LedgerDB: A Centralized Ledger Database for Universal Audit and Verification

Xinying Yang[†], Yuan Zhang[†], Sheng Wang[§], Benquan Yu[†], Feifei Li[§], Yize Li[†], Wenyuan Yan[†]

[†]Ant Financial Services Group [§]Alibaba Group

{xinying.yang,yuenzhang.zy,sh.wang,benquan.ybq,lifeifei,yize.lyz,daniel.ywy}@alibaba-inc.com

ABSTRACT

The emergence of Blockchain has attracted widespread attention. However, we observe that in practice, many applications on permissioned blockchains do not benefit from the decentralized architecture. When decentralized architecture is used but not required, system performance is often restricted, resulting in low throughput, high latency, and significant storage overhead. Hence, we propose LedgerDB on Alibaba Cloud, which is a centralized ledger database with tamper-evidence and non-repudiation features similar to blockchain, and provides strong auditability. LedgerDB has much higher throughput compared to blockchains. It offers stronger auditability by adopting a TSA two-way peg protocol, which prevents malicious behaviors from both users and service providers. LedgerDB supports verifiable data removals demanded by many real-world applications, which are able to remove obsolete records for storage saving and hide records for regulatory purpose, without compromising its verifiability. Experimental evaluation shows that LedgerDB's throughput is 80× higher than state-of-the-art permissioned blockchain (i.e., Hyperledger Fabric). Many blockchain customers (e.g., IP protection and supply chain) on Alibaba Cloud have switched to LedgerDB for its high throughput, low latency, strong auditability, and ease of use.

PVLDB Reference Format:
Xinying Yang, Yuan Zhang, Sheng Wang, Benquan Yu, Feifei Li, Yize Li, and Wenyuan Yan. LedgerDB: A Centralized Ledger Database for Universal Audit and Verification. *PVLDB*, 13(12): 3138-3151, 2020.
DOI: https://doi.org/10.14778/3415478.3415540

1. INTRODUCTION

Since the launch of Bitcoin [32], blockchain has been introduced to many applications as a new data collaboration paradigm. It enables applications to be operated by mutually distrusting participants. A blockchain system implements hash-entangled data blocks to provide a tamper-evident ledger with bulks of transactions committed by a certain consensus protocol (e.g., PoW [32], PBFT [14], HoneyBadgerBFT [28]). Decentralization is a fundamental basis for blockchain systems, including both permissionless (e.g., Bitcoin, Ethereum [21]) and permissioned (e.g., Hyperledger Fabric [6], Corda [11], Quorum [31]) systems.

A permissionless blockchain usually offers its cryptocurrency to incentivize participants, which benefits from the decentralized ecosystem. However, in permissioned blockchains, it has not been shown that the decentralized architecture is indispensable, although they have been adopted in many scenarios (such as IP protection, supply chain, and merchandise provenance). Interestingly, many applications deploy all their blockchain nodes on a BaaS (Blockchain-as-a-Service) environment maintained by a single service provider (e.g., Microsoft Azure [8] and Alibaba Cloud [4]). This instead leads to a centralized infrastructure. As there is still no widespread of truly decentralized deployments in permissioned blockchain use cases, many advocates are veering back to re-examine the necessity of its decentralization. Gartner predicts that at least 20% of projects envisioned to run on permissioned blockchains will instead run on centralized auditable ledgers by 2021 [23].

To reform decentralization in permissioned blockchains, we propose LedgerDB – a ledger database that provides tamper-evidence and non-repudiation features in a centralized manner, which realizes both strong auditability and high performance. We call it a *centralized ledger database* (CLD) system, which pertains to *centralized ledger technology* (CLT) that is opposed to *decentralized ledger technology* (DLT) in blockchain. LedgerDB is developed and widely adopted at Ant Financial from Alibaba Group. Its major designs are inspired by limitations in conventional permissioned blockchains [6, 26, 11, 31, 18, 29, 30] and ledger databases [7, 40].

