软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007677] [CSTR: 32375.14.jos.007677]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

TDSM: 变更数据捕获驱动的数据库-区块链同步系统*

卿昱潇[1], 高建彬[1], 吴 敏[2], 夏 琦[1], 肖杨磊[3], 夏 虎[1], 张劲松[3], 罗锦涛[1]

[1](电子科技大学 计算机科学与工程学院 (网络空间安全学院), 四川 成都 611731)

[2](福建省数字政务建设运营控股有限公司, 福建 福州 350207)

[3](福建大数据信息安全建设运营有限公司, 福建 福州 350101)

通信作者: 高建彬, E-mail: gaojb@uestc.edu.cn

摘 要: 随着区块链技术在金融科技、供应链管理等关键领域的应用不断深化, 兼具传统数据库高性能与区块链高可信性的混合式存储架构已成为研究热点. 然而, 现有方案普遍未能系统性地解决数据库与区块链之间的架构不兼容问题, 具体表现为事务模型差异、性能与成本矛盾以及数据生命周期管理困难这 3 大核心技术难点. 针对上述挑战, 提出并实现一种基于变更数据捕获 (change data capture, CDC) 的数据库-区块链同步系统 TDSM, 用于在异构环境下实现高效、可信的数据同步. 首先, 设计集成可验证撤销证明的轻量级原子提交协议, 利用稀疏默克尔树为删除操作生成密码学证明, 在不引入重量级两阶段提交的前提下保障跨系统事务的原子性映射, 实现数据全生命周期的链上可信审计. 其次, 提出资源感知的自适应批处理算法, 通过实时监控链上交易成本并动态调整批处理策略, 有效解决数据库高通量与区块链高成本之间的矛盾. 实验结果表明, TDSM 在保持较高吞吐量的同时显著降低了同步成本, 自适应批处理机制相比固定策略节省了显著的资源开销, 撤销证明机制在处理大量删除操作时仍保持高效, 原子性保障机制在各种故障场景下均能保证数据一致性.

关键词: 区块链数据库; 混合存储; 数据同步; 事务一致性; 默克尔树; 变更数据捕获

中图法分类号: TP311

中文引用格式: 卿昱潇, 高建彬, 吴敏, 夏琦, 肖杨磊, 夏虎, 张劲松, 罗锦涛. TDSM: 变更数据捕获驱动的数据库-区块链同步系统. 软件学报. http://www.jos.org.cn/1000-9825/7677.htm

英文引用格式: Qing YX, Gao JB, Wu M, Xia Q, Xiao YL, Xia H, Zhang JS, Luo JT. TDSM: Database-blockchain Synchronization System Driven by Change Data Capture. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7677.htm

TDSM: Database-blockchain Synchronization System Driven by Change Data Capture

QING Yu-Xiao[1], GAO Jian-Bin[1], WU Min[2], XIA Qi[1], XIAO Yang-Lei[3], XIA Hu[1], ZHANG Jin-Song[3], LUO Jin-Tao[1]

[1](School of Computer Science and Engineering (School of Cyber Security), University of Electronic Science and Technology of China, Chengdu 611731, China)

[2](Fujian Digital Government Construction and Operation Holdings Co. Ltd., Fuzhou 350207, China)

[3](Fujian Big Data Information Security Construction and Operation Co. Ltd., Fuzhou 350101, China)

Abstract: With the increasing application of blockchain technology in critical domains such as financial technology and supply chain management, hybrid storage architectures that combine the high performance of traditional databases with the high trustworthiness of blockchains have become a research hotspot. However, existing solutions generally fail to systematically address the architectural incompatibilities between databases and blockchains, which manifest in three core technical challenges: transaction model differences, performance-cost contradictions, and data lifecycle management difficulties. To address these challenges, this study proposes and implements TDSM, a CDC-based database-blockchain synchronization system for achieving efficient and trustworthy data synchronization

* 基金项目: 国家自然科学基金 (U22B2029)
收稿时间: 2025-11-18; 修改时间: 2026-01-26; 采用时间: 2026-03-03; jos 在线出版时间: 2026-07-08

in heterogeneous environments. First, the study designs a lightweight atomic commit protocol integrated with verifiable revocation proofs and utilizes sparse Merkle trees to generate cryptographic proofs for deletion operations, ensuring the atomic mapping of cross-system transactions without introducing heavyweight two-phase commit, and thus achieving trusted on-chain auditing of the complete data lifecycle. Second, it proposes a resource-aware adaptive batching algorithm, which dynamically adjusts batching strategies by monitoring on-chain transaction costs in real time, effectively resolving the contradiction between the high throughput of databases and the high cost of blockchains. Experimental results demonstrate that TDSM maintains high throughput while significantly reducing synchronization costs. The adaptive batching mechanism achieves substantial resource savings compared to fixed strategies, the revocation proof mechanism remains efficient when processing large numbers of deletion operations, and the atomicity guarantee mechanism ensures data consistency across various failure scenarios.

Key words: blockchain database; hybrid storage; data synchronization; transaction consistency; Merkle tree; change data capture (CDC)

随着区块链技术在金融科技、供应链管理、数字身份等关键领域的应用不断深化[1], 兼具传统数据库高性能与区块链高可信性的混合式存储架构已成为研究热点与行业趋势[2]. 据研究预测, 到 2030 年, 全球物联网连接设备数量将超过 300 亿台, 由此产生的海量、高频数据不仅对存储系统的实时处理能力提出了极高要求, 更在多方协作场景下催生了对数据透明、可追溯、防篡改的迫切需求. 区块链技术以其去中心化和不可篡改的特性, 为解决多方信任问题提供了理想解决方案[3,4]. 但主流区块链系统的处理能力与传统高性能数据库系统之间存在数个数量级的性能鸿沟. 因此, 设计一种能够将链下数据库的高通量变更实时、可靠地同步至链上信任锚的解决方案, 对于推动区块链技术规模化应用具有至关重要的理论与实践意义[5].

当前, 学术界和工业界对数据库与区块链的融合已进行了广泛探索, 主要形成了 3 类架构范式: 一是为传统数据库添加区块链特性的区块链增强型数据库, 如 LedgerDB[6]; 二是为区块链引入高性能数据库作为存储引擎的数据库支持型区块链, 如 BigchainDB[7]; 三是构建独立系统, 作为桥接中间件. 其中, 中间件模式因非侵入性的优势, 无需修改现有业务系统, 在将现有系统与区块链结合方面展现出巨大的应用潜力.

现有的中间件方案普遍未能系统性地解决数据库与区块链之间固有的架构不兼容问题[8], 具体表现为 3 大核心技术难点: 一是事务模型不匹配, 数据库的 ACID 原子事务与区块链的概率性最终确定性模型难以直接映射, 存在数据一致性风险; 二是性能与成本不匹配, 高通量的数据库变更流与区块链的低通量、高成本特性形成尖锐矛盾, 静态的批处理策略难以实现资源成本最优; 三是数据生命周期不匹配, 虽然区块链可通过墓碑标记等机制记录删除意图, 但逐条记录的方式缺乏对删除集合的整体性密码学承诺, 难以高效支撑合规审计中对删除完整性和非成员关系的验证需求, 导致数据全生命周期的可信审计链路存在缺口[9].

为系统性地解决上述挑战, 本文提出并实现了一个通用的、非侵入式的数据库-区块链同步中间件——可信数据同步中间件 (trusted data synchronization middleware, TDSM). 我们的核心技术路线基于业界成熟的、基于日志的变更数据捕获 (change data capture, CDC) 技术和分布式事件流平台. 该基础架构确保了对源数据库的最低性能侵入和数据变更的实时捕集, 为中间件提供了一个可靠、有序的事件源. 在此基础架构之上, TDSM 的核心思想是对两端系统主动感知并适应协调. 它通过消费由 CDC 产生的事件流, 分别从可靠性与完整性、经济性两个维度, 系统性地解决了前述的 3 大技术难点, 从而构建了一个健壮的同步解决方案.

直接的同步方案为数据库的每个变更都生成一笔链上交易, 会破坏源端事务的原子性. 简单的聚合方案虽可通过状态标记记录 DELETE 意图, 但分散的标记缺乏对删除集合的密码学承诺, 无法高效支撑第三方对删除完整性的独立验证. 为此, 本协议引入了一种基于稀疏默克尔树的可验证撤销证明机制, 该机制将一批源事务内的所有 DELETE 操作的标识聚合成一个紧凑的默克尔根并编码进区块链交易中. 这种设计以极低的成本为删除这一链下行为提供了可公开验证的、不可篡改的密码学证明, 补全了数据在混合系统中的全生命周期可信审计环路.

同时, 为解决性能与成本不匹配的问题, 本文设计并实现了一种成本感知的自适应批处理算法. 静态的批处理策略, 如固定批次大小或固定时间窗口, 无法适应区块链资源成本的实时波动, 可能导致在费用高峰期产生不必要的开销, 或在费用低谷期未能充分利用网络容量. 为此, 本文提出的算法通过实时监控链上资源成本, 并结合可配置的成本效益函数, 动态调整数据批次的大小和提交时机. 该算法旨在同步延迟与资源之间寻求一个动态的最优平衡, 从而在满足业务时效性要求的前提下, 最大限度地降低链上同步的总成本.

总体而言, 本文的主要贡献如下.

• 提出一种名为 TDSM 的自适应同步中间件, 并实现一个完整的、非侵入式的数据库-区块链同步方案, 将链下高性能数据库的变更可靠、经济、完整地锚定至链上.

• 提出一种集成了可验证撤销证明的轻量级原子提交协议. 该协议通过在中间件内聚合源事务的变更事件, 并利用稀疏默克尔树为删除操作生成密码学证明, 在不引入重量级两阶段提交的前提下, 保障了跨系统事务的原子性映射, 实现了数据全生命周期的链上可信审计.

• 设计并实现一种成本感知的自适应批处理算法. 该算法通过实时感知链上交易成本并动态调整批处理策略, 在同步延迟与资源成本之间取得了较优的平衡.

• 对 TDSM 的原型系统进行全面的实验评估. 实验结果从成本优化效果、原子性保障、自适应效率等多个维度, 验证了本文所提架构与核心机制的有效性, 并通过与同类系统的数据对比分析了系统的性能特征.

本文第 1 节介绍区块链与数据库融合架构的研究现状. 第 2 节介绍本文所需的基础知识. 第 3 节介绍 TDSM 架构. 第 4 节和第 5 节将分别深入介绍轻量级原子提交协议与资源感知的自适应批处理机制. 第 6 节介绍本文的原型系统实现. 第 7 节通过对比实验验证 TDSM 系统的有效性和高性能. 最后对全文进行总结.

1 相关工作

当前, 将区块链与数据库进行融合的探索主要形成了 3 类主流架构范式: 区块链增强型数据库、数据库支持型区块链以及中间件模式. 这些架构在信任模型、系统侵入性及对遗留系统的兼容性上各有取舍.

1.1 区块链增强型数据库

区块链增强型数据库通过在传统数据库之上增加区块链特性来构建可信数据管理系统. 代表性工作包括阿里巴巴的 LedgerDB[6]和 IBM 的 blockchain relational database[10]. LedgerDB 采用中心化账本技术, 通过批量累积 Merkle 树实现 O(m+d) 复杂度的高效验证, 其吞吐量达 30 万 TPS, 是 Hyperledger Fabric[11]的 80 倍. 系统引入了 TSA 两路锚定协议以实现外部可审计性, 并通过 Purge 和 Occult 操作支持可验证的数据删除. IBM 的 blockchain relational database 则修改了 PostgreSQL 内核, 基于区块高度实现了新型可串行化快照隔离, 支持丰富的 SQL 查询和溯源功能, 其 execute-order-in-parallel 模式可达 2 700 TPS.

这类方案的核心优势在于能够提供完整的 ACID 保证和丰富的查询能力, 同时具备区块链级别的审计能力. 但存在一些局限性. 首先, 依然存在事务模型不匹配问题: 区块链共识机制和数据库一致性之间的协调存在显著开销. Blockchain relational database 的论文指出, 事务冲突检测发生在提交阶段, 导致成功执行的事务仍可能被中止, 在高并发场景下中止率可达 40%–60%. 其次, 系统侵入性问题不容忽视: Blockchain relational database 需要修改 PostgreSQL 核心代码, 增加了维护成本和迁移风险, 而 LedgerDB 虽号称非侵入式, 但实质上需要应用层完全重构以适应其 execute-commit-index 工作流.

1.2 数据库支持型区块链

数据库支持型区块链将传统数据库作为区块链的存储后端, 代表性系统包括 BigchainDB[7]、ChainSQL[12]和 SEBDB[13]. BigchainDB 使用 MongoDB 存储交易数据, 通过 Tendermint 实现 BFT 共识, 理论吞吐量可达 640–1 200 TPS. 虽然提升了查询能力, 但这类系统的实际性能受共识协议的选择影响显著. 采用 CFT 共识 (如 Raft) 的 Hyperledger Fabric 在共识与存储分离的架构下仍可达到数千 TPS, 而 BigchainDB 等采用 BFT 共识的系统, 由于拜占庭容错所需的高通信复杂度, 实际测试中吞吐量通常低于 200 TPS, 远不能满足高通量业务场景的需求[7]. ChainSQL 集成 Ripple 区块链与关系型数据库, 在区块链达成共识后转发操作至数据库执行, 共识时间约 4 s, 实现了零恢复时间的多活数据库中间件. SEBDB 首次将关系语义引入区块链, 通过重新设计区块结构支持 SQL 查询, 采用 3 级索引加速链上连接和范围查询操作.

这类方案的主要优势是保留了数据库的查询功能, 但依然存在跨层事务一致性挑战. 其中, BigchainDB 等将数据库直接作为区块链存储后端的紧耦合系统, 虽然回避了独立的一致性协调问题, 但数据库操作受区块链语义

严格约束, 难以支持完整的关系操作. 而 ChainSQL 等在共识后将操作转发至独立数据库执行的松耦合系统, 则面临真实的跨层协调问题: 共识成功但数据库执行失败需要回滚, 而部分数据库更新需要 ACID 保证. OmniLedger[14] 等分片系统在跨分片交易中使用 2PC 协议, 每个阶段需运行完整 BFT, 导致双倍的开销和阻塞风险. 研究表明, BFT 共识的处理速度通常仅为 CFT 的 1/3–1/2, 加上存储层协调开销, 性能仅为传统数据库的 1/10–1/100. 更严峻的是 DELETE 操作的根本性矛盾: BigchainDB 和 BlockchainDB 通过元数据分离或软删除标记实现逻辑删除, 但原始数据永久保留在区块链中, 无法真正删除. FalconDB[15]甚至未明确讨论 DELETE 语义. 这使得这类系统在需要真实删除数据的场景中难以应用.

1.3 中间件模式

中间件模式通过在区块链和数据库之间提供协调层, 无需修改底层系统即可实现混合存储, 这种模式的核心优势在于非侵入性和灵活性. Wang 等人[16]提出的事务同步中间件通过流水线处理和批量打包, 将 Hyperledger Fabric 的 TPS 提升至 16000+, 相比直接提交方法提升 32 倍.

