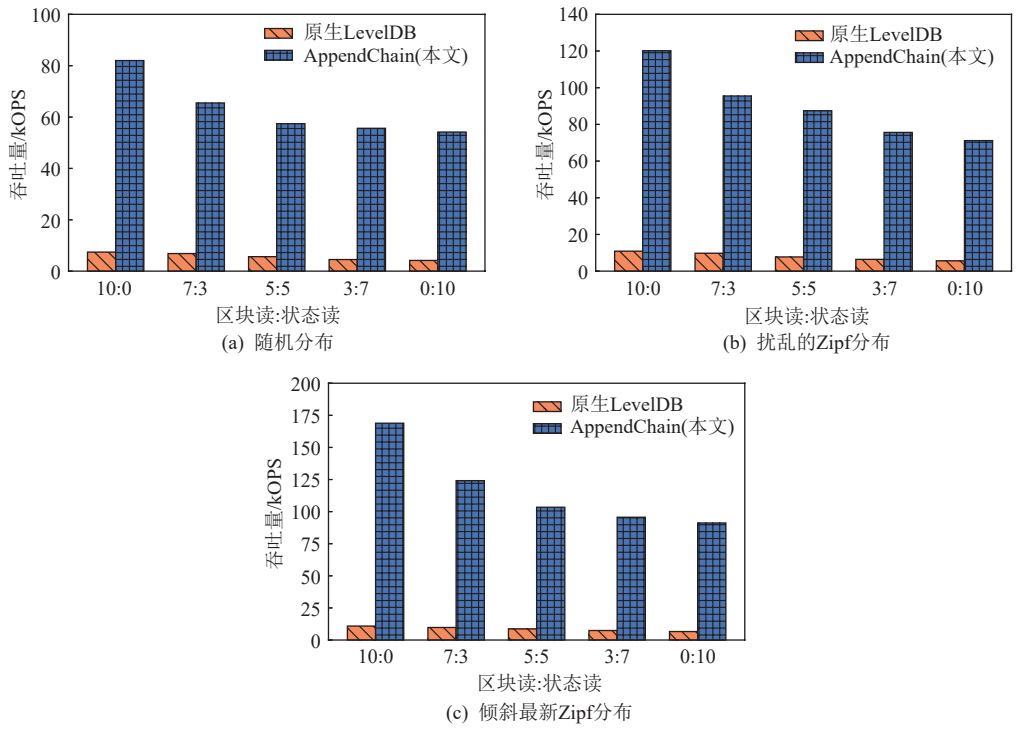


Fig. 13 Read performance of Native LevelDB and AppendChain under synthetic load

图 13 原生 LevelDB 与 AppendChain 在合成负载下的读性能

6 相关工作

本节回顾与我们研究密切相关的区块链存储流程优化工作, 主要从 3 个方面进行总结: 1)区块链存储开销优化; 2)状态结构与索引设计改进; 3)面向以太坊负载特性的存储感知优化。同时与主流实现方案进行对比, 具体如表 2 所示。

6.1 区块链存储优化

随着链上数据量的爆炸式增长, 大量研究致力于降低全节点的存储负担。现有方法主要可归为 2 类:

1)节点内的部分存储。此类方法通过在单节点内存储部分数据, 减少单个节点需要保存的数据量。例如, SPV(simplified payment verification)[24] 模型允许节点仅存储区块头信息, 从而验证交易是否存在于区块链中, 无需存储完整区块数据[25-26]。一些学者引入了区块体的部分存储[27,51], 例如 SlimChain[27] 将完整交易数据替换为摘要, 使用压缩证明确保数据存储的可验证性。然而这类方法虽有效缓解存储压力, 但通常会牺牲系统的可验证性与节点数据的完整性, 不适用于全节点或归档节点。