First, we observe that many use cases of permissioned blockchains (e.g., provenance and notarization) only seek for the tamper resistance from cryptography-protected structures, such as hashed chains and Merkle trees [27]. Oftentimes, none of the other features (e.g., decentralization and smart contract) are utilized. In this case, a simplified auditable ledger service from a credible central authority with essential data manipulation support is sufficient. It tends to be more appealing against blockchains, due to its high throughput, low latency, and ease of use.

Second, the threat model of permissioned blockchain is inadequate for real-world audit. For external (i.e., third-party) auditors beyond its consortium, data replication guarded by consensus protocols seems deceptive. For example,

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415540

Table 1: Key comparisons between LedgerDB and other systems.

System	Throughput (max TPS)	Auditability				Removal		Non-Repudiation		Provenance
		external	third party	peg	capability	purge	occult	server-side	client-side	native clue
LedgerDB	100K+	✓	TSA	✓	strong	✓	✓	✓	✓	✓
QLDB [7]	1K+	✗	✗	✗	weak	✗	✗	✗	✗	✗
Hyperledger [6]	1K+	✗	✗	✗	weak	✗	✗	✓	✓	✗
ProvenDB [40]	10K+	✗	Bitcoin	✓	medium	✗	✓	✗	✗	✗
Factom [43]	10+	✓	Bitcoin	✓	strong	✗	✗	✓	✓	✗

with the endorsement from all peers, they are able to fabricate fake evidence (e.g., modify a transaction from ledger) against external auditors without being noticed. What is worse, the partial endorsement from a majority is already sufficient to conduct above deception, if others fail to provide original chain proofs to external auditor. As a consequence, permissioned blockchains only retain limited auditability.

Third, though some centralized auditable ledger databases emerge as alternatives to permissoned blockchains, they however excessively assume a trustful service provider. For example, AWS discloses QLDB [7, 13] that leverages an immutable and verifiable ledger; Oracle announces its blockchain table [35]. These auditable ledgers require that the ledger service provider (LSP) is fully trusted, such that historical records can be archived and verified at the server side. It remains unknown whether the returned records are tampered by LSP, especially when it colludes with a certain user.

Fourth, the immutability[1] provided by existing systems implies that each piece of committed data is permanently disclosed on the ledger as the tamper evidence, which indeed causes prohibitive storage overhead and regulatory issues in real-world applications. Many blockchain users wish to clean up obsolete records for storage savings, since it will otherwise become unaffordable as data volume grows. In terms of regulatory issues, those records violating law or morality (e.g., upload someone's residential address without prior approval) cannot be erased once it has been committed, which is undesirable in many applications.

To address these issues, LedgerDB adopts a centralized architecture and a stateless journal storage model, rather than tightly-coupled transaction state model in most blockchain systems [21, 30], to achieve high system throughput. It adopts multi-granularity verification and non-repudiation protocols, which prevents malicious behaviors from (colluded) users and LSP, eliminating the trust assumption to LSP. By integrating with digital certification and timestamp authority (TSA) service [10, 37], LedgerDB service on Alibaba Cloud [4] is capable of offering judicial level auditability. It has already served many real-world customers for their blockchain-like applications, such as supply chain, IP protection, RegTech (regulatory technology), and enterprise inner audit. We compare LedgerDB with other systems in Table 1 (detailed in § 2) and summarize our main contributions as follows:

- We investigate the gap between existing auditable systems (i.e., blockchains and ledger databases) and real-world requirements from the aspects of auditability, functionality and performance.
- We develop LedgerDB, which, to the best of our knowledge, is the first centralized ledger database that provides strong auditability, broad verification coverage, low storage cost, as well as high throughput.
- To ensure strong auditability, we adopt non-repudiable signings and a TSA peg protocol to prevent malicious behaviors (defined in § 2.2) from users and LSP.
- We provide verifiable data removal operators to resolve storage overhead and regulatory issues. They help remove obsolete records (*purge*) or hide record content (*occult*) without compromising verifiability. We also provide native primitives (*clue*) to facilitate the tracking of application-level data provenance.
- To sustain high throughput for write-intensive scenarios, we enhance LedgerDB with tailored components, such as an *execute-commit-index* transaction workflow, a batch accumulated Merkle-tree ($bAMT$), and a clue index (cSL). Our experimental result shows that LedgerDB achieves excellent throughput (i.e., 300,000 TPS), which is 80× and 30× higher than Hyperledger Fabric and QLDB respectively.