增量计算是中间件实现高效数据同步的理论基础. DBSP 框架[17]通过差分数据流 (differential dataflow) 实现了自动化的增量视图维护, 其时间复杂度为 O(Δ), 其中 Δ 为变更的大小. DBSP 将查询表示为流处理算子的组合, 通过增量算子 (如 Δjoin、Δagg) 自动推导增量更新逻辑, 无需开发者手动编写增量代码. 该框架在 TPC-H 基准测试中实现了 99.2% 的增量计算准确率和 103 倍的性能提升.

跨系统原子提交是中间件的核心难点. 传统的两阶段提交协议[18]通过协调者和参与者之间的 prepare-commit 消息交换保证分布式事务的原子性. 但其存在阻塞问题: 协调者故障会导致参与者长时间等待. 此外, 区块链的异步确认特性与两阶段提交协议的同步协调范式存在根本性矛盾. 针对区块链场景 AC3WN 协议[19]通过无许可见证网络协调多个区块链之间的原子提交. AC3WN 的关键创新在于将协调逻辑外包给去中心化的见证网络, 避免了单点故障. 见证者通过质押代币参与协调, 并在协调成功后获得奖励. AC3WN 在以太坊和 Polygon 之间的跨链转账实验中, 成功率达到 99.8%, 平均延迟为 18 s. 但该协议需要额外的网络基础设施和见证者激励机制, 增加了系统复杂度和运营成本.

为了应对区块链的低吞吐量限制, 中间件需要采用更高效的批处理策略. Hackwrench[20]通过细粒度重执行和依赖跟踪实现了高效的批量事务提交, 显著优于 FoundationDB[21]等系统. Drizzle[22]提出的组调度 (group scheduling) 技术通过预先调度多个微批次, 将端到端处理延迟降至 100 ms 以下, 故障恢复速度提升 4 倍. MBS 框架[23]展示了成本感知的自适应批处理, 通过贝叶斯优化和动态缓冲区管理, 实现了 8 倍的成本节约. 这些自适应调度机制为中间件提供了在吞吐量和延迟之间动态权衡的能力.

为了更系统地比较现有代表性工作与本文方案的技术差异, 表 1 从 5 个关键维度对上述工作中最具代表性的 3 个系统进行了对比分析. 其中, Wang 等人[16]的同步中间件是当前最接近的完整同步方案, AC3WN 代表了跨系统原子提交方向的研究, Hackwrench 则代表了基于批处理的性能优化方向.

表 1 TDSM 与代表性相关工作技术对比

技术维度	Wang等人[16]	AC3WN[19]	Hackwrench[20]	TDSM
源系统侵入性	√	×	×	√
跨系统一致性	×	√	√	√
生命周期管理	×	×	×	√
性能优化机制	√	×	×	√
资源成本优化	×	×	×	√

如表 1 所示, Wang 等人[16]的中间件同样采用非侵入式的数据拉取方式, 并通过流水线和批量打包实现了性能优化, 但未提供跨系统事务的原子性保证, 也未涉及数据删除等生命周期管理. AC3WN 通过见证网络实现了跨链原子提交, 但需要部署额外的网络基础设施, 对参与系统有侵入性要求. Hackwrench 通过细粒度重执行保证了批量事务的可串行化, 但其作为数据库内部优化机制, 需要应用使用专用的数据流 API 编写存储过程. 相比之下,

TDSM 通过基于 CDC 的非侵入式架构, 在同一系统中同时解决了事务一致性、数据生命周期管理和资源成本优化问题.

尽管现有工作在中间件架构、增量计算、原子提交和批处理优化等方面取得了重要进展, 但仍存在以下不足: (1) 现有同步方案主要针对传统数据库场景, 然而区块链引入了交易成本的显著波动和交易确认的长延迟, 难以直接应用; (2) 现有协议多假设参与者具有同构特性. 但数据库与区块链在性能、一致性模型和故障模式上存在根本差异, 需引入额外协调机制, 增加了系统复杂度; (3) 现有算法多聚焦单一目标 (延迟或吞吐量), 区块链场景下需考虑系统成本与性能的权衡, 现有静态策略难以应对这种动态变化.

2 基础知识

本节主要介绍本文所需的基础技术, 包括变更数据捕获技术以及稀疏默克尔树的数据验证机制. 这些技术为本文提出的 TDSM 中间件架构提供了理论基础和技术支撑.

2.1 变更数据捕获技术

变更数据捕获 (CDC) 是一种用于识别和捕获数据库中数据变更的技术, 被广泛应用于数据同步、数据仓库更新、实时分析等场景. CDC 能够以低侵入性的方式实时捕获数据库的增、删、改操作, 是实现非侵入式数据同步的关键技术基础.

根据数据捕获的来源和方法不同, CDC 主要有 3 种实现方式. 基于触发器的 CDC (trigger-based CDC) 通过在数据库表上创建触发器来捕获数据变更. 当表上发生插入、更新或删除操作时, 触发器被激活并将变更记录写入专门的变更日志表. 这种方式实现简单, 但会对数据库的写入性能产生明显影响, 且触发器的维护成本较高. 基于查询的 CDC 通过定期轮询表中的时间戳字段或版本号字段来识别变更的数据. 例如, 可以查询 update_time > last_sync_time 来获取自上次同步以来的所有变更. 这种方式对数据库性能影响较小, 但存在两个主要缺陷: 一是无法捕获删除操作, 因为被删除的记录已从表中移除; 二是存在轮询间隔导致的延迟. 基于日志的 CDC 通过读取和解析数据库的事务日志来捕获变更. 这种方式对数据库性能影响最小, 能够捕获所有类型的变更操作 (包括删除), 且具有较低的延迟, 已成为当前主流的 CDC 实现方案.

现代关系型数据库通常采用预写日志 (write-ahead logging, WAL) 机制[24]来保证事务的持久性和可恢复性. 在 WAL 机制下, 数据库在修改数据页之前, 会先将修改操作记录到事务日志中. 事务日志中记录了详细的变更信息, 包括操作类型、表名、主键、变更前后的列值以及事务的时间戳和事务 ID. 以 MySQL 的 binlog 为例, 其记录格式包括 statement 格式 (记录 SQL 语句)、row 格式 (记录每行的具体变更) 和 mixed 格式 (混合前两种格式). PostgreSQL 的 WAL 则记录了数据页的物理修改以及逻辑解码信息. CDC 组件通过持续读取和解析这些日志文件, 提取出其中的变更事件, 并将其转换为标准化的数据变更消息.

CDC 组件解析日志后, 通常会将变更事件发送到消息队列或事件流平台进行后续处理. 如图 1 所示, 事件流平台为 CDC 系统提供了可靠的消息传输和存储能力. 事件流具有 3 个关键特性: (1) 有序性, 同一分区内的事件按照产生顺序严格排列; (2) 持久性, 事件被写入磁盘并可配置多副本备份; (3) 可回放性, 消费者可以从任意历史位置开始消费事件. 下游的消费者通过订阅事件流来获取变更通知并执行相应的业务逻辑. 这种架构实现了数据生产者和消费者之间的解耦, 具有良好的可扩展性和容错能力, 能够支持多个下游系统并行消费同一份变更数据, 且在系统故障恢复时可以从特定位置重新消费事件以保证数据完整性.

基于日志的 CDC 技术以其低侵入性、高实时性和完整性, 已成为数据同步领域的主流方案. 该技术无需修改应用代码或数据库模式, 对源数据库的性能影响微乎其微, 且能够捕获所有类型的数据变更. 本文提出的 TDSM 中间件采用基于日志的 CDC 方案, 通过解析数据库事务日志捕获数据变更. 由于数据库在事务提交时才将完整的事务变更记录写入可供外部读取的事务日志, 基于日志的 CDC 以已提交事务为捕获粒度, 而非逐条记录地捕获单个操作——未提交或已回滚的事务不会产生变更事件. 捕获的变更事件随后被发布到事件流平台, 实现与区块链的可靠同步.

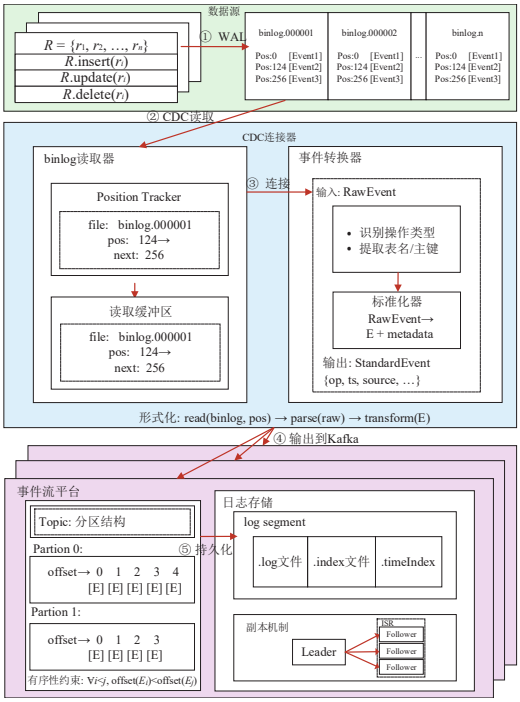


图 1 CDC 工作流程

在实际应用中, 基于日志的 CDC 常与发件箱模式 (outbox pattern) 配合使用, 以解决业务操作与事件发布的原子性问题. 在拒绝分布式事务的大规模系统中, 应将待发送的消息存储在实体的本地事务内, 通过异步传递实现跨系统协调, 从而将分布式原子性问题转化为本地事务问题[25]. 基于这一思想, 发件箱模式[26]的核心机制是: 在业务数据库中创建专用的事件表, 应用程序在执行业务操作的同一数据库事务中, 将待发布的事件记录写入该表. 由于业务操作与事件写入处于同一本地事务中, 数据库的 ACID 特性天然保证了二者的原子性——业务操作成功则事件必然被记录, 业务操作回滚则事件也不会被记录. 随后, CDC 组件监听 outbox 表的变更, 将新写入的事件记录捕获并转发至消息队列, 完成事件的可靠发布. 本文提出的 TDSM 中间件同样基于该模式保证数据库事务与变更事件捕获的原子性, 具体设计详见第 4.2 节.

2.2 稀疏默克尔树

在分布式系统和区块链中, 数据的完整性验证是一项基础性需求. 默克尔树 (Merkle tree)[27]是一种经典的哈希树数据结构, 能够高效地验证大规模数据集的完整性和某个数据项的存在性. 稀疏默克尔树 (sparse Merkle tree, SMT)[28]是默克尔树的一种变体, 特别适用于数据稀疏的场景.

默克尔树是一种二叉树结构, 其叶节点存储数据项的哈希值, 非叶节点存储其两个子节点哈希值的哈希. 通过自底向上的哈希计算, 最终得到根节点的哈希值, 称为默克尔根 (Merkle root). 默克尔根唯一地表示了整个数据集的状态, 任何数据的变化都会导致默克尔根发生变化. 默克尔树的关键优势在于其高效的验证机制: 要证明某个数据项存在于数据集中, 只需提供从该叶节点到根节点路径上的所有兄弟节点哈希 (称为默克尔证明或验证路径), 验证者通过这些哈希值重新计算根哈希, 并与已知的默克尔根比对即可验证. 这一过程的时间和空间复杂度均为 $O(\log n)$, 其中 n 为数据集大小. 默克尔树被广泛应用于区块链系统中, 例如比特币和以太坊使用默克尔树来组织和验证区块中的交易.

传统的默克尔树要求所有的叶节点位置都被填充, 这在数据稀疏的场景下会造成严重的空间浪费. 例如, 如果密钥空间为 256 位, 而实际只存储了数千条数据, 标准默克尔树仍需要为所有可能的位置分配节点. 稀疏默克尔树

针对这一问题进行了优化: 它只显式存储包含实际数据的节点, 空节点使用预定义的默认哈希值表示. 稀疏默克尔树通常被构建为一棵固定深度的二叉树, 深度通常为哈希函数输出的位数 (如 256 位). 数据项的键 (key) 通过哈希函数映射后, 其哈希值的每一位决定了该数据在树中的路径: 0 表示向左子树, 1 表示向右子树. 如图 2 所示, 在标准默克尔树中, 即使某些位置没有数据, 仍需要为所有节点分配存储空间; 而在稀疏默克尔树中, 空子树可以用一个预定义的空哈希值表示, 大幅节省了存储空间.

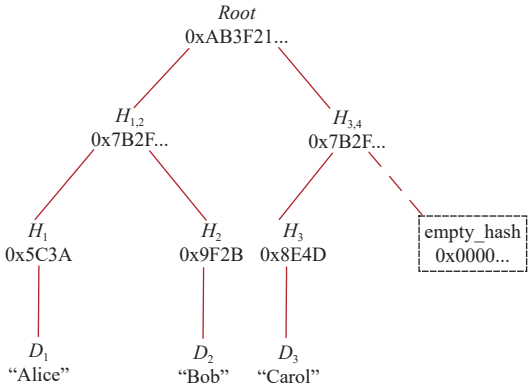


图 2　稀疏默克尔树结构示例

稀疏默克尔树的核心优势在于其空间效率. 在数据稀疏的场景下, 标准默克尔树的空间复杂度为 $O(2^n)$, 其中 n 为树的深度; 而稀疏默克尔树的空间复杂度降低为 $O(k \cdot \log n)$, 其中 k 为实际存储的数据量. 当 $k \ll 2^n$ 时, 稀疏默克尔树可以节省 90% 以上的存储空间. 同时, 稀疏默克尔树保持了与标准默克尔树相同的验证效率, 时间复杂度仍为 $O(\log n)$. 稀疏默克尔树还支持高效的动态更新: 插入新数据时, 只需沿着键的哈希值确定的路径向下, 更新或创建必要的节点; 删除数据时, 只需将对应的叶节点替换为空哈希值, 并向上更新祖先节点.

稀疏默克尔树的验证过程与标准默克尔树类似. 假设需要验证数据 D_3 存在于树中, 验证者需要: (1) 从证明提供者获取验证路径, 即从 D_3 到根节点路径上所有兄弟节点的哈希值; (2) 计算 $H_3 = Hash(D_3)$; (3) 使用验证路径中的 H_4, 计算 $H_{3,4} = Hash(H_3 \| H_4)$; (4) 使用验证路径中的 $H_{1,2}$, 计算 $Root' = Hash(H_{1,2} \| H_{3,4})$; (5) 比对 $Root'$与已知的 $Root$ 是否相等. 如果相等, 则证明 D_3 确实存在于树中且未被篡改. 由于验证路径的长度为 $O(\log n)$, 验证过程非常高效.

稀疏默克尔树以其高效的空间利用和快速的验证能力, 成为区块链和分布式系统中数据验证的理想选择. 本文利用稀疏默克尔树的这些特性, 为数据库的删除操作构建可验证撤销证明, 从而实现数据全生命周期的链上可信审计. 具体机制将在第 4 节中详细介绍.