Table 2 Comparison Between AppendChain and Existing Mainstream Solutions

表 2 AppendChain 与现有主流方案的对比

方案	写吞吐量提升倍数	写放大率降低	存储开销变化	保留哈希溯源功能
COLE[37]	5.4		减少 93%	否
SlimArchive[9]	22.6		减少 98%	否
ChainKV[16]	2	64%	MPT 每 Key 增加 8 B	是
AppendChain(本文)	28	94%	MPT 每 Key 增加 10 B	是

2)多节点分布存储。另一类方法通过在节点间引入协作存储机制,实现链上数据的分布存储。例如,部分学者引入纠删码机制[52],设计轻节点来存储不同的纠删码编码块[28-30];部分学者提出分片机制,将整个区块链网络划分为不同的子分片,每个分片处理和存储局部交易数据。例如 Elastico[31], OmniLedger[32], ShardSchedule[33], BrokerChain[34], LB-Chain[35] 等协议。尽管该类方法提升了区块链网络整体的性能,但削减了系统的信任保证能力与去中心化程度,并且需要处理分片划分、跨分片交易处理、动态分片调整等问题,引入了额外的复杂度。

上述方法多从链上数据存储角度出发,忽视了底层存储引擎特性与区块链数据语义之间的不一致问题。而我们所提出的 AppendChain 则从键值存储引擎语义感知的角度进行优化,在保持数据完整性与可验证性的前提下显著减少写放大率与合并开销。

6.2 认证结构和索引设计

为了优化状态数据的存储,学术界还提出了多种替代 MPT 的状态认证结构或索引优化机制。

Erigon[9] 根据当前状态与状态读取集构建倒排索引来加速历史状态访问过程。但其访问路径需多次表连接和索引跳转,代价较高,且伴随着大量冗余中间数据的存储。LVMT(layered vector commitment-based Merkle tree)[36] 结合了向量承诺协议与多层认证结构,可在不依赖全路径哈希的前提下实现状态根更新,但其从根本上替换了 MPT 的哈希语义,与当前以太坊状态验证机制不兼容。LETUS[53] 提出了将 ADS 深度集成至日志结构存储系统中,也与当前以太坊的分层系统架构不兼容。COLE[37] 采用列存加学习索引替代 MPT,提升了历史状态的查询效率与压缩性能;SlimArchive[9] 剪除了 MPT 树中的所有中间认证哈希,仅存储了叶子节点,设计了基于版本号的键。但 COLE 和 SlimArchive 这 2 种方法都破坏了原有 MPT 的认证路径,牺牲了 MPT 的哈希溯源功能。

相反,AppendChain 并未引入新的状态结构,而是通过结构感知的键编码机制,在不破坏 MPT 树哈希认证路径的前提下实现数据局部优化,保持现有验证逻辑不变。

6.3 结合以太坊语义的存储优化

一些研究尝试利用以太坊数据的语义特性进行系统优化。RainBlock[14] 提出分层分片状态树 DSM-Tree(distributed, sharded Merkle tree),将 MPT 划分为 16 棵子树,并卸载到 16 个不同的存储节点,在上层设立 I/O Helper,执行状态预读与缓存,矿工只负责执行交易并提交状态修改到 I/O Helper。Block-LSM[10] 针对以太坊 MPT 节点键哈希的无序问题,提出了对状态数据加共享前缀(区块号)来缓解写放大效应。而后在 ChainKV[16] 又进一步提出了状态分离与空间竞争缓存来优化状态数据和其他数据的缓存空间,随后又对读写路径的 I/O 冗余提出了进一步优化[21]。与上述工作对比,我们将所有类型都分离到不同列族,做了更加精细化的管理;对于状态数据的前缀,我们的排序键相对于区块号有更细的粒度,能够对 MPT 进行全排列从而完全消除存储引擎的排序过程。

总体而言,已有研究未能充分考虑链上键值语义与 LSM-Tree 维持键全局有序特性间的本质冲突。本工作通过引入类型感知与顺序保持机制,在不破坏以太坊原有一致性语义的基础上,实现了数据在底层存储引擎中写路径的细粒度优化。