The rest of this paper is organized as follows. We discuss limitations in current solutions and our design goals in § 2. We overview LedgerDB in § 3, and then present our detailed designs (i.e., journal management, verification mechanism, and provenance support) in § 4, § 5 and § 6 respectively. Use cases and real-world practices are provided in § 7. We provide experimental evaluation in § 8 and related work in § 9, before we conclude in § 10.

2. PRELIMINARIES AND DESIGN GOALS

We work on the design of a *ledger database* that supports blockchain-like (especially permissioned blockchain) applications using a centralized architecture. It records immutable application transactions (from mutually distrusting participants) on the ledger with tamper-proofs, and provides universal audit and verification capabilities. In this section, we introduce our design goals that make LedgerDB a more powerful alternative to existing blockchain systems, from aspects of auditability, threat model, functionality and performance.

2.1 Auditability

Auditability is the capability of observing user actions and operation trails, which is an essential property for ledger databases (and blockchain systems). A successful auditing trail should guarantee the authenticity of all audited data. For example, if a transaction is being audited, the auditor should confirm that its content has not been tampered (i.e., integrity) and its issuer is authentic (i.e., non-repudiation). Here we further divide this concept into two scenarios: internal audit and external audit. *Internal audit* ensures that an internal user of the ledger can observe and verify the authenticity of all actions (we call them *transactions*) conducted by all other (distrusted) users. *External audit* ensures that an external third-party entity (e.g., auditor) can observe and verify the authenticity of all transactions conducted by all

[1]Here we mean that any piece of data, once committed into the system, cannot be modified by subsequent operations and becomes permanently available.

(distrusted) participants of that ledger, e.g., users and ledger service provider (LSP). In other words, a participant cannot maliciously tamper or repudiate own transactions when they are to be audited.

Most existing ledger and blockchain systems support internal audit (with certain assumptions or constraints), but few of them is able to provide external audit. We discuss the auditability of several typical systems as follows (which is also summarized in Table 1):

QLDB. It is a centralized ledger database developed at AWS, where users submit their transactions into a shared ledger. In this system, the LSP is fully trusted by all users, and it adopts server-side verification mechanisms to detect and alert user tampers. Users are not aware of whether the LSP tampers their data. In summary, QLDB provides internal audit under the context that the LSP must be trusted.

Hyperledger Fabric. It is a popular permissioned blockchain for mutually distrusted users to run collaborative applications among a consortium. A typical consortium in permissioned chain can forge timestamps and fork their ledger when majority members collude (e.g., more than 2/3 of peers in a PBFT-based consensus protocol). In summary, Hyperledger provides internal audit under the context that majority peers are honest.

ProvenDB. It is a centralized ledger database that leverages Bitcoin-based anchors to validate timestamps on the ledger. However, due to the inherent commitment latency in Bitcoin (e.g., 10 minutes), there is a vulnerable time window issue, i.e., recent transactions might be compromised during the time window they are to be committed. Besides, it suffers from intentional submission delay issues due to its one-way anchoring protocol. In summary, ProvenDB provides internal audit similar to QLDB and limited external audit (from large vulnerable time window).

Factom. It is a permissionless blockchain for decentralized data notarization, which forms the strongest auditable model among conventional ledgers. Its permissionless decentralization makes it a powerful source of credible timestamps where applications can rely on. Along with non-reputable transaction signings, Factom (as well as Bitcoin) supports both internal and external audits, but suffers poor system performance (from permissionless).