3　TDSM: 可信数据同步中间件架构

设计支持数据库-区块链同步的中间件架构的目标是在保证非侵入性和高可用性的前提下, 实现将传统数据库的数据变更高效、经济、完整地同步到区块链. 本文的思路是利用 CDC 技术捕获数据变更, 通过事件流平台传输变更事件, 在中间件层进行自适应协调和批处理, 最终提交到区块链, 从而保证跨系统的事务原子性和数据完整性[29].

为了实现这一目标, 本文设计的 TDSM 架构如图 3 所示, 采用 4 层架构设计, 分为数据源层、变更捕获层、自适应同步层和区块链层. 在数据源层, 传统数据库无需任何修改即可接入系统; 在变更捕获层, CDC 组件以低延迟、非侵入的方式捕获数据库事务, 并依赖事件流平台提供可靠的消息传输和持久化[30]; 在自适应同步层, 本文设计了多个核心模块来协调事务处理、批量调度和故障恢复[31]; 在区块链层, 智能合约验证并执行批量提交的事务.

自适应同步层是 TDSM 的核心创新所在. 为了保证事务的原子性和数据的完整性, 本文设计了轻量级原子提

交协议, 该协议通过在中间件内聚合源事务的变更事件, 并利用稀疏 Merkle 树为 DELETE 操作生成密码学证明, 在不引入重量级两阶段提交的前提下, 保障了跨系统事务的原子性映射. 有关原子提交协议的细节本文将在第 4 节介绍. 此外, 为了应对区块链交易成本的动态变化, 本文设计了基于成本感知的自适应批处理算法, 该算法通过实时监控链上资源成本并动态调整批处理策略, 在同步延迟与资源成本之间取得了较优的平衡. 这一点本文将在第 5 节中介绍.

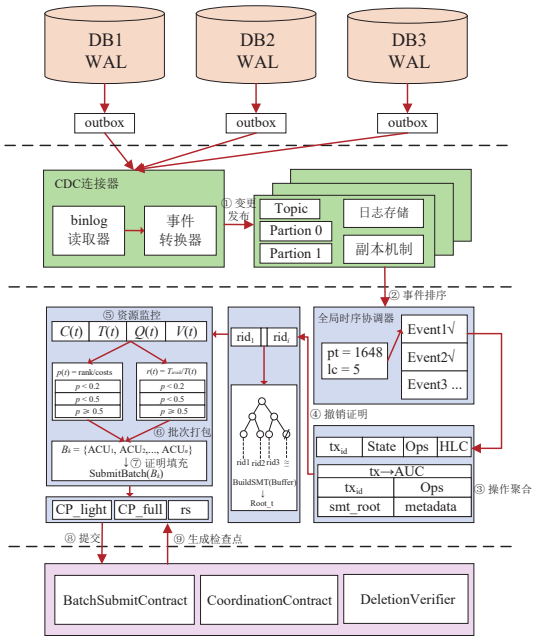


图 3 TDSM 可信数据同步中间件架构

如图 3 所示, 数据源层由一个或多个异构的传统数据库组成, 这些数据库通过事务发件箱模式[26]接入 TDSM. 变更捕获层部署 CDC 连接器, 通过读取数据库的事务日志以低侵入、高实时的方式捕获所有数据变更操作, 并为每个事务打上混合逻辑时钟 (hybrid logical clock, HLC)[32]时间戳以建立全局顺序. CDC 捕获的变更事件被发布到 Kafka 主题中, Kafka 提供了有序性、持久化和可回放性这 3 个关键特性, 确保变更事件不会丢失. 区块链层由去中心化的区块链网络组成, 智能合约部署在链上, 负责验证 TDSM 提交的批量事务和撤销证明的有效性.

自适应同步层中的每个处理节点配备有如下组件: 一个全局时序协调器、一个事务协调引擎、一个批处理调度器、一个撤销证明生成器、一个一致性管理器.

• 全局时序协调器负责为来自多个数据源的事务建立全局因果顺序. 它利用混合逻辑时钟 (HLC) 为每个事务分配全局时间戳, 并按照 HLC 定义的顺序将事件传递给事务协调引擎. HLC 结合了物理时间和逻辑计数器, 在无需 GPS 硬件的情况下提供全局一致的时间戳. 由于 HLC 的因果性保证, 若事务 T_2 依赖事务 T_1 的写入结果, 则 T_1 的事件必然先于 T_2 被协调器处理和传递.

• 事务协调引擎负责从有序事件流中识别和聚合完整的数据库事务. 它根据 CDC 提供的事务边界标记将属于同一个事务的所有操作聚合为一个原子提交单元 (atomic commit unit, ACU). 对于包含 DELETE 操作的事务, 协调引擎调用撤销证明生成器为其生成可验证撤销证明.

• 批处理调度器负责动态聚合多个 ACU 形成批次, 并基于实时的交易成本决定提交时机. 调度器持续监控链上资源成本情况以预测资源成本走势, 在延迟与成本间动态权衡, 选择最优的批次大小和提交时机.

• 撤销证明生成器基于稀疏 Merkle 树 (SMT) 为 DELETE 操作提供密码学证明. 它周期性地收集所有 DELETE

事件, 构建 SMT 并将树根哈希提交到链上, 同时在链下存储完整的树结构以支持证明查询.

• 一致性管理器负责维护系统状态的一致性视图, 并通过分层检查点机制追踪系统的处理进度. 它定期生成轻量级和完整检查点, 记录已处理的 Kafka offset、已提交的批次 ID 等关键状态, 为故障恢复提供一致性保障.

基于 TDSM 架构的数据同步处理机制. 以图 3 中为例, 数据在 TDSM 中的同步流程包括 9 步. 首先, 业务应用在数据库中执行事务 T_1, CDC 组件捕获该事务的变更并打上 HLC 时间戳 HLC_1, 将变更事件发布到 Kafka (第①步). 例如, 事务 T_1 包含一条 INSERT 操作 $I_1(x)$、一条 UPDATE 操作 $U_1(y)$ 和一条 DELETE 操作 $D_1(z)$. 接下来, 全局时序协调器从 Kafka 消费事件, 并按照 HLC 顺序将事件传递给事务协调引擎 (第②步). 事务协调引擎识别到 T_1 的 COMMIT 标记后, 聚合该事务的所有操作生成原子提交单元 ACU_1 (第③步). 由于 T_1 包含 DELETE 操作 $D_1(z)$, 协调引擎调用撤销证明生成器, 将 z 的记录 ID 加入当前周期的缓冲区, 并在 ACU_1 中预留撤销证明字段 (第④步). 接下来, 批处理调度器将 ACU_1 加入待提交队列. 假设此时队列中已累积了 n 个 ACU, 调度器监控到当前交易成本处于历史平均值的低位 (第⑤步). 调度器评估当前批次的成本效益, 并决定继续等待以累积更多 ACU 来降低平均成本. 当队列中的 ACU 数量达到阈值或等待时间超过上限时, 调度器将队列中的所有 ACU 按 HLC 时间戳排序后打包成批次 B_k (第⑥步). 在打包前, 撤销证明生成器完成当前周期 SMT 的构建, 将树根哈希 $root_k$ 填充到包含 DELETE 操作的 ACU 中 (第⑦步). 随后, 调度器按照批次生成顺序依次调用批量提交的智能合约接口, 将批次 B_k 和 SMT 根哈希 $root_k$ 提交到链上. 智能合约验证批次签名, 并按顺序执行 B_k 中的每个 ACU, 将数据存储在链上, 同时将 $root_k$ 存储以支持后续的 DELETE 操作审计 (第⑧步). 最后, 一致性管理器等待区块确认, 更新系统的已提交状态, 并生成检查点记录当前处理进度 (第⑨步). 此时, 批次 B_k 中的事务已完整地从数据库同步到区块链.

需要注意的是, 当单个 ACU 的大小超过区块链的成本限制时, 事务协调引擎会将该 ACU 拆分为多个子 ACU, 并通过链上的协调合约追踪所有子 ACU 的提交状态, 以保证原子性. 这一点将在第 4 节详细介绍. 此外, 当批次提交失败时 (如 Gas 耗尽、网络超时等), 一致性管理器会根据故障类型执行相应的补偿策略, 如增加成本重试、延迟提交等, 以保证系统的最终一致性.

4 轻量级原子提交协议

在介绍协议细节之前, 本节首先对其一致性语义作出声明: 本协议实现的是最终一致性, 而非传统两阶段提交的强一致性. 这一设计选择基于以下考量: (1) 区块链的异步确认特性: 区块链交易需要等待多个区块确认才能达到“最终确定性”, 这个过程通常需要数秒到数分钟, 与 2PC 的同步提交范式存在根本性差异. (2) 不可回滚性约束: 已上链的交易无法像数据库事务那样回滚, 这使得传统 2PC 的 PREPARE-COMMIT-ROLLBACK 语义无法在区块链上实现. (3) 性能与可用性要求: 如果强制实现同步原子性, 需要阻塞数据库事务直到区块链确认完成, 这会将系统吞吐量降至区块链的水平, 完全丧失混合架构的性能优势.

因此, 本协议通过以下机制实现可靠的最终一致性: (1) Outbox pattern: 在数据库端确保业务操作与事件发布的原子性; (2) 可靠事件传递: 通过 CDC 和 Kafka 保证事件的 at-least-once 投递; (3) 幂等性设计: 智能合约通过 $\mathrm{tx}_{\mathrm{id}}$ 去重机制支持安全重试; (4) 补偿机制: 检测并修复失败的同步操作. 在正常情况下, 系统能够在有界时间内 (P99 < 15 s) 达到一致状态, 对于极少数需要人工介入的情况 (<0.1%), 通过监控告警机制及时处理.

4.1 跨系统原子事务的形式化定义与映射

保障跨系统事务原子性的核心挑战在于数据库的 ACID 原子事务与区块链的最终一致性模型之间的矛盾. 为系统性地解决这一问题, 本节首先给出跨系统原子事务的形式化定义, 明确在混合存储环境下“原子性”的语义; 然后阐述如何在不引入重量级两阶段提交 (2PC) 的前提下, 实现轻量级的原子映射.

定义 1. 数据库事务. 一个数据库事务 T_{db} 可表示为一个有序操作序列: $T_{\mathrm{db}} = \langle \mathrm{op}_1, \mathrm{op}_2, \ldots, \mathrm{op}_n \rangle$, 其中每个操作 $\mathrm{op}_i \in \{\mathrm{INSERT}, \mathrm{UPDATE}, \mathrm{DELETE}\}$, 且满足 ACID 性质. 特别地, 事务 T_{db} 具有明确的开始标记 $\mathrm{BEGIN}(T_{\mathrm{db}})$ 和结束标记 $\mathrm{COMMIT}(T_{\mathrm{db}})$ 或 $\mathrm{ABORT}(T_{\mathrm{db}})$. 只有 COMMIT 的事务才需要同步到区块链.

定义 2. 原子提交单元. 对于一个已提交的数据库事务 T_{db}, 其对应的原子提交单元 ACU 定义为: ACU = (tx_{id}, HLC, Ops, DP), 其中, tx_{id} 为全局唯一的事务标识符; HLC 为混合逻辑时钟时间戳, 用于建立全局因果顺序; Ops = $\{op_1, op_2, \ldots, op_n\}$ 为操作集合, 每个操作编码为 (type, table, key, value_hash); DP 为删除证明, 当且仅当 Ops 中包含 DELETE 操作时非空.

轻量级原子映射的核心原则是: 将数据库的一个原子事务 T_{db} 作为不可分割的整体映射为区块链上的单个交易 T_{bc}, 利用区块链交易自身的原子性来保障跨系统的原子性, 而无需在数据库和区块链之间引入额外的协调机制. 形式化地, 定义原子映射函数 $M: T_{db} \to T_{bc}$, 满足: (1) 完整性: T_{bc} 包含 T_{db} 的所有操作信息; (2) 有序性: T_{bc} 中操作的相对顺序与 T_{db} 一致; (3) 原子性等价: committed(T_{bc}) ⟺ 所有 op_i 的效果都反映在链上状态, aborted(T_{bc}) ⟺ 所有 op_i 的效果都不反映在链上状态.

与传统 2PC 协议相比, 本协议的“轻量级”体现在中间件层协调开销的显著降低, 具体包含 3 个方面. 第一, 无阻塞单向流: 数据库事务正常提交后即释放锁资源, CDC 以异步、非侵入方式捕获变更并通过事件流传递到 TDSM, 整个过程对数据库性能影响微乎其微; 数据库无需等待区块链确认, 也无需感知区块链的存在, 避免了 2PC 中 prepare 阶段的全局阻塞. 第二, 无中心化协调者: 传统 2PC 需要维护全局事务管理器, 负责协调所有参与者的状态并在故障时执行恢复. 本协议中, TDSM 仅作为事件消费者和批处理调度器, 不维护跨系统的事务状态, 故障恢复依赖 CDC 日志的可回放性和区块链的不可篡改性, 显著降低了系统复杂度. 第三, 利用已有原子性: 本协议通过“聚合+映射”模式, 将数据库的本地原子性和区块链的交易原子性串联, 形成端到端的原子性保障, 无需引入额外的分布式协调协议. 上述设计使得中间件层的处理延迟和资源消耗相比传统 2PC 方案大幅降低, 第 7 节的消融实验将验证这一效率优势. 需要明确的是, 本协议不保证强原子性, 而是通过上述机制实现最终一致性加补偿. 这种设计在分布式系统中被广泛采用, 已被证明在大规模系统中是可行且高效的.

在具体实现上, TDSM 为每个 ACU 生成统一的编码格式, 以适配区块链的交易结构. 编码包含 4 个层次, 如图 4 所示: (1) 元数据层, 包含 tx_{id}、源数据库标识、HLC 时间戳、操作数量; (2) 操作层, 对于 INSERT 和 UPDATE 操作, 编码为<type, table_id, pk, cols_hash, data_ptr>, 其中 cols_hash 为被修改列的哈希值, data_ptr 为链下完整数据的存储指针; 对于 DELETE 操作, 编码为<type, table_id, pk, proof_idx>, 其中 proof_idx 指向撤销证明在 DP 中的位置; (3) 依赖层, 记录事务间的读写依赖关系, 包括当前事务所依赖的前置事务标识, 智能合约据此验证 ACU 的执行顺序是否满足可串行化条件; (4) 签名层, TDSM 对整个编码进行数字签名, 保证数据来源的可追溯性. 该编码格式兼顾了空间效率和验证效率.