7 总　　结

本文提出了 AppendChain,一种基于类型感知与顺序保持的区块链数据存储优化框架。AppendChain 不改变以太坊的共识与状态语义,而是在存储层实现了写入路径与键空间布局的重构。我们首先将 RocksD 集成到 Geth 中,然后通过 3 项设计:类型分离、大值分离和排序 MPT,显著缓解了区块链工作负载的无序性对存储引擎的冲击,充分释放了底层 RocksDB 的性能潜力。实验评估结果表明了 AppendChain 在数据组织与减少冗余写入方面的实际效果。本工作证明了区块链存储瓶颈并非仅源于写入压力,而在于链上数据语义与存储引擎排序特性之间的不匹配。AppendChain 提供了一个通用范式:通过语义感知的布局调整与写入策略设计,可在不修改上层逻辑的前提下,提升存储引擎在区块链负载下的效率。

未来,我们考虑在以下方向拓展本研究:1)将 AppendChain 应用于更轻量、异构结构的以太坊客户端,如 Erigon 等。2)探索更普适的顺序编码机制,支持不同类型状态树(如 Verkle tree)的有序存储优化。3)引入基于事务流的并发调度模型,进一步提升吞吐能力。通过这些后续工作,我们希望进一步推动区块链存储在高性能、高效率方面的发展。

作者贡献声明:王锦江、于锦阳负责方案整体设计并撰写与修改论文;张晓、黄志杰提出指导意

见并修改论文；苏琳萱、甘一伟、何赫烺、Michael Ngulube 负责部分算法思路和实验方法。张晓和黄志杰为共同通信作者。

参 考 文 献

[1] Werner S, Perez D, Gudgeon L, et al. Sok: Decentralized finance (defi) [C]//Proc of the 4th ACM Conf on Advances in Financial Technologies. New York: ACM, 2022: 30−46

[2] Dutta P, Choi T M, Somani S, et al. Blockchain technology in supply chain operations: Applications, challenges and research opportunities[J]. Transportation Research Part E: Logistics and Transportation Review, 2020, 142: 102067

[3] Queiroz M M, Telles R, Bonilla S H. Blockchain and supply chain management integration: A systematic review of the literature[J]. Supply Chain Management: An International Journal, 2020, 25(2): 241−254

[4] Dai Hongning, Zheng Zibin, Zhang Yan. Blockchain for Internet of Things: A survey[J]. IEEE Internet of Things Journal, 2019, 6(5): 8076−8094

[5] Tang Fei, Xu Tingxian, Peng Jinlan, et al. TP-PBFT: A scalable PBFT based on threshold proxy signature for IoT-blockchain applications[J]. IEEE Internet of Things Journal, 2023, 11(9): 15434−15449

[6] Wood G. Ethereum: A secure decentralised generalised transaction ledger[J]. Ethereum Project Yellow Paper, 2014, 151: 1−32

[7] Zhang Baochen, Huang Yue, Kong Lanju, et al. A trustworthy and fair blockchain framework supporting adaptive federated learning task[J]. Journal of Computer Research and Development, 2023, 60(11): 2504−2519 (in Chinese)
(张宝晨, 黄月, 孔兰菊, 等. 一种支持自适应联邦学习任务的可信公平区块链框架[J]. 计算机研究与发展, 2023, 60(11): 2504−2519)

[8] Zheng Peilin, Zheng Zibin, Wu Jiajing, et al. Xblock-eth: Extracting and exploring blockchain data from Ethereum[J]. IEEE Open Journal of the Computer Society, 2020, 1: 95−106

[9] GitHub, Inc. Erigon client [EB/OL]. 2023 [2025-06-05]. https://github.com/ledgerwatch/erigon