2.2 Threat Model

We formally provide the threat model considered in our design, which inherently implies the full support of both internal and external audits. For any user and the LSP, we only assume that their identities are trusted (i.e., by disclosing their public keys certified by a CA). In addition, a third-party timestamp authority (TSA) needs to be trusted, which is able to attach true and verifiable timestamps (of current time) to a given piece of data. We assume that cryptographic primitives (e.g., message authentication code and signature) guarantee tamper resistance and non-repudiation.

Except for trusted components above, any behavior from a participant can be malicious, i.e., they might deliberately try to tamper or repudiate existing transactions. In particular, we consider two typical attack scenarios: (a) server-side malicious tampering and (b) LSP-user collusion tampering, where three types of threats are involved (*threat-A*, *threat-B*, *threat-C*). For scenario (a), the adversary can be either an outside attacker, an internal user or the LSP, who compromises the entire database server. The adversary may access any transaction on the ledger and tamper incoming transactions when user requests arrive (*threat-A*). He/she may tamper (e.g., update, delete, insert) historical transactions on the ledger, so as to forge a fake ledger to cheat users (*threat-B*). For scenario (b), one or more client(s) can collude with the LSP as the adversary. The adversary may tamper or remove any historical transaction to cheat an external auditor (*threat-C*). ***We guarantee that all above threats can be detected by both internal participants and external auditors.***

2.3 Design Goals

Strong external auditability. Consider the threat model above, we observe that existing ledger databases and permissioned blockchains are vulnerable to some of mentioned threats and inadequate for real-world auditability. For example, QLDB is vulnerable to all three threats; ProvenDB is vulnerable to *threat-A* and has partially prevented *threat-B* and *threat-C*. The lack of external auditability makes them unsuitable for real-world applications. In contrast, LedgerDB works the assumption of honest majority and provides strong external auditability, by leveraging a third-party TSA. In short, it is impossible for an adversary to tamper earlier data once a TSA *journal* is anchored on ledger.

High write performance. The write performance in permissioned blockchains is significantly limited by their decentralized architecture. However, such kind of decentralization is not indispensable in many scenarios. For example, some applications deploy all blockchain nodes in a consortium on a BaaS (Blockchain-as-a-Service) environment maintained by a single service provider, which instead leads to a centralized infrastructure eventually. In this case, a centralized ledger database (CLD) is more appealing, due to its high throughput, low latency, and ease of use. Moreover, we observe that most customers concern more about write performance compared to read and verification performance. Hence, LedgerDB is a system mainly optimized for high write performance, and at the same time sustains acceptable read and verification performance.

Data removal support. Data immutability is inherent in existing blockchain systems, which infers that all committed transactions must be permanently disclosed on the ledger as tamper evidences in order to support verification and audit. However, this causes prohibitive storage overhead and regulatory issues in real-world applications. Note that the mission of audit does not enforce the prerequisite of immutability in real-world applications. Therefore, we aim to allow the removal of obsolete or violating records, without compromising the capabilities for verification and audit. Hence, LedgerDB breaks immutability, but instead provides two operations: *purge* removes obsolete records to save storage cost and *occult* hides violating records to resolve regulatory issues (detailed in § 5.4). These operations enhance the system with more flexible auditability that facilitates both customers and regulators.

Table 1 summarizes major differences between LedgerDB and existing systems from all aspects discussed above.

3. LEDGERDB OVERVIEW

LedgerDB is a CLD system that conforms to the threat model and design goals discussed above, i.e., strong external auditability, high write performance, and data removal

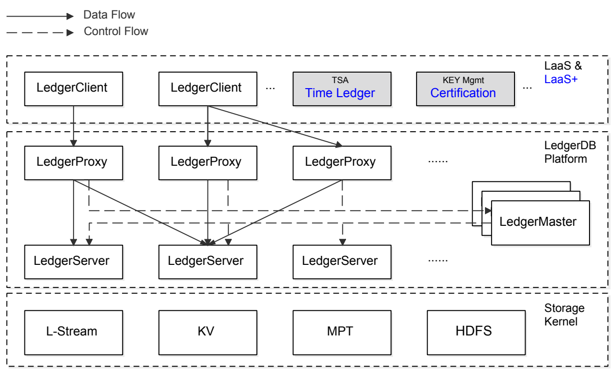


Figure 1: LedgerDB system architecture.

support. In this section, we introduce LedgerDB's overall architecture, as well as its data model and APIs. LedgerDB can be either used as a standalone software or deployed as a LaaS (Ledger-as-a-Service) on cloud, i.e., a CLD service. It has been deployed on Alibaba Cloud, where digital certification and TSA services are integrated to provide judicial-level auditability. We call this deployment LaaS+.