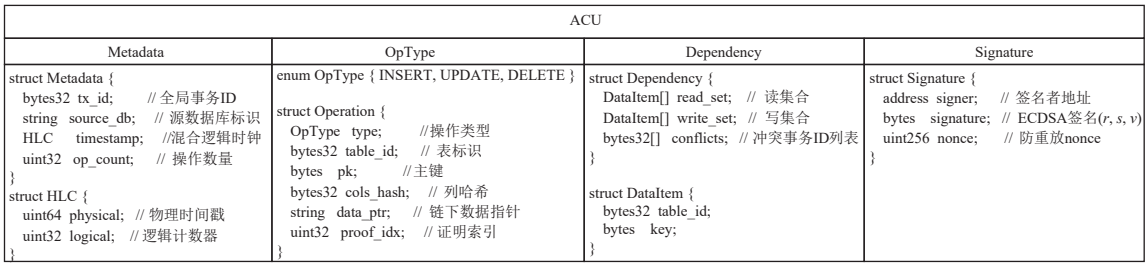


图 4 ACU 数据结构示意图

值得说明的是, 上述 ACU 编码设计与第 3 节的处理流程协同保证了因果相关事务的上链顺序. 对于存在因果依赖的事务 (如 T_1 先更新了某键值, T_2 随后基于 T_1 的写入结果进行再次更新), HLC 的因果性保证了 HLC(T_1) < HLC(T_2), 全局时序协调器据此将 T_1 的事件排在 T_2 之前传递给事务协调引擎. 在批处理阶段, 调度器按 HLC 时间戳对 ACU 排序后打包成批次, 批次按生成顺序依次提交到链上, 智能合约按序执行批次中的每个 ACU (第⑧步). 即使 T_1 与 T_2 被分配到不同批次, T_1 所在批次也必然先于 T_2 所在批次被提交. 此外, ACU 编码中的依赖层显式记录了 T_2 对 T_1 的读写依赖, 智能合约可据此对执行顺序进行可串行化验证, 确保链上状态的一致性.

4.2 事件捕获与补偿机制

为确保数据库事务与 CDC 事件发布的原子性, TDSM 在数据库端采用 outbox pattern. 该模式通过在业务数据库中创建专用的 outbox 事件表, 在同一数据库事务中同时写入业务数据和事件记录, 利用数据库的 ACID 特性保证二者的原子性.

具体实现如下. 首先, 在源数据库中创建 outbox_events 表, 包含字段: event_id (主键)、aggregate_id (业务主键)、event_type (INSERT/UPDATE/DELETE)、event_payload (变更数据的 JSON 编码)、tx_{id} (全局事务 ID, 基于 HLC 时间戳)、status (PENDING/PUBLISHED/CONFIRMED)、created_at 和 confirmed_at. 应用程序 (或数据库触发器) 在执行业务操作的同一事务中, 原子性地向 outbox_events 表插入事件记录. CDC 连接器配置为监听 outbox_events 表的变更, 捕获新插入的事件并发布到 Kafka. 这保证了: 若业务操作成功, 事件必然被记录; 若业务操作失败回滚, 事件也不会被记录. TDSM 消费 Kafka 事件后, 通过独立的事务更新事件状态: 将 status 从 PENDING 更新为 PUBLISHED (表示已被 TDSM 接收), 并在区块链交易确认后进一步更新为 CONFIRMED (表示已成功上链). 这种状态追踪机制支持故障恢复和一致性审计: 系统可以定期扫描长时间处于 PENDING 状态的事件, 识别潜在的同步失败并触发重试.

尽管 outbox pattern 保证了链下的原子性, 但区块链交易仍可能因为交易成本不足、网络超时、nonce 冲突或合约验证失败等原因失败. TDSM 通过分类补偿策略保障最终一致性.

针对不同失败类型, 系统采取差异化的补偿策略: (1) 交易成本不足: 检测到交易因交易成本不足失败或长时间挂起后, 尝试重新提交, 最多重试 3 次; (2) nonce 冲突: 当交易因 nonce 错误被拒绝时, 重新同步账户 nonce 后重试, 最多 5 次; (3) 网络超时: 对于 RPC 调用超时的情况, 采用指数退避策略 (延迟 2^n s) 重试, 最多 10 次; (4) 合约验证失败: 若交易因智能合约约束检查失败而回滚, 则不再重试, 将该事件记录到死信队列 (dead letter queue, DLQ) 并触发人工介入告警; (5) 节点不可达: 当主区块链节点连接失败时, 自动切换到备用节点, 无重试次数限制.

所有补偿操作都利用智能合约的幂等性设计: 每个 ACU 携带唯一的 tx_{id}, 合约在执行前检查该 tx_{id} 是否已处理 (通过维护 processedTxs 映射), 若已存在则直接返回成功, 避免重复执行. 这保证了即使重试多次, 数据也不会被重复写入. 对于超过最大重试次数仍失败的事件, 系统将其记录到 DLQ 表, 包含完整的事件数据、失败原因、重试历史和错误堆栈.

系统还实现了持续的一致性监控: 定期查询 outbox_events 表, 统计各状态事件的数量和最老的挂起事件的等待时间. 当挂起事件累积超过 1 000 条或最老事件等待超过 1 h 时, 触发告警提示可能的同步延迟. 这种主动监控机制能够及早发现并处理潜在的一致性问题.

4.3 基于稀疏默克尔树的可验证撤销证明

在可信审计场景中, INSERT 和 UPDATE 操作的验证较为直观: 操作产生的新值直接编码在链上 ACU 中, 审计方读取链上状态即可验证特定时刻的数据内容. DELETE 操作则面临不同的审计验证需求. 一方面, 审计方可能需要验证某条记录确实在指定时间被删除; 另一方面, 也可能需要验证某条记录未被误删或恶意删除. 前者要求成员证明, 后者要求非成员证明. 虽然区块链可通过墓碑标记等机制逐条记录删除意图, 但分散的墓碑标记不具备对删除集合的整体性密码学约束: 验证某条记录是否在删除集合中需要遍历所有记录 (O(k)), 且无法检测删除集合本身是否被篡改 (如遗漏某次删除或事后插入伪造的墓碑). 为此, 本节提出基于稀疏默克尔树的可验证撤销证明机制, 通过对每个时间窗口内的删除集合构建密码学承诺, 以 O(log$N$) 的代价同时支撑成员证明与非成员证明, 并保证删除集合一经锚定即不可篡改.

撤销证明的构建分为 3 个阶段. 第 1 阶段 (缓冲期): TDSM 维护一个删除缓冲区 Buffer_del, 收集一个时间窗口内所有 ACU 中的 DELETE 操作, 记录被删除记录的唯一标识符 rid. 第 2 阶段 (树构建): 窗口结束时, TDSM 基于 Buffer_del 中的所有 rid 构建一棵 SMT. 具体地, 对于深度为 d=256 的二叉 SMT, 每个 rid 的哈希值 H(rid) 作为从根到叶的路径编码: 第 i 位为 0 则向左, 为 1 则向右. 叶节点存储 H(rid), 内部节点存储其两个子节点哈希的哈希. 对于 Buffer_del 中不存在的路径 (空节点), 使用预定义的空哈希值 empty_hash 填充. SMT 的根哈希 $root_k$ 唯一

地代表这一批删除操作的集合. 第 3 阶段 (链上锚定): TDSM 将 $root_k$ 嵌入到这一时间窗口内所有包含 DELETE 的 ACU 中, 并在智能合约中注册 $root_k$ 与时间窗口的映射关系. 完整的 SMT 结构存储在链下, 以供后续查询时生成 Merkle 证明.

为确保 DELETE 操作的正确性, TDSM 明确定义: SMT 撤销证明撤销的是截止到 DELETE 操作执行时刻该记录的最新有效版本. 为处理网络延迟导致的操作乱序问题, 事务协调引擎为每个 key 维护操作序列及 HLC 时间戳. 当 DELETE 操作到达时, 若检测到存在 HLC 更早但尚未确认的前序 UPDATE 操作, 则将该 DELETE 暂存在待定队列, 直到前序操作确认后再处理. 这一依赖延迟机制确保 DELETE 始终撤销正确版本, 额外延迟约 5–15 s, 对审计型应用可接受.

为支持删除后重新插入同一 key 的场景, TDSM 采用版本化 rid: rid = H(table_id || primary_key || version_counter), 其中 version_counter 为每个 (table, key) 维护的单调递增计数器. 每次插入或删除后递增计数器, 使不同版本在 SMT 中占据独立叶节点. 例如, “INSERT(key=1) → DELETE(key=1) → INSERT(key=1)”产生 rid_1 和 rid_2, 分别对应不同版本的生命周期, 避免冲突的同时完整记录历史.

在历史可验证性方面, TDSM 通过时间窗口化的 SMT 快照链保障历史删除状态的可追溯性. 每个窗口 k 的 $root_k$ 锚定在链上, 完整 SMT 存储在链下可公开查询. 智能合约维护有序的 $(\text{window}_{\text{id}}, root_k, \text{timestamp}_k)$ 序列. 审计方验证“记录 rid 是否在时间 t 被删除”时, 定位时间窗口 k, 从合约查询 $root_k$, 请求 Merkle 证明 π_k(rid) 并本地验证. 即使 TDSM 下线, 任何持有 SMT 快照的第三方均可独立验证, 实现去中心化审计.

撤销证明的验证协议如下: 当审计方 (或任意第三方) 需要验证某个记录 rid 在特定时间窗口 k 内是否被删除时, 可向 TDSM 请求 Merkle 证明 π_k(rid). π_k(rid) 由从 rid 对应的叶节点到 $root_k$ 路径上所有兄弟节点的哈希组成, 其大小为 O(log$N$), 其中 $N=2^d$ 为 SMT 的最大容量. 验证者执行以下步骤: (1) 获取链上锚定的 $root_k$ (通过查询智能合约); (2) 计算叶节点哈希 leaf_hash = H(rid); (3) 使用 π_k(rid) 中的兄弟节点哈希, 自底向上重新计算根哈希 $root'$; (4) 检查 $root' == root_k$. 若相等, 则证明 rid 确实被包含在这批删除操作中; 若不等, 则证明 rid 未被删除或证明无效.

撤销证明机制的安全性基于哈希函数的单向性和抗碰撞性. 攻击者无法在不破解哈希函数的前提下, 为一个未被删除的 rid 伪造有效的 Merkle 证明, 也无法在不改变 $root_k$ 的前提下修改已提交的删除集合. 由于 $root_k$ 锚定在不可篡改的区块链上, 任何对历史删除记录的篡改都会被检测到. 在性能方面, SMT 的空间复杂度为 $O(k \cdot \log N)$, 其中 k 为实际删除的记录数; 证明生成和验证的时间复杂度均为 $O(\log N)$. 相比于为每个 DELETE 单独生成一棵 Merkle 树, 本方案通过批量构建 SMT, 显著降低了链上存储成本 (只需存储一个 $root_k$ 而非 k 个树根) 和验证成本.

4.4 原子性保障的扩展机制

尽管第 4.1 节所述的原子映射原则在理论上保障了跨系统原子性, 但在实际系统中仍面临两个扩展性挑战: 一是大事务问题, 当数据库事务包含大量操作时, 编码后的 ACU 可能超过区块链的资源限制; 二是故障场景下的一致性恢复, 需要在 TDSM 崩溃重启后, 准确识别哪些 ACU 已成功提交到链上, 哪些需要重新提交. 本节阐述针对这两个挑战的扩展机制.

对于大事务 T_{db}, 其 ACU 超过单笔区块链交易的资源限制时, TDSM 采用分片协调协议 (sharding coordination protocol) 将 ACU 拆分为多个子 ACU, 并通过链上协调合约保障原子性. 协议分为 3 个阶段, 如算法 1 所示. 阶段 1 (声明): TDSM 首先在链上创建一个协调记录 $\text{Coord}_{\text{tx}}(\text{tx}_{\text{id}}$, shard_count, hash), 声明即将提交的大事务被拆分为 shard_count 个子 ACU, 并预承诺每个子 ACU 的哈希值 hash, 防止后续篡改. 阶段 2 (并发提交): TDSM 将各子 ACU 并发提交到区块链, 每个子 ACU 在其元数据中携带 $(\text{tx}_{\text{id}}, \text{shard}_{\text{id}}$, shard_count) 信息. 智能合约接收到子 ACU 后, 首先验证其哈希是否与 Coord_{tx} 中预承诺的 hash 一致, 然后将子 ACU 存储在待确认状态, 并更新 Coord_{tx} 的接收计数器. 阶段 3 (确认): 当 Coord_{tx} 的接收计数器达到 shard_count 时, 智能合约自动触发 finalize 操作, 将 tx_{id} 标记为已完成 (finalized=true), 此时所有子 ACU 的状态变更才对外可见. 若在超时时间内未收齐所有子 ACU, 则 Coord_{tx} 被标记为失败, 所有已接收的子 ACU 被回滚, 保证了原子性.

算法 1. 大事务分片协调算法.

输入: 大事务 ACU large_tx, 资源限制 gas_limit;
输出: 是否成功提交 success.

```
1. function COORDINATE_LARGE_TX(large_tx, gas_limit)
2.    // 阶段 1: 评估与拆分
3.    estimated_gas ← ESTIMATE_GAS(large_tx)
4.    if estimated_gas ≤ gas_limit × 0.8 then
5.       return SUBMIT_SINGLE(large_tx)
6.    end if
7.    // 智能拆分操作序列
8.    sub_acus ← [], current_sub ← EMPTY_ACU()
9.    for each op in large_tx.operations do
10.      if current_sub.gas + op.gas > gas_limit × 0.8 then
11.         sub_acus.APPEND(current_sub), current_sub ← EMPTY_ACU()
12.      end if
13.      current_sub.ADD(op)
14.   end for
15.   sub_acus.APPEND(current_sub)
16.   // 阶段 2: 部署协调合约并发提交
17.   coordinator ← DEPLOY_COORDINATOR(large_tx.tx_id, sub_acus.LENGTH, 30 s)
18.   for i ← 0 to sub_acus.LENGTH – 1 do
19.      THREAD_POOL.SUBMIT(SUBMIT_SUB_ACU(sub_acus[i], coordinator, i))
20.   end for
21.   // 阶段 3: 等待确认
22.   status ← coordinator.WAIT_UNTIL_FINALIZED(60 s)
23.   if status == COMMITTED then return true
24.   else TRIGGER_COMPENSATION(large_tx, status), return false
25.   end if
26. end function
```

为实现故障恢复, TDSM 采用分层检查点机制维护系统的一致性状态. 检查点分为两类: 轻量级检查点 (light checkpoint) 和完整检查点 (full checkpoint). 轻量级检查点每处理一个批次 (详见第 5 节) 后生成, 记录: (1) 当前 Kafka 消费 offset; (2) 已提交到链上的最后一个批次 ID; (3) 删除缓冲区的状态 (已缓冲的 rid 列表和当前窗口的开始时间). 轻量级检查点的生成成本极低 (仅写入数百字节的元数据), 可以高频率执行而不影响系统吞吐. 完整检查点每隔 T 个轻量级检查点 (如 T=100) 或在 SMT 树根提交到链上后生成, 额外记录: (1) 所有待确认的大事务的 Coord_{tx} 状态; (2) 最近 N 个块的链上确认状态; (3) 当前 SMT 的序列化快照. 完整检查点的生成成本较高, 但提供了更强的一致性保障.