[10] Chen Zehao, Li Bingzhe, Cai Xiaojun, et al. Block-LSM: An ether-aware block-ordered LSM-tree based key-value storage engine[C]//Proc of the 39th IEEE Int Conf on Computer Design (ICCD). Piscataway, NJ: IEEE, 2021: 25−32

[11] Etherscan. Ethereum full node sync (Archive) chart | Etherscan [EB/OL]. 2025 [2025-04-25]. https://etherscan.io/chartsync/chainarchive

[12] Bez M, Fornari G, Vardanega T. The scalability challenge of ethereum: An initial quantitative analysis [C]//Proc of IEEE Int Conf on Service-Oriented System Engineering (SOSE). Piscataway, NJ: IEEE, 2019: 167−176

[13] Buterin V. A next-generation smart contract and decentralized application platform[EB/OL]. 2014[2025-5-28]. https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf

[14] Ponnapalli S, Shah A, Banerjee S, et al. RainBlock: Faster transaction processing in public Blockchains [C]//Proc of USENIX Annual Technical Conf (USENIX ATC 21). Berkeley, CA: USENIX Association, 2021: 333−347

[15] Etherscan. Ethereum unique address growth chart[EB/OL]. 2024 [2025-04-25]. https://etherscan.io/chart/address

[16] O'Neil P, Cheng E, Gawlick D, et al. The log-structured merge-tree (LSM-tree)[J]. Acta Informatica, 1996, 33: 351−385

[17] Chen Zehao, Li Bingzhe, Cai Xiaojun, et al. ChainKV: A semantics-aware key-value store for Ethereum system[C]// Proc of ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2023: 1−23

[18] Dong S, Kryczka A, Jin Yanqin, et al. RocksDB: Evolution of development priorities in a key-value store serving large-scale applications[J]. ACM Transactions on Storage, 2021, 17(4): 1−32

[19] Facebook. RocksDB: A persistent key-value store for fast storage [EB/OL]. 2013 [2025-04-20]. https://github.com/facebook/rocksdb

[20] Google. LevelDB: A fast key-value storage library[EB/OL]. 2011 [2025-04-20]. https://github.com/google/leveldb

[21] Wei Qian, Chen Zehao, Chen Xiaowei, et al. A semantic-integrated LSM-Tree-based key-value storage engine for blockchain systems[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2023, 43(6): 1794−1807

[22] Lu Lanyue, Pillai T S, Gopalakrishnan H, et al. Wisckey: Separating keys from values in ssd-conscious storage[J]. ACM Transactions on Storage, 2017, 13(1): 1−28

[23] Facebook, Inc. BlobDB[EB/OL]. 2025 [2025-03-20]. https://github.com/ facebook/rocksdb/wiki/BlobDB

[24] Nakamoto S. Bitcoin: A peer-to-peer electronic cash system [EB/OL]. 2008 [2025-05-10]. https://bitcoin.org/bitcoin.pdf

[25] Antonopoulos A M. Mastering Bitcoin: Unlocking Digital Cryptocurrencies[M/OL]. O'Reilly Media, Inc, 2014[2025-05-20]. https://dl.acm.org/doi/10.5555/2695500

[26] Ali S T, McCorry P, Lee P H J, et al. ZombieCoin 2.0: Managing next-generation botnets using bitcoin[J]. International Journal of Information Security, 2018, 17(4): 411−422

[27] Xu Chen, Zhang Ce, Xu Jianliang, et al. SlimChain: Scaling blockchain transactions through off-chain storage and parallel processing[J]. Proceedings of the VLDB Endowment, 2021, 14(11): 2314−2326

[28] Zhang Yushu, He Xiaotong, Xiao Xiangli, et al. An optimisation method for blockchain ledger group storage based on erasure coding[J]. Computer Science, 2023, 50(10): 350−361 (in Chinese)
(张玉书, 何晓彤, 肖祥立, 等. 一种基于纠删码的区块链账本分组存储优化方法[J]. 计算机科学, 2023, 50(10): 350−361)