当 TDSM 从故障中恢复时, 执行如下恢复协议. 第 1 步, 加载最近的完整检查点, 恢复系统的全局状态 (包括 SMT 和大事务协调状态). 第 2 步, 从完整检查点到最新的轻量级检查点之间, 依次回放各轻量级检查点记录的批次提交状态, 通过查询链上智能合约确认每个批次是否已成功上链: 若已上链, 则更新本地状态并继续; 若未上链或状态不明, 则将该批次加入重试队列. 第 3 步, 从最新的轻量级检查点记录的 Kafka offset 开始, 重新消费 CDC

事件流, 重新构建未完成的 ACU 并提交. 由于 ACU 具有全局唯一的 tx_{id}, 智能合约会自动去重重复提交的 ACU, 保证了幂等性. 第 4 步, 对于重试队列中的批次, TDSM 根据故障原因采取不同策略: 若为 Gas 不足, 则增加 Gas price 后重试; 若为网络超时, 则延迟后重试; 若为语义错误 (如违反智能合约约束), 则记录到失败日志并触发人工介入. 该恢复协议在最坏情况下可能导致部分批次的重复提交, 但通过智能合约的幂等性检查和 tx_{id} 去重, 保证了系统最终达到与故障前完全一致的状态, 满足了混合系统的最终一致性要求.

定理 1. 跨系统原子性. 对于任意数据库事务 T_{db}, 在 TDSM 的轻量级原子提交协议和补偿机制下, 系统最终达到一致状态: T_{db} 在数据库的提交状态与其在区块链上的可见性最终保持一致. 即, 若 T_{db} 在数据库已提交, 则在有界时间内 (正常情况下 P99 < 15 s, 故障情况下通过补偿机制最终完成), T_{db} 的所有操作在链上可见; 若 T_{db} 在数据库回滚, 则其操作在链上不可见.

证明: 根据 ACU 的映射定义, 需证明 T_{db} 的所有操作要么全部可见, 要么全部不可见. 分 3 种情况讨论. (1) 单笔交易情况: 若 ACU 未超过资源限制, 则 T_{db} 被映射为单笔链上交易 T_{bc}, 区块链交易的原子性直接保证了该性质. (2) 分片交易情况: 若 ACU 被拆分为多个子 ACU, 根据第 4.4 节的分片协调协议, 只有当 $Coord_{tx}$.finalized = true 时, 智能合约的可见性查询才返回 true. $Coord_{tx}$.finalized = true 当且仅当所有 shard_count 个子 ACU 均已接收且哈希验证通过, 因此 T_{db} 的所有操作全部可见. 反之, 若任一子 ACU 未收到或验证失败, 则 $Coord_{tx}$.finalized = false, 查询返回 false, T_{db} 的所有操作均不可见. (3) 故障恢复情况: 根据第 4.2 节的恢复协议, 未完成的 ACU 会从 Kafka 重新消费并重新提交, 智能合约通过 tx_{id} 去重保证了幂等性, 最终必达到情况 (1) 或 (2) 的稳定状态. 综合 3 种情况, 原子性得证. 此外, 对于区块链交易失败的情况, 补偿机制通过自动重试和幂等性保证确保操作最终成功提交, 或在多次重试失败后将事件记录到 DLQ 并触发人工介入. 由于 outbox pattern 保证了数据库操作与事件记录的原子性, CDC 和 Kafka 保证了事件的 at-least-once 投递, 智能合约的幂等性保证了重试的安全性, 系统最终必然达到一致状态, 满足最终一致性要求.

5 基于资源感知的自适应批处理机制

TDSM 需要将数据库变更同步到区块链, 而区块链的交易提交具有不可忽略的成本和延迟. 一方面, 数据库的变更频率可达数千至数万 TPS, 而主流区块链网络的吞吐量通常在数十至数百 TPS 之间, 两者之间存在数个数量级的性能差距. 另一方面, 区块链交易的成本和延迟受网络状态动态影响: 在公链场景下, Gas 价格会随网络拥堵程度大幅波动; 在联盟链场景下, 节点的 TPS 配额和网络延迟也会影响批处理决策. 如果为每个数据库事务都生成一笔链上交易, 将导致极高的同步成本和网络压力. 因此, TDSM 采用批处理机制, 将多个原子提交单元聚合为一个批次后统一提交到区块链. 然而, 批处理策略面临两个核心挑战: 第一, 不同类型的区块链网络资源约束差异巨大, 如公链强调交易成本优化, 而联盟链更关注 TPS 利用率和延迟; 第二, 区块链网络状态动态变化, 静态的批处理策略难以适应这些变化, 可能导致在成本高峰期产生不必要的开销, 或在低成本窗口未能充分利用网络容量.

为系统性地解决这些挑战, 本节提出了资源感知的自适应批处理机制. 该机制的核心思想是: 通过建立统一的跨链资源抽象模型, 实时监控目标区块链的资源状态, 并基于成本效益函数动态调整批次大小和提交时机, 在延迟与资源开销之间取得动态平衡. 本节首先介绍跨链资源模型和分层监控架构, 然后阐述自适应批处理调度算法的设计.

5.1 跨链资源模型与感知机制

不同类型的区块链网络在资源约束和成本模型上存在显著差异. 公链采用基于市场的定价机制, 交易费用随网络拥堵程度动态波动; 联盟链通常没有交易费用, 但存在 TPS 配额和权限限制; 私有链的交易成本固定且很低, 但需要关注区块确认延迟. 为了使 TDSM 能够适配不同的区块链网络, 本文提出了统一的跨链资源抽象模型.

资源向量定义. 对于目标区块链网络, 其在时刻 t 的资源状态定义为四元组 $R(t) = (C(t), T(t), Q(t), V(t))$, 其中, $C(t)$ 为交易成本参数, 表示提交一笔交易的资源开销 (在公链如以太坊中表现为 Gas 价格, 在联盟链中表现为配额消耗或固定成本); $T(t)$ 为吞吐量上限, 表示网络当前的 TPS 容量; $Q(t)$ 为资源配额, 表示当前可用的交易配额 (联

盟链场景下为组织配额, 公链场景下为区块 Gas limit); $V(t)$ 为波动性指标, 表示资源状态的变化趋势 (如 Gas 价格的标准差和预测趋势).

为实现对异构区块链网络的资源感知, 本文设计了 3 层监控架构, 如图 5 所示. 第 1 层为基础资源监控, 定义了统一的资源监控接口 BlockchainResourceMonitor, 该接口提供 get_current_cost()、get_throughput_limit() 和 predict_resource_trend() 这 3 个核心方法, 分别用于获取当前成本、吞吐量上限和预测资源趋势. 第 2 层为特定链监控器, 针对不同类型的区块链实现具体的监控逻辑. 对于以太坊公链, EthereumMonitor 通过查询节点的 eth_gasPrice 接口获取实时 Gas 价格, 并通过分析交易池 (Mempool) 中待处理交易的数量和 Gas 价格分布来预测价格走势; 对于 Fabric 联盟链, FabricMonitor 查询系统配置获取 TPS 上限和组织配额, 并监控 Orderer 的队列长度来评估网络负载. 第 3 层为策略选择层, 根据监控到的资源特性自动选择优化目标: 当 $V(t)$ 较高 (表示资源状态波动剧烈) 时, 系统采用成本优化策略; 当 $V(t)$ 较低 (表示资源状态稳定) 时, 系统采用吞吐量优化策略.

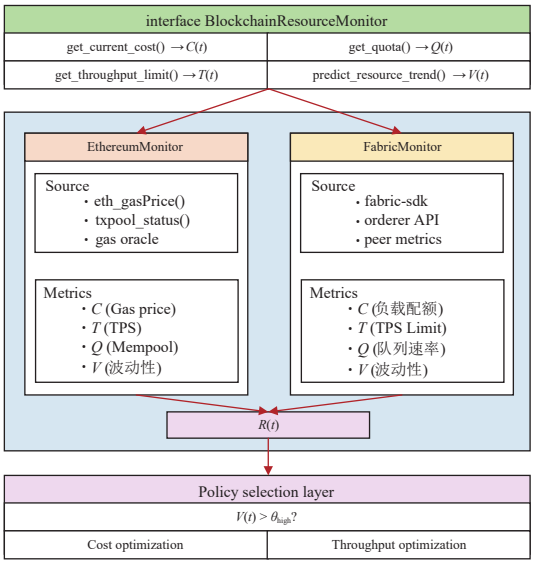


图 5　分层资源监控架构

TDSM 的批处理调度器持续监控资源状态, 并维护历史数据以支持趋势预测. 对于公链场景, 调度器每隔固定时间窗口查询一次交易成本, 并计算最近 N 个样本的均值 $\overline{C}$ 和标准差 σ_C. 当检测到当前成本 $C(t) < \overline{C} - \sigma_C$ 时, 系统识别为低成本窗口, 并倾向于立即提交较大批次以降低平均成本; 当 $C(t) > \overline{C} + \sigma_C$ 时, 系统识别为高成本期, 并延迟提交或减小批次大小. 对于联盟链场景, 调度器监控当前配额使用率 $u(t) = Q_{\text{used}}(t)/Q(t)$. 当 $u(t) < \theta_{\text{low}}$ 时, 表示配额充裕, 系统倾向于立即提交; 当 $u(t) > \theta_{\text{high}}$ (如 0.8) 时, 表示配额紧张, 系统延迟提交以避免超出配额限制.

通过上述资源模型和监控架构, TDSM 能够统一感知不同区块链网络的资源状态, 为后续的自适应批处理调度提供实时信息.

5.2　自适应批处理调度算法

批处理调度的核心目标是在满足业务延迟约束的前提下, 最小化资源开销. 然而, 不同类型的区块链网络具有不同的资源约束和优化目标. 为此, 本文设计了双模式自适应批处理算法, 根据区块链类型和实时资源状态动态调整决策策略.

批处理决策问题形式化定义如下: 给定当前队列中的待处理 ACU 数量 n_{queue} 和资源状态 $R(t)$, 决策变量为批次大小 n^* 和提交时机 τ^* (立即或延迟 Δt), 优化目标为最小化加权成本:

$$\min_{n,\tau} \{\alpha \cdot Cost(n, R(t)) + \beta \cdot Delay(n, n_{\text{queue}}, \tau)\}.$$

约束条件包括: $n \leqslant n_{\text{queue}}$ (批次不超过队列长度); $n \cdot \text{size}_{\text{tx}} \leqslant \text{Limit}$ (不超过区块链的大小限制); $Delay \leqslant \text{SLA}$ (满足延迟服务水平协议). 其中, 权重 α 和 β 根据区块链类型自动调整: 对于公链, α 较大以重视成本优化; 对于联盟链, β 较大以重视延迟优化.

双模式调度算法. 算法根据资源波动性 $V(t)$ 自动切换模式. 模式 A: 成本驱动 (适用于公链). 算法实时计算当前成本 $C(t)$ 在历史分布中的分位数 $p(t)$. 当 $p(t) < 0.2$ 时 (处于低成本区间), 算法立即提交大批次 $n^* = \min(n_{\text{queue}}, n_{\max})$ 以抓住低成本窗口; 当 $p(t) > 0.8$ 时 (处于高成本区间), 算法延迟提交 $\tau^* = \Delta t$ 以等待成本下降; 当 $0.2 \leqslant p(t) \leqslant 0.8$ (中等成本), 算法根据队列积压程度 n_{queue} 决策: $n_{\text{queue}} > \theta_{\text{queue}}$, 则提交中等批次 $n^* = \min(n_{\text{queue}}, n_{\text{mid}})$ 以缓解积压; 否则继续等待. 模式 B: 吞吐量驱动 (适用于联盟链/私有链). 算法估算当前网络可用 TPS 容量 $T_{\text{avail}}(t) = T(t) - L(t)$, 其中 $L(t)$ 为当前网络负载. 当 $T_{\text{avail}}(t) > \theta_{\text{TPS}}$ 时 (网络空闲), 算法立即提交大批次 $n^* = \min(n_{\text{queue}}, T_{\text{avail}}(t) \cdot w, n_{\max})$, 其中 w 为批处理时间窗口; 当 $T_{\text{avail}}(t) \leqslant \theta_{\text{TPS}}$ 时 (网络繁忙), 算法根据紧急程度决策: 若 $n_{\text{queue}} > \theta_{\text{urgent}}$, 提交小批次 $n^* = \min(n_{\text{queue}}, n_{\text{small}})$; 否则延迟提交 $\tau^* = \Delta t$.

算法 2. 双模式自适应批处理调度算法.

输入: 待处理 ACU 队列 acu_queue, 资源状态 $R(t) = (C, T, Q, V)$;
输出: 批次大小与提交时机 (n^*, τ^*).

```
1. function ADAPTIVE_BATCH_SCHEDULER(acu_queue, R(t))
2.     n_queue ← acu_queue.SIZE()
3.     // 根据资源波动性选择模式
4.     if V(t) > θ_high then return COST_DRIVEN_MODE(n_queue, C(t))
5.     else return THROUGHPUT_DRIVEN_MODE(n_queue, T(t))
6.     end if
7. end function
8. function COST_DRIVEN_MODE(n_queue, C(t)) // 模式 A: 成本优化
9.     p(t) ← PERCENTILE(C(t), history) // 计算成本分位数
10.    if p(t) < 0.2 then // 低成本窗口: 提交大批次
11.        return (min(n_queue, n_max), IMMEDIATE)
12.    else if p(t) > 0.8 then // 高成本期: 延迟提交
13.        return (0, DELAY)
14.    else // 中等成本: 根据队列积压决策
15.        if n_queue > θ_queue then return (min(n_queue, n_mid), IMMEDIATE)
16.        else return (0, DELAY)
17.        end if
18.    end if
19. end function
20. function THROUGHPUT_DRIVEN_MODE(n_queue, T(t)) // 模式 B: 吞吐量优化
21.    T_avail ← T(t) – CURRENT_LOAD() // 计算可用容量
22.    if T_avail > θ_TPS then // 网络空闲: 提交大批次
23.        return (min(n_queue, T_avail×w, n_max), IMMEDIATE)
24.    else if n_queue > θ_urgent then // 队列积压: 提交小批次
25.        return (min(n_queue, n_small), IMMEDIATE)
26.    else return (0, DELAY) // 等待空闲
```

27. **end if**
28. **end function**
29. **function** PARAMETER_LEARNING() // 参数自适应学习
30. $\eta_{\text{cost}} \leftarrow \Sigma(actual\_cost)/\Sigma(predicted\_cost)$
31. $\eta_{\text{delay}} \leftarrow \Sigma(actual\_delay)/\Sigma(\text{SLA})$
32. // 动态调整权重
33. **if** $\eta_{\text{cost}} > 1.2$ **then** $\alpha \leftarrow \min(\alpha + 0.05, 0.9)$ **end if**
34. **if** $\eta_{\text{delay}} > 1.2$ **then** $\beta \leftarrow \min(\beta + 0.05, 0.9)$ **end if**
35. // 动态调整阈值
36. **if** CONSECUTIVE_LOW_COST > 5 **then** $n_{\max} \leftarrow n_{\max} + 50$ **end if**
37. **if** CONSECUTIVE_HIGH_COST > 5 **then** $n_{\text{mid}} \leftarrow n_{\text{mid}} - 20$ **end if**
38. **end function**

参数自适应机制: 为避免静态阈值带来的次优决策, 算法引入了参数学习模块. 该模块维护历史决策和实际效果的记录, 包括预测成本、实际成本、提交延迟和吞吐量等. 每处理 100 个批次后, 模块评估最近决策的成本效率 $\eta_{\text{cost}} = \sum actual\_cost / \sum n$ 和延迟惩罚 $\eta_{\text{delay}} = \sum delay / \sum n$. 若 η_{cost} 高于目标值, 系统增大权重 α 以更重视成本; 若 η_{delay} 高于 SLA 要求, 系统增大权重 β 以更重视延迟. 此外, 算法动态调整批次大小阈值 $n_{\max}$、n_{mid} 和 n_{small}: 在连续观察到低成本窗口时, 增大 $n_{\max}$ 以充分利用网络容量; 在观察到频繁的高成本期时, 减小 n_{mid} 以降低风险.

资源监控和决策计算的时间复杂度为 $O(1)$ (基于近期历史数据的统计计算). 参数学习模块每 100 个批次触发一次, 时间复杂度为 $O(m)$, 其中 m 为历史窗口大小 (通常 $m = 100$). 因此, 算法的整体开销极低, 不会成为系统性能瓶颈. 空间复杂度方面, 算法需要维护历史资源状态和决策记录, 空间开销为 $O(m+k)$, 其中 k 为待处理队列的最大长度. 在实际部署中, m 和 k 通常在数百到数千的量级, 内存占用可忽略不计.

6 系统实现

为验证 TDSM 架构及其核心机制的有效性, 本文构建了 TDSM 的原型系统. 该系统实现了第 3 节提出的 4 层架构, 并在自适应同步层中实现了第 4 节的轻量级原子提交协议和第 5 节的成本感知自适应批处理机制.

6.1 原型系统架构

TDSM 原型系统由数据源、CDC 连接器、事件流平台、自适应同步层和区块链交互层组成. 本文选择 MySQL 8.0 作为数据源数据库, 使用 Debezium 作为 CDC 连接器捕获 binlog 变更事件, 采用 Apache Kafka 作为事件流平台保证事件的有序传输和持久化, 自适应同步层使用 Java 实现, 区块链层部署 Solidity 智能合约.

自适应同步层是 TDSM 的核心, 包含 5 个关键模块: 全局时序协调器、事务协调引擎、批处理调度器、SMT 构建器和一致性管理器. 为了避免各模块互相抢占计算资源, 将它们绑定到不同的线程池上独立运行. 全局时序协调器与事务协调引擎实现在同一进程中. 从功能上来说, 这两个模块是紧密结合的: 时序协调器需要为每个事件分配 HLC 时间戳, 而事务协调引擎需要立即使用这些时间戳来维护事务的全局顺序. 如果将它们拆分到不同进程, 意味着每个事件的处理都需要跨进程通信, 这将显著增加延迟开销. 类似地, SMT 构建器与批处理调度器也紧密集成: SMT 构建完成后需要立即将根哈希嵌入待提交批次, 频繁的跨进程通信将成为性能瓶颈.

基于第 4 节和第 5 节的设计, 本文在系统实现中做出了以下考量. 首先, 对于 DELETE 操作的撤销证明生成, 本文采用周期性批量构建而非实时构建的方式. 这是因为实时为每个 DELETE 操作构建完整的 SMT 将产生大量的树更新开销, 而通过设置缓冲窗口收集一批 DELETE 操作后批量构建, 可以显著降低平均构建成本. 虽然这会增加 DELETE 操作的证明延迟, 但对于大多数业务场景而言, 删除操作的审计时效性要求低于插入和更新操作, 这一权衡是可接受的. 其次, 对于大事务的分片协调, 本文在链上部署了专门的协调合约来追踪各子 ACU 的提交

状态. 这避免了在中间件层维护复杂的分布式状态, 利用了区块链的共识机制来保证状态的一致性.

6.2 关键机制实现

SMT 构建器作为自适应同步层的独立子模块, 通过生产者-消费者模式与事务协调引擎集成. 事务协调引擎将 DELETE 事件推送到内存队列, SMT 构建器从队列中批量消费事件并更新树. 系统启动时创建专用的 SMT 构建线程, 该线程定期或定量触发批量构建任务. 构建过程中, 系统首先从队列取出所有待处理的 DELETE 事件, 然后按路径前缀排序以提高缓存局部性, 最后并行计算更新后的树根哈希并持久化到 RocksDB. 构建完成后, 构建线程通知批处理调度器, 将最新的根哈希嵌入当前待提交批次. 如果调度器决策提前提交, 则会强制触发当前周期的 SMT 构建, 通过事件驱动机制实现模块间协调.

对于超过区块链单笔交易资源限制的大事务, 系统在本地维护协调状态表, 记录每个大事务的已提交子 ACU 数量、各子 ACU 哈希和链上协调合约地址. 当识别到大事务时, 系统首先在链上创建协调记录并获取合约地址, 然后根据资源估算将事务拆分为多个子 ACU. 拆分策略基于操作类型和数据量, 确保每个子 ACU 的预估资源低于限制的 80%. 拆分后的子 ACU 通过线程池并发提交, 线程池大小默认为 5. 每个子 ACU 提交成功后, 提交线程立即更新协调状态表. 链上协调合约采用计数器机制追踪子 ACU 到达情况, 当计数器达到总分片数时将事务标记为完成, 超时未收齐则自动回滚. 对于失败的大事务, 系统根据失败原因 (Gas 不足、nonce 冲突、合约验证失败) 执行不同的补偿策略: 资源不足时重试, nonce 冲突调整 nonce 重试, 验证失败记录到补偿日志并触发人工审核.

Gas 监控模块通过独立监控线程轮询区块链节点的 Mempool 接口, 获取当前 pending 交易列表和 Gas 价格. 系统解析返回数据, 统计 Gas 价格分布并计算分位数, 存储在 24 h 滑动窗口中. 批处理调度器每 10 s 评估批次队列状态, 从 Gas 监控模块读取最新价格数据后调用决策函数计算是否提交批次. 决策函数输入包括队列中待处理 ACU 数量、当前 Gas 价格、价格历史统计和最老 ACU 等待时间, 输出提交决策和批次大小. 如果决策为提交, 调度器从优先级队列取出相应数量的 ACU 打包成批次, 调用区块链交互器提交到链上. 交互器负责编码智能合约调用、签名交易、估算 Gas 并提交, 提交成功后启动异步监听线程轮询交易状态直至达到安全确认深度. 系统关键参数通过配置文件设置, 支持运行时热更新, 管理员通过信号触发配置重载无需重启服务.

混合逻辑时钟管理器集成在事务协调引擎主线程中, 维护物理时间戳和逻辑计数器两个变量. 每当 CDC 事件到达时, 管理器立即为事件分配全局时间戳. 对于本地生成的事件, 管理器将当前物理时间与本地时间戳比较取较大值, 如果时间戳增加则计数器重置为 0, 否则计数器递增. 对于从 Kafka 消费的远程事件, 管理器同时比较本地时间、远程时间戳和当前物理时间按规则更新. 为避免时间戳相同的事务产生歧义, 系统使用事务 ID 的字典序作为最终排序依据. HLC 时间戳与事件元数据一起存入事务缓冲区, 后续所有模块基于这个时间戳维护事务全局顺序. 检查点中记录当前的 HLC 值, 故障恢复时从检查点加载该值并继续递增.

6.3 并发控制与性能优化

TDSM 采用流水线式并发处理模型, 各模块通过异步队列通信提高吞吐量. CDC 消费线程池从 Kafka 拉取事件并推送到事件队列, 事务协调线程从事件队列消费并推送到 ACU 队列, 批处理调度线程从 ACU 队列消费并推送到提交队列, 区块链交互线程从提交队列消费并执行链上提交. 这种流水线设计使得各阶段可以并发进行, 当区块链交互线程等待链上确认时, 事务协调线程仍在继续聚合新事务.

为进一步提升性能, 本文实现了以下优化. 首先, 采用零拷贝序列化技术, Kafka 消息体通过内存映射直接访问, 避免了用户空间与内核空间的数据拷贝. 其次, 检查点写入采用批量插入优化, 系统将多个检查点记录打包成一个数据库事务提交, 降低了磁盘 I/O 次数. 实验表明, 批量插入将检查点写入延迟从平均 50 ms 降低到 10 ms. 第三, 维护与区块链节点的长连接池, 系统预先建立并维护多个 HTTP/WebSocket 长连接, 避免了每次提交时的 TCP 握手开销. 连接池大小默认为节点数的 2 倍, 当连接失败时自动剔除并重连. 对于 SMT 构建, 本文采用多线程并行优化, 将顶层 256 个子树分配给不同工作线程并行计算, 每个线程独立计算其负责子树的哈希值, 最后由主线程汇总计算根哈希.

7 实验分析

为验证 TDSM 架构及其核心机制的有效性, 本文构建了 TDSM 的原型系统并进行了全面的实验评估.

7.1 实验设置

• 测试平台

考虑到实验资源限制, 本文采用虚拟化技术在单台物理机上模拟分布式环境, 硬件配置为 Intel Core i7-10700 处理器 (8 核 16 线程)、32 GB 内存、2 TB SSD 固态硬盘. 部署 3 个虚拟节点, 节点 1: TDSM 核心组件+ Kafka Broker (分配 8 GB 内存, 4 核), Kafka 配置为 3 副本, 分区数调整为 6; 节点 2: MySQL + Debezium (分配 8 GB 内存, 2 核); 节点 3: Ganache 以太坊节点 (分配 8 GB 内存, 2 核). 同时通过虚拟网络引入可控的网络延迟 (平均 2 ms), 以更真实地评估系统在分布式场景下的性能特征.

区块链环境采用 Ganache 7.7 本地以太坊测试网络, Gas limit 设置为 8 000 000, 支持动态 Gas 价格模拟. TDSM 的核心贡献在于成本优化与数据生命周期管理, 实验设计的重点是验证自适应批处理的 Gas 成本节省效果、撤销证明机制的有效性以及各组件的增量贡献, 而非追求绝对吞吐量. 基于这一目标, 本文选择 Ganache 以太坊测试环境, 主要基于以下考虑: (1) Ganache 完整实现了以太坊的 EVM 和 Gas 计费机制, 智能合约的 Gas 消耗计算与主网一致, 这是准确评估成本优化效果的前提; (2) Gas 价格可灵活配置, 本文通过 Ornstein-Uhlenbeck 模型模拟真实的 Gas 价格波动特征 ($R^2 \approx 0.977$), 这种精细的价格控制在公链或联盟链测试网上难以实现; (3) 环境完全可控, 适合消融实验在严格控制变量的条件下逐一评估各组件贡献. 此外, TDSM 的批处理算法基于统一的跨链资源抽象模型设计, 其核心决策逻辑不依赖于特定区块链类型, 对于联盟链场景可直接切换至吞吐量优化模式.

• 工作负载

性能测试使用 YCSB 负载[33]. YCSB 是雅虎公司开源的数据库基准测试工具, 提供了可调节参数以进行全面评估. 本文使用 YCSB 的核心工作负载, 数据库包含 100 万条记录, 每条记录大小约 1 KB. 每个事务执行 10 次读/写操作, 事务访问服从 Zipf 分布. 默认的倾斜率为 0.3, 写读比为 0.3 (即 30% 读写事务和 70% 只读事务). 对于只读事务, TDSM 不进行区块链同步; 对于读写事务, CDC 捕获变更并通过 TDSM 同步到区块链.

为模拟真实区块链网络的 Gas 价格波动特征, 本文采用 Ornstein-Uhlenbeck (OU) 均值回归过程叠加泊松跳跃的随机过程模型[34]. Gas 价格的演化方程为:

$$G_{t+1} = X_t + J_t \tag{1}$$

其中, X_t 为 OU 基础过程:

$$X_{t+1} = X_t + \theta(\mu - X_t)\Delta t + \sigma\sqrt{\Delta t}\cdot\varepsilon_t,\ \varepsilon_t \sim N(0,1) \tag{2}$$

J_t 为泊松跳跃过程:

$$J_t = \sum_{i=1}^{N_t} Y_i,\ N_t \sim Poisson(\lambda\Delta t),\ Y_i \sim LogNormal\left(\mu_j, \sigma_j^2\right) \tag{3}$$

模型参数 θ (回归速度)、μ (目标水平)、σ (波动率) 和 λ (跳跃频率) 根据以太坊历史数据校准. 其中, ETH 为以太坊的原生代币 (以太币), Gwei 为以太坊中 Gas 价格的常用计量单位, 1 Gwei = 10^{-9} ETH. 具体实现中, 将 30 min 周期分为 4 个阶段: 低谷期 ($\mu = 25$ Gwei, $\theta = 1.0$)、上升期 ($\mu = 70$ Gwei, $\theta = 0.8$)、拥堵期 ($\mu = 130$ Gwei, $\theta = 0.5, \lambda = 0.15$) 和恢复期 ($\mu = 60$ Gwei, $\theta = 1.5$), 各阶段参数基于文献报告的统计特征设置. 该模型能捕捉 Gas 价格的强自相关性 ($R^2 \approx 0.977$)、突发峰值 (约 6.4% 极端事件) 和非对称波动等真实特征.

• 其他配置

除了上述工作负载中的调试参数外, 本文额外引入了 3 个配置指标: 批处理策略、成本权重 α 和延迟权重 β. 批处理策略包括固定批次大小 (Fixed-Size-10、Fixed-Size-50、Fixed-Size-100)、固定时间窗口 (Fixed-Time-100 ms、Fixed-Time-250 ms、Fixed-Time-500 ms) 以及 TDSM 的自适应批处理算法. 对于自适应算法, 默认配置 $\alpha = 0.6$, $\beta = 0.4$, 批次大小阈值 $n_{\max} = 200$, $n_{\text{mid}} = 100$, $n_{\text{small}} = 20$. SMT 撤销证明的构建周期默认设置为 DELETE 事

件累积到 1 000 条时触发.

本文每个实验重复进行 5 次, 去掉最高值和最低值, 报告剩余 3 个结果的平均值. 本文每次实验开始前都有一个 60 s 的预热阶段, 接着是 300 s 的数据收集阶段.

7.2 TDSM 优化效果验证

本节通过消融实验验证 TDSM 各优化机制的增量贡献. 实验在 YCSB 负载下进行, 写读比为 0.3, 包含 30% DELETE 操作以测试 SMT 撤销证明的影响. Gas 价格采用动态波动模型 (如实验设置所述).

图 6 展示了消融实验的结果. 基线 (Baseline) 表示采用最简单的批处理策略: 固定批次大小 50, 无 SMT 撤销证明 (DELETE 操作简单标记), 无自适应优化, 仅基础故障恢复. “+O1”表示在基线基础上增加 SMT 撤销证明机制, “+O1+O2”表示在“+O1”的基础上增加自适应批处理算法, “+O1+O2+O3”表示完整的 TDSM 系统.