[29] Qi Xiaodong, Zhang Zhao, Jin Cheqing, et al. A reliable storage partition for permissioned blockchain[J]. IEEE Transactions on Knowledge and Data Engineering, 2020, 33(1): 14−27

[30] Fan Yuqi, Sheng Dong, Wang Lunfei. Blockchain storage optimization based on erasure coding[J]. Chinese Journal of Computers, 2022, 45(4): 858−876 (in Chinese)
(樊玉琦, 盛东, 王伦飞. 基于纠删码的区块链存储优化[J]. 计算机学报, 2022, 45(4): 858−876)

[31] Luu L, Narayanan V, Zheng Chaodong, et al. A secure sharding protocol for open blockchains [C]//Proc of the 2016 ACM SIGSAC Conf on Computer and Communications security. New York: ACM, 2016: 17−30

[32] Kokoris-Kogias E, Jovanovic P, Gasser L, et al. OmniLedger: A secure, scale-out, decentralized ledger via sharding[C]//Proc of the 2018 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2018: 583−598

[33] Król M, Ascigil O, Rene S, et al. Shard scheduler: Object placement and migration in sharded account-based blockchains[C]//Proc of the 3rd ACM Conf on Advances in Financial Technologies. New York: ACM, 2021: 43−56

[34] Huang Huawei, Peng Xiaowen, Zhan Jianzhou, et al. BrokerChain: A cross-shard blockchain protocol for account/balance-based state sharding [C] //Proc of the 2022 IEEE Conf on Computer Communications. Piscataway, NJ: IEEE, 2022: 1968−1977

[35] Que Qifeng, Chen Zhihao, Zhang Zhao, et al. A coordinator-free cross-shard transaction execution for sharded permissioned blockchains[J]. Journal of Computer Research and Development, 2023, 60(11): 2469−2488 (in Chinese)
(阙琦峰, 陈之豪, 张召, 等. 面向分片许可链的无协调者跨片交易处理[J]. 计算机研究与发展, 2023, 60(11): 2469−2488)

[36] Li Chenxing, Beillahi S M, Yang Guang, et al. LVMT: An efficient authenticated storage for blockchain[J]. ACM Transactions on Storage, 2024, 20(3): 1−34

[37] Zhang Ce, Xu Cheng, Hu Haibo, et al. COLE: A column-based learned storage for blockchain systems [C]//Proc of USENIX Conf on File and Storage Technologies (FAST 24). Berkeley, CA: USENIX Association, 2024: 329−345

[38] Ethereum Foundation. Go-ethereum (Geth) [EB/OL]. 2025 [2025-04-15]. https://github.com/ethereum/go-ethereum

[39] Facebook, Inc. Column families [EB/OL]. 2025 [2025-04-15]. https://github.com/facebook/rocksdb/wiki/Column-Families

[40] Ethereum Foundation. Geth sync modes [EB/OL]. 2025 [2025-04-15]. https://geth.ethereum.org/docs/fundamentals/sync-modes

[41] Swan M. Blockchain: Blueprint for A New Economy[M/OL]. O'Reilly Media, Inc, 2015 [2025-5-20]. https://dl.acm.org/doi/10.5555/3006358

[42] Yli-Huumo J, Ko D, Choi S, et al. Where is current research on blockchain technology? A systematic review[J]. Plos One, 2016, 11(10): 163477

[43] Zheng Zibin, Xie Shaoan, Dai Hongning, et al. An overview of blockchain technology: Architecture, consensus, and future trends [C]//Proc of IEEE Int Congress on Big Data (BigData Congress). Psicataway, NJ: IEEE, 2017: 557−564

[44] Casino F, Dasaklis T K, Patsakis C. A systematic literature review of blockchain-based applications: Current status, classification and open issues[J]. Telematics and Informatics, 2019, 36: 55−81