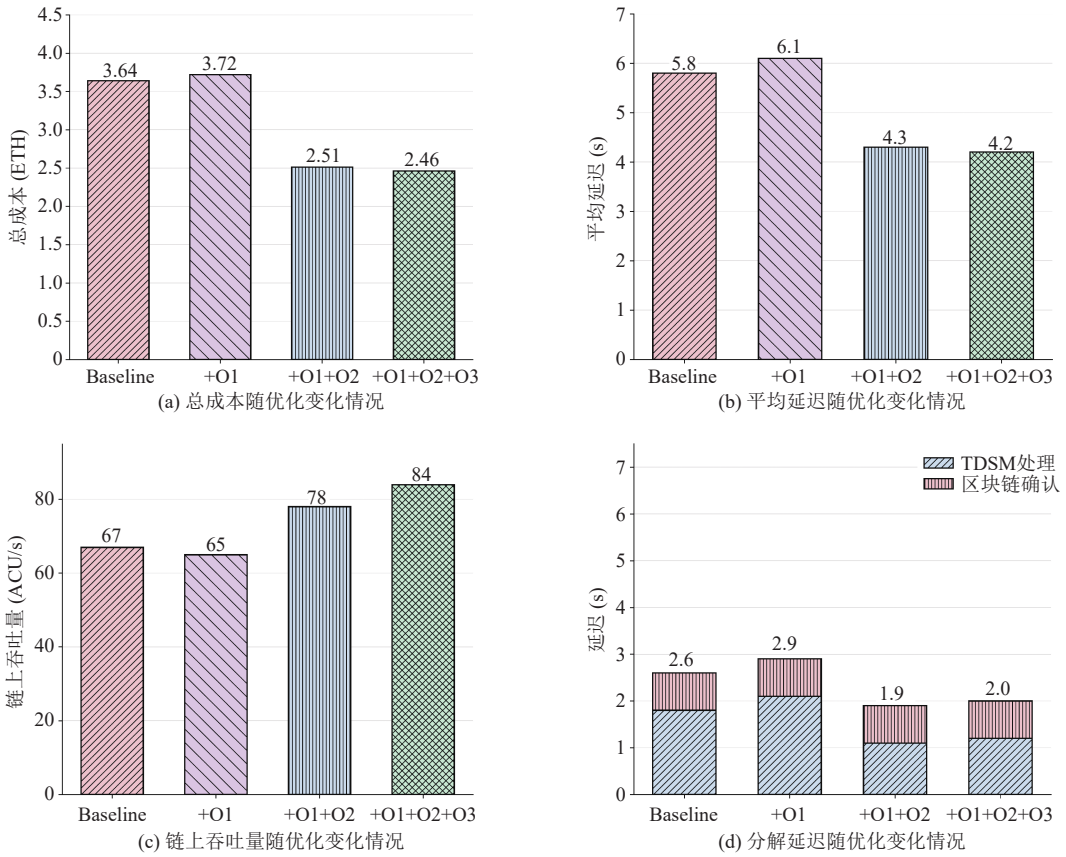


图 6 TDSM 优化效果测试

从图 6(a) 中可以看出, TDSM 的总 Gas 成本随着优化机制的增多而显著下降. 基线系统的成本为 3.64 ETH; 增加 SMT 撤销证明后 (+O1), 成本略微增加至 3.72 ETH, 增加了 2.2%, 主要来自根哈希的链上存储开销; 增加自适应批处理后 (+O1+O2), 成本大幅下降至 2.51 ETH, 降低了 30.1%, 证明了自适应机制是成本优化的核心; 完整系统 (+O1+O2+O3) 的成本为 2.46 ETH, 相比基线降低了 32.4%.

图 6(b) 展示了各配置平均延迟. 基线系统延迟为 5.8 s; 增加 SMT 后 (+O1), 延迟轻微增加至 6.1 s, 因为 SMT 构建需要额外计算; 增加自适应批处理后 (+O1+O2), 延迟下降至 4.3 s; 完整系统 (+O1+O2+O3) 延迟为 4.2 s.

对于自适应批处理改善延迟的原因, 需要澄清 3 个关键点: 第一, 对于单个事务而言, 相比直接提交, 任何批处理策略都会增加延迟; 第二, 基线采用 Fixed-Size-50 策略存在僵化等待问题. 即使网络空闲, 也必须等待凑够 50

个事务才提交, 同时在 Gas 价格高峰期仍按固定批次提交, 缺乏灵活性; 第三, 自适应机制通过实时感知网络状态, 在 Gas 价格适中且队列积压时可以更快提交, 避免不必要的等待, 在低成本窗口虽然会提交大批次 (增加本批次延迟), 但整体上减少了总提交次数. 因此, 自适应批处理的延迟改善主要来自避免僵化等待和动态权衡策略, 而非消除批处理的固有延迟成本.

图 6(c) 展示了链上提交吞吐量的变化. 这里的吞吐量定义为单位时间内成功提交到区块链的 ACU 数量. 基线系统的吞吐量为 67 ACU/s, 完整系统提升至 84 ACU/s, 提升了 25.4%. 这一改善主要来自两方面: (1) 自适应批处理减少了僵化等待时间, 使得 ACU 能够更快地被打包提交; (2) 在低成本窗口提交大批次, 单次提交包含更多 ACU, 从而提高了单位时间的 ACU 吞吐量. 需要注意的是, 此处的吞吐量提升反映的是中间件层批处理策略优化带来的效率改善, 关于系统整体吞吐量水平及其与底层链的关系, 将在第 7.2 节末结合整体吞吐量分析进一步讨论.

图 6(d) 展示了延迟分解分析. 基线系统中 TDSM 处理阶段的延迟为 1.8 s, 增加 SMT 后增加至 2.1 s (SMT 构建开销), 增加自适应批处理后下降至 1.0 s, 这主要是因为调度器的决策效率提升和避免了不必要的等待. 区块链确认延迟在各配置下保持稳定 (约 0.7 s), 这是区块链网络的固有特性.

值得注意的是, 尽管 SMT 撤销证明增加了约 0.3 s 的处理延迟和轻微的成本 (需要将 SMT 根哈希写入链上), 但它为 DELETE 操作提供了密码学级别的可审计性, 这是系统完整性的关键保障. 在不考虑 DELETE 审计需求的场景下, 可以禁用 SMT 以进一步降低延迟.

综合以上消融实验的结果, 可以从两个层面分析 TDSM 的性能特征. 在中间件层面, 轻量级原子提交协议与自适应批处理的协同优化带来了显著的效率提升——完整系统相比基线配置, Gas 成本降低 32.4%, 延迟降低 27.6%, 吞吐量提升 25.4%. 图 6(d) 的延迟分解进一步表明, TDSM 处理阶段的延迟从基线的 1.8 s 降至 1.0 s, 而区块链确认延迟在各配置下保持稳定 (约 0.7 s), 说明上述改善均来自中间件层的协议设计与调度策略优化, 原子提交协议本身的处理开销低且不构成系统瓶颈. 在系统整体吞吐量层面, TDSM 约为 100 TPS (84 ACU/s, 每个 ACU 包含多个操作), 这一数值主要受限于底层以太坊 EVM 的执行能力——以太坊的区块 Gas 上限和 EVM 串行执行模型对智能合约写入吞吐量施加了硬性约束. 以以太坊为目标链的既有系统面临同样的限制: BlockchainDB 的基准测试[7]表明, 其基于以太坊 PoA 在写入场景下仅约 30 TPS, 且 99% 的处理时间消耗在共识层; SEBDB[13]的吞吐量同样在几十 TPS 量级. TDSM 的吞吐量处于上述系统的正常区间, 在一定程度上说明中间件层并未引入显著的额外开销, 底层链是决定整体吞吐量的主要因素. 需要说明的是, 上述系统的实验环境和测量条件不完全相同, 此处引用仅用于说明以太坊 EVM 对吞吐量的共性约束. 当前实验未纳入与其他中间件系统的直接对比, 主要原因在于近年来大多数数据库-区块链中间件已转向 Fabric、Tendermint 等共识链, 底层共识机制的巨大差异使得直接对比吞吐量数字难以公平地反映中间件本身的效率. 在更高吞吐量的区块链平台上部署 TDSM 是有价值的后续工作方向.

7.3 原子性保障验证

本节验证 TDSM 的轻量级原子提交协议在各种场景下的可靠性. 实验设计了 4 个测试场景, 每个场景运行 1 000 个包含多操作的复杂事务, 每个事务包含 5–10 个 INSERT、UPDATE、DELETE 操作的组合.

表 2 展示了原子性保障验证的结果. 在正常提交场景下, 所有 1 000 个事务均成功提交到区块链, 提交成功率为 100%. 通过人工检查, 链上数据与数据库数据完全一致, 验证了协议在正常情况下的正确性.

表 2　原子性保障验证结果

测试场景	事务总数	提交成功率 (%)	故障恢复时间 (s)	平均重试次数	数据一致性
正常提交	1 000	100	N/A	0	√
中间件故障	1 000	100	8.3	0	√
区块链失败	1 000	100	N/A	2.1	√
并发事务	4 000	100	N/A	0	√

注: N/A 表示该测试场景不涉及故障恢复

在中间件故障场景下, 实验在随机时刻终止 TDSM 进程 (模拟崩溃), 然后立即重启. 由于 TDSM 维护了分层检查点并能从 Kafka 重新消费事件, 所有未完成的事务在重启后被正确恢复并提交. 故障恢复平均耗时 8.3 s, 期间无数据丢失. 提交成功率仍为 100%, 证明了故障恢复机制的可靠性.

在区块链失败场景下, 实验通过将 Gas limit 设置为低于批次需求模拟了 Gas 耗尽和通过注入延迟模拟了网络超时. TDSM 检测到提交失败后, 根据失败原因自动增加 Gas price 或延迟重试. 经过平均 2.1 次重试后, 所有事务最终成功提交. 这一结果表明, 协议能够正确处理区块链层面的临时故障. 在并发事务场景下, 实验同时从 4 个数据源并发提交事务来模拟多数据库同步场景. 尽管事务来自不同数据源, HLC 时间戳机制确保了全局因果顺序的一致性. 来自 4 个数据源的所有 4000 个事务均按正确的顺序提交到链上, 无冲突或乱序现象. 这验证了全局时序协调器的正确性.

此外, 本文还测试了大事务分片协调机制. 实验构造了 100 个超过 Gas limit 的大事务, 每个事务包含 100–200 个操作. TDSM 自动将其拆分为 2–4 个子 ACU 并通过链上协调合约保证原子性. 所有大事务均成功提交, 且拆分和协调的平均开销为 1.2 s, 开销可接受.

7.4 自适应批处理机制评估

本节评估 TDSM 的自适应批处理机制相比静态策略的优势. 核心评估目标是验证自适应算法能否在成本与延迟之间找到更优的权衡点. 实验对比了自适应算法与多种固定策略在成本、延迟和吞吐量方面的表现.

(1) 批处理策略对比

图 7 展示了不同批处理策略在 30 min 运行周期内的性能表现. 实验对比了 TDSM 的自适应算法 (TDSM-Adaptive) 与 6 种固定策略: 3 种固定批次大小策略 (Fixed-Size-10、Fixed-Size-50、Fixed-Size-100) 和 3 种固定时间窗口策略 (Fixed-Time-100 ms、Fixed-Time-250 ms、Fixed-Time-500 ms). 其中 X 轴为平均延迟, Y 轴为总 Gas 成本, 气泡大小表示 P99 延迟, 灰度深浅表示链上吞吐量, 颜色越深表示吞吐量越高. 从图 7 中可以清晰地观察到不同策略在成本-延迟权衡空间中的分布特征.

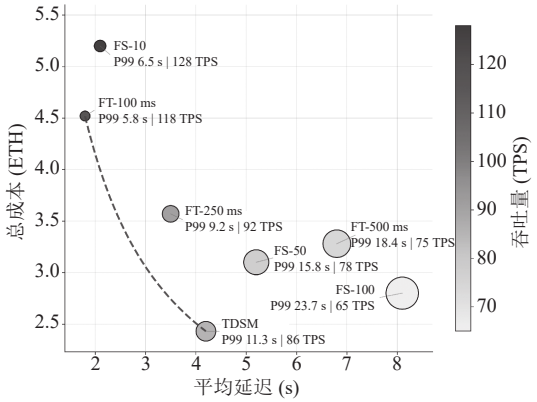


图 7 批处理策略对比

从成本维度分析, TDSM-Adaptive 的总 Gas 成本为 2.43 ETH, 显著低于所有固定策略. Fixed-Size-10 的成本最高 (5.20 ETH), 因为小批次导致链上交易数量过多; Fixed-Time-100 ms 的成本次之 (4.52 ETH), 虽然响应迅速, 但高频提交策略导致成本激增; Fixed-Time-500 ms 的成本为 3.28 ETH, 表现相对较好, 但仍比自适应算法高出 35%. 这表明 TDSM 通过感知 Gas 价格波动并动态调整批次大小, 有效降低了同步成本.

从延迟维度分析, Fixed-Time-100 ms 的平均延迟最低 (1.8 s), 但其总 Gas 成本达到 4.52 ETH, 是自适应算法的 1.86 倍, 性价比较差. TDSM-Adaptive 的平均延迟为 4.2 s, 处于中等水平, 但总成本最低. 这说明自适应算法在成本-延迟的权衡空间中找到了更优的平衡点: 相比 Fixed-Time-100 ms, 牺牲 2.4 s 延迟换来了 46.2% 的成本节省; 相比 Fixed-Size-100, 在降低延迟的同时也降低了 13.2% 的成本.

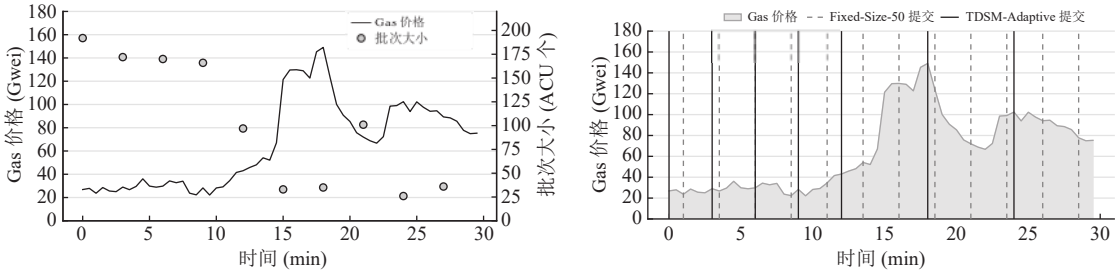
从 P99 延迟分布来看, TDSM-Adaptive 的 P99 延迟为 11.3 s, 满足了大多数业务场景的时效性要求. 相比之下, Fixed-Size-100 和 Fixed-Time-500 ms 的 P99 延迟分别达到 23.7 s 和 18.4 s, 这是追求低成本的代价; Fixed-Time-100 ms 的 P99 延迟较低 (5.8 s), 但综合成本过高.

从吞吐量角度分析, TDSM-Adaptive 的吞吐量为 86 TPS, 与 Fixed-Time-250 ms (92 TPS) 接近, 但成本降低了 31.9%. Fixed-Size-10 的吞吐量最高 (128 TPS), 但这是因为其提交了过多的小批次交易, 导致成本飙升. 这说明关键在于如何在吞吐量、成本和延迟之间取得平衡.

综合 4 个维度的分析, 可以发现 TDSM-Adaptive 在四维空间中占据了接近左下角理想区域的位置, 即在保持合理延迟和吞吐量的同时, 实现了最低的成本开销. 相比之下, 固定窗口策略沿着成本-延迟的权衡曲线分布, 无法同时优化多个目标; 固定批次策略如 Fixed-Size-100 和 Fixed-Size-50 位于帕累托前沿的右上方, 在成本和延迟上都被其他策略支配, 是明显的次优选择. 这证明了自适应批处理算法通过实时感知资源状态并动态调整策略, 能够在多目标优化问题中达到更优解.

(2) 资源波动适应性分析

为进一步分析 TDSM 自适应算法的决策过程, 图 8 展示了 Gas 价格波动与批次大小调整的时序关系.



(a) Gas价格和批次大小变化情况 (b) 提交时机对比

图 8 资源波动性适应分析

图 8(a) 绘制了 30 min 内 Gas 价格的变化曲线以及 TDSM-Adaptive 决策的批次大小. 可以观察到, TDSM 在 Gas 价格低谷期倾向于提交较大批次 (批次大小接近 200), 充分利用低成本窗口; 在 Gas 价格峰值期倾向于延迟提交或提交较小批次 (批次大小约 20–50), 避免高成本. 这一动态调整策略是 TDSM 节省成本的关键.

图 8(b) 对比了 TDSM-Adaptive 与 Fixed-Size-50 在相同 Gas 价格曲线下的提交时机. 虚线表示 Fixed-Size-50 的提交时刻, 实线表示 TDSM-Adaptive 的提交时刻. 可以看出, Fixed-Size-50 在 Gas 价格高峰期仍按固定频率提交, 导致不必要的高成本; 而 TDSM-Adaptive 在这些时刻选择延迟提交, 等待 Gas 价格下降后再提交, 成功规避了高成本区间.

8 总结和展望

本文针对混合区块链-数据库存储架构中间件模式的技术难点, 提出并实现了可信数据同步中间件 TDSM. 该中间件通过非侵入式架构, 系统性地解决了事务模型不匹配、性能与成本不匹配以及数据生命周期不匹配这 3 大核心技术难点. 本文的主要贡献包括: (1) 提出了完整的 TDSM4 层架构与同步方案, 实现了数据库与区块链的可靠、经济、完整同步; (2) 设计了集成可验证撤销证明的轻量级原子提交协议, 通过稀疏默克尔树为 DELETE 操作生成密码学证明, 保障了跨系统事务的原子性映射和数据全生命周期的链上可信审计; (3) 实现了成本感知的自适应批处理算法, 通过实时监控链上状态动态调整批处理策略, 在同步延迟与经济成本之间取得动态平衡; (4) 构建了原型系统并进行了全面的实验评估, 验证了 TDSM 各核心机制的有效性和中间件层的高处理效率. 实验结果表明, TDSM 在成本优化、原子性保障和自适应调度等多个维度均表现良好, 为数据库与区块链的融合提供了一种实用、经济的同步方案.

未来我们有 3 方面的工作: 一是可以扩展为支持跨链同步, 引入跨链协议实现原子跨链提交和一致性协调. 其

次, 可以探索将零知识证明技术集成到 TDSM 中, 在不暴露原始数据的情况下提供可验证的数据完整性证明, 或设计数据分级同步策略. 第三, 当前的自适应批处理算法基于实时资源成本状态进行决策, 未来可以引入更先进的时间序列预测模型或强化学习方法, 提供更准确的短期 Gas 价格预测.

References

[1] Pennekamp J, Matzutt R, Klinkmüller C, Bader L, Serror M, Wagner E, Malik S, Spiß M, Rahn J, Gürpinar T, Vlad E, Leemans SJJ, Kanhere SS, Stich V, Wehrle K. An interdisciplinary survey on information flows in supply chains. ACM Computing Surveys, 2024, 56(2): 32. [doi: 10.1145/3606693]

[2] Shao QF, Jin CQ, Zhang Z, Qian WN, Zhou AY. Blockchain: Architecture and research progress. Chinese Journal of Computers, 2018, 41(5): 969–988 (in Chinese with English abstract). [doi: 10.11897/SP.J.1016.2018.00969]

[3] Tsai WT, Yu L, Wang R, Liu N, Deng EY. Blockchain application development techniques. Ruan Jian Xue Bao/Journal of Software, 2017, 28(6): 1474–1487 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5232.htm [doi: 10.13328/j.cnki.jos.005232]

[4] Dinh TTA, Liu R, Zhang MH, Chen G, Ooi BC, Wang J. Untangling blockchain: A data processing view of blockchain systems. IEEE Trans. on Knowledge and Data Engineering, 2018, 30(7): 1366–1385. [doi: 10.1109/TKDE.2017.2781227]

[5] Zhang ZW, Wang GR, Xu JL, Du XY. Survey on data management in blockchain systems. Ruan Jian Xue Bao/Journal of Software, 2020, 31(9): 2903–2925 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6091.htm [doi: 10.13328/j.cnki.jos.006091]

[6] Yang XY, Zhang Y, Wang S, Yu BQ, Li FF, Li YZ, Yan WY. LedgerDB: A centralized ledger database for universal audit and verification. Proc. of the VLDB Endowment, 2020, 13(12): 3138–3151. [doi: 10.14778/3415478.3415540]

[7] Ge ZR, Loghin D, Ooi BC, Ruan PC, Wang TW. Hybrid blockchain database systems: Design and performance. Proc. of the VLDB Endowment, 2022, 15(5): 1092–1104. [doi: 10.14778/3510397.3510406]

[8] Ruan PC, Dinh TTA, Loghin D, Zhang MH, Chen G, Lin Q, Ooi BC. Blockchains vs. distributed databases: Dichotomy and fusion. In: Proc. of the 2021 Int'l Conf. on Management of Data. ACM, 2021. 1504–1517. [doi: 10.1145/3448016.3452789]

[9] Wang LP, Guan Z, Li QS, Chen Z, Hu MS. Survey on blockchain-based security services. Ruan Jian Xue Bao/Journal of Software, 2023, 34(1): 1–32 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6402.htm [doi: 10.13328/j.cnki.jos.006402]

[10] Nathan S, Govindarajan C, Saraf A, Sethi M, Jayachandran P. Blockchain meets database: Design and implementation of a blockchain relational database. Proc. of the VLDB Endowment, 2019, 12(11): 1539–1552. [doi: 10.14778/3342263.3342632]

[11] Androulaki E, Barger A, Bortnikov V, *et al*. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In: Proc. of the 13th EuroSys Conf. Porto: ACM, 2018. 30. [doi: 10.1145/3190508.3190538]

[12] Muzammal M, Qu Q, Nasrulin B. Renovating blockchain with distributed databases: An open source system. Future Generation Computer Systems, 2019, 90: 105–117. [doi: 10.1016/j.future.2018.07.042]

[13] Zhu YC, Zhang Z, Jin CQ, Zhou AY, Yan Y. SEBDB: Semantics empowered BlockChain database. In: Proc. of the 35th IEEE Int'l Conf. on Data Engineering (ICDE). Macao: IEEE, 2019. 1820–1831. [doi: 10.1109/ICDE.2019.00198]

[14] Kokoris-Kogias E, Jovanovic P, Gasser L, Gailly N, Syta E, Ford B. OmniLedger: A secure, scale-out, decentralized ledger via sharding. In: Proc. of the 2018 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2018. 583–598. [doi: 10.1109/SP.2018.000-5]

[15] Peng YQ, Du M, Li FF, Cheng R, Song D. FalconDB: Blockchain-based collaborative database. In: Proc. of the 2020 ACM SIGMOD Int'l Conf. on Management of Data. Portland: ACM, 2020. 637–652. [doi: 10.1145/3318464.3380594]

[16] Wang N, Wang B, Liu TY, Li W, Yang SH. A middleware approach to synchronize transaction data to blockchain. In: Proc. of the 29th Int'l Conf. on Computer Communications and Networks (ICCCN). Honolulu: IEEE, 2020. 1–8. [doi: 10.1109/ICCCN49398.2020.9209722]

[17] Budiu M, Chajed T, McSherry F, Ryzhyk L, Tannen V. DBSP: Incremental computation on streams and its applications to databases. ACM SIGMOD Record, 2024, 53(1): 87–95. [doi: 10.1145/3665252.3665271]

[18] Gray J, Lamport L. Consensus on transaction commit. ACM Trans. on Database Systems, 2006, 31(1): 133–160. [doi: 10.1145/1132863.1132867]

[19] Zakhary V, Agrawal D, El Abbadi A. Atomic commitment across blockchains. Proc. of the VLDB Endowment, 2020, 13(9): 1319–1331. [doi: 10.14778/3397230.3397231]

[20] Dong ZY, Wang ZG, Zhang XD, Xu X, Zhao CG, Chen HB, Panda A, Li JY. Fine-grained re-execution for efficient batched commit of distributed transactions. Proc. of the VLDB Endowment, 2023, 16(8): 1930–1943. [doi: 10.14778/3594512.3594523]

[21] Zhou JY, Xu M, Shraer A, *et al*. FoundationDB: A distributed unbundled transactional key value store. In: Proc. of the 2021 Int'l Conf. on Management of Data. ACM, 2021. 2653–2666. [doi: 10.1145/3448016.3457559]

[22] Venkataraman S, Panda A, Ousterhout K, Armbrust M, Ghodsi A, Franklin MJ, Recht B, Stoica I. Drizzle: Fast and adaptable stream processing at scale. In: Proc. of the 26th Symp. on Operating Systems Principles. Shanghai: ACM, 2017. 374–389. [doi: 10.1145/3132747.

3132750]
[23] Ali A, Pinciroli R, Yan F, Smirni E. Optimizing inference serving on serverless platforms. Proc. of the VLDB Endowment, 2022, 15(10): 2071–2084. [doi: 10.14778/3547305.3547313]
[24] Mohan C, Haderle D, Lindsay B, Pirahesh H, Schwarz P. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. on Database Systems (TODS), 1992, 17(1): 94–162. [doi: 10.1145/128765.128770]
[25] Helland P. Life beyond distributed transactions: An apostate's opinion. Queue, 2016, 14(5): 69–98. [doi: 10.1145/3012426.3025012]
[26] Richardson C. Microservices patterns: With Examples in Java. Shelter Island: Manning Publications, 2018.
[27] Merkle RC. A certified digital signature. In: Proc. of the 1989 Conf. on Advances in Cryptology. New York: Springer, 1989. 218–238. [doi: 10.1007/0-387-34805-0_21]
[28] Dahlberg R, Pulls T, Peeters R. Efficient sparse Merkle trees: Caching strategies and secure (non-)membership proofs. In: Proc. of the 21st Nordic Conf. on Secure IT Systems. Oulu: Springer, 2016. 199–215. [doi: 10.1007/978-3-319-47560-8_13]
[29] DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. Dynamo: Amazon's highly available key-value store. ACM SIGOPS Operating Systems Review, 2007, 41(6): 205–220. [doi: 10.1145/1323293.1294281]
[30] Carbone P, Ewen S, Fóra G, Haridi S, Richter S, Tzoumas K. State management in Apache Flink®: Consistent stateful distributed stream processing. Proc. of the VLDB Endowment, 2017, 10(12): 1718–1729. [doi: 10.14778/3137765.3137777]
[31] Das S, Botev C, Surlaker K, Ghosh B, Varadarajan B, Nagaraj S, Zhang D, Gao L, Westerman J, Ganti P, Shkolnik B, Topiwala S, Pachev A, Somasundaram N, Subramaniam S. All aboard the Databus! LinkedIn's scalable consistent change data capture platform. In: Proc. of the 3rd ACM Symp. on Cloud Computing. San Jose: ACM, 2012. 18. [doi: 10.1145/2391229.2391247]
[32] Kulkarni SS, Demirbas M, Madappa D, Avva B, Leone M. Logical physical clocks. In: Proc. of the 18th Int'l Conf. on Principles of Distributed Systems. Cortina d'Ampezzo: Springer, 2014. 17–32. [doi: 10.1007/978-3-319-14472-6_2]
[33] Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. In: Proc. of the 1st ACM Symp. on Cloud Computing. Indianapolis: ACM, 2010. 143–154. [doi: 10.1145/1807128.1807152]
[34] Meister BK, Price HCW. Gas fees on the Ethereum blockchain: From foundations to derivative valuations. Frontiers in Blockchain, 2024, 7: 1462666. [doi: 10.3389/fbloc.2024.1462666]

附中文参考文献

[2] 邵奇峰, 金澈清, 张召, 钱卫宁, 周傲英. 区块链技术: 架构及进展. 计算机学报, 2018, 41(5): 969–988. [doi: 10.11897/SP.J.1016.2018.00969]
[3] 蔡维德, 郁莲, 王荣, 刘娜, 邓恩艳. 基于区块链的应用系统开发方法研究. 软件学报, 2017, 28(6): 1474–1487. http://www.jos.org.cn/1000-9825/5232.htm [doi: 10.13328/j.cnki.jos.005232]
[5] 张志威, 王国仁, 徐建良, 杜小勇. 区块链的数据管理技术综述. 软件学报, 2020, 31(9): 2903–2925. http://www.jos.org.cn/1000-9825/6091.htm [doi: 10.13328/j.cnki.jos.006091]
[9] 王利朋, 关志, 李青山, 陈钟, 胡明生. 区块链数据安全服务综述. 软件学报, 2023, 34(1): 1–32. http://www.jos.org.cn/1000-9825/6402.htm [doi: 10.13328/j.cnki.jos.006402]

作者简介

卿昱潇, 硕士生, 主要研究领域为区块链技术, 区块链泛金融.
高建彬, 博士, 副教授, CCF 高级会员, 主要研究领域为区块链技术, 大数据安全.
吴敏, 高级工程师, 主要研究领域为"数字政府"相关领域信息化总体规划, "数字福建"项目建设.
夏琦, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为网络数据安全, 区块链理论及应用.
肖杨磊, 工程师, 主要研究领域为大数据分析, 网络与数据安全, 政务信息安全, 区块链应用, 人工智能安全.
夏虎, 博士, 副研究员, CCF 专业会员, 主要研究领域为区块链理论与应用, 大数据安全.
张劲松, 硕士, 主要研究领域为数据安全, 隐私保护, 区块链应用, 人工智能安全.
罗锦涛, 博士生, CCF 学生会员, 主要研究领域为数据安全, 可信计算.