[45] Xu Xiwei, Weber I, Staples M, et al. A taxonomy of blockchain-based systems for architecture design [C]//Proc of IEEE Int Conf on Software Architecture (ICSA). Psicataway, NJ: IEEE, 2017: 243−252

[46] Wang Qin, Li Rujia, Wang Qi, et al. Non-fungible token (NFT): Overview, evaluation, opportunities and challenges [J]. arXiv preprint, arXiv: 2105.07447, 2021

[47] Wang Shuai, Ding Wenwen, Li Juanjuan, et al. Decentralized autonomous organizations: Concept, model, and applications[J]. IEEE Transactions on Computational Social Systems, 2019, 6(5): 870−878

[48] Ethereum Foundation. Nodes and clients | Ethereum documentation [EB/OL]. 2025 [2025-03-20]. https://ethereum.org/en/developers/docs/ nodes-andclients/

[49] Li Ming, Cheng Wendi, Wei Jiahe, et al. BVLSM: Write-efficient LSM-Tree storage via WAL-time key-value separation[J]. arXiv preprint, arXiv: 2506.04678, 2025

[50] Cockroach Labs. Pebble: A RocksDB-inspired key-value store in Go [EB/OL]. 2020 [2025-06-01]. https://github.com/cockroachdb/pebble

[51] Zhang Qizhi, He Yale, Lai Ruilin, et al. A survey on the efficiency, reliability, and security of data query in blockchain systems[J]. Future Generation Computer Systems, 2023, 145: 303−320

[52] Weatherspoon H, Kubiatowicz J D. Erasure coding vs replication: A quantitative comparison [C]//Proc of Int Workshop on Peer-to-Peer Systems. Berlin: Springer, 2002: 328−337

[53] Tian Shikun, Lu Zhonghao, Zhuo Haizhen, et al. LETUS: A log-structured efficient trusted universal blockcchain storage [C]//Proc of ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2024: 161−174



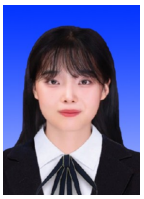
Wang Jinjiang, born in 1999. PhD candidate. His main research interests include blockchain storage system, cloud computing, and tiered memory management.

王锦江, 1999 年生。博士研究生。主要研究方向为区块链存储系统、云计算和分层内存管理。



Zhang Xiao, born in 1978. PhD, professor, PhD supervisor. His main research interests include distributed storage systems, cloud computing and cloud storage systems, system evaluation and simulation, and blockchain. (zhangxiao@nwpu.edu.cn)

张　晓, 1978 年生。博士, 教授, 博士生导师。主要研究方向为分布式存储系统、云计算和云存储系统、系统测评与仿真、区块链。



Su Linxuan, born in 2001. Master candidate. Her main research interests include blockchain and distributed system.

苏琳萱, 2001 年生。硕士研究生。主要研究方向为区块链、分布式系统。



Yu Jinyang, born in 2001. Master candidate. His main research interests include blockchain and distributed storage system.

于锦阳, 2001 年生。硕士研究生。主要研究方向为区块链、分布式存储系统。



Huang Zhijie, born in 1983. PhD, associate professor. His main research interests include coding theory, storage systems, and blockchain. (jayzy.huang@nwpu.edu.cn)

黄志杰, 1983 年生。博士，副教授。主要研究方向为编码理论、存储系统、区块链。



Gan Yiwei, born in 2002. Master candidate. His main research interests include distributed storage system, erasure coding, and data placement.

甘一伟, 2002 年生。硕士研究生。主要研究方向为分布式存储系统、纠删码、数据放置。



He Helang, born in 2001. Master candidate. His main research interests include blockchain and distributed system.

何赫烺, 2001 年生。硕士研究生。主要研究方向为区块链、分布式系统。



Michael Ngulube, born in 1993. PhD candidate. His main research interests include cloud computing, parallel programming, blockchain, and big data.

Michael Ngulube, 1993 年生。博士研究生。主要研究方向为云计算、并行编程、区块链、大数据。