3.1 Architecture

The architecture of LedgerDB is shown in Figure 1, which contains three major components, i.e., ledger master, ledger proxy, and ledger server. A *ledger master* manages the runtime metadata of the entire cluster (e.g., status of servers and ledgers) and coordinates cluster-level events (e.g., load balance, failure recovery). A *ledger proxy* receives client requests and preprocesses, and then dispatches them to the corresponding *ledger server*. A *ledger server* completes the final processing of requests, and interacts with the underlying storage layer that stores ledger data. Different storage engines are supported, such as file systems (e.g., HDFS, *L-Stream*), KV stores (e.g., RocksDB), and MPT (Merkle Patricia Tree) [25]. In particular, *L-Stream* (LedgerDB stream) is a linear-structured append-only file system tailored for LedgerDB. It eliminates costly compaction (e.g., RocksDB) and can be fast accessed via storage offsets. Note that ledger proxy can communicate with the storage layer directly when handling transaction payloads (§ 4.2).

LedgerDB provides high service availability and data reliability. It leverages end-to-end CRC (cyclic redundancy check) to ensure data integrity during transmission and retrieval. The storage layer adopts a shared storage abstraction, where ledger data is replicated in multiple available zones and regions following a raft protocol. Therefore, ledger servers and ledger proxies are completely stateless, which makes their failover and restart easy.

3.2 Data Model and APIs

In LedgerDB, a ledger manipulation operation (initiated by either a user or the LSP) is modeled as a *transaction*, which results in a *journal entry* (*journal* for short) appended to the ledger. Multiple *journals* are further grouped as a *block* on the ledger (analogous to the block in blockchain). When a user executes an operation, he/she first initializes the transaction at client and signs the content with his/her own secret key. Then, this signed message is wrapped as a request payload and sent to ledger proxy. Table 2 lists main operators provided by LedgerDB. In these APIs, *ledger_uri* is the unique identifier of each ledger instance; *journal* sequence number (*jsn*) is the unique identifier of each *journal* assigned by the ledger server using a monotonic counter. We explain some categories of operator as follows:

Table 2: Main operators and APIs in LedgerDB.

Operator	Method
Create	`Create(ledger_uri, enum, op_metadata)`

Append	`AppendTx(ledger_uri, tx_data, clue, set)`
	`SetTrustedAnchor(ledger_uri, jsn, level)`
	`GrantRole(ledger_uri, member_id, role)`
	`GrantTime(ledger_uri, timestamp, proof)`

Retrieve	`GetTx(ledger_uri, jsn)`
	`ListTx(ledger_uri, ini_jsn, limit, clue)`
	`GetTrustedAnchor(ledger_uri, jsn, level)`
	`GetLastGrantTime(ledger_uri, timestamp)`

Verify	`Verify(ledger_uri, jsn | clue, data, level)`
Purge	`Purge(ledger_uri, block)`
Occult	`Occult(ledger_uri, jsn | clue)`
Recall	`Recall(ledger_uri, purged_point)`
Delete	`Delete(ledger_uri, enum, op_metadata)`

- **Append** – *append user transaction or system-generated transaction to ledger.* The user transaction can contain any type of digital information representing application semantics (such as transaction logs, audit trails, and operation traces). There are four major operators in this category: 1) `AppendTx` appends a user transaction, which can be associated with a *clue set* to express application-level data provenances (§ 6); 2) `SetTrustedAnchor` sets up a trusted anchor of at a certain trust *level* to facilitate subsequent verification requests (§ 5.3); 3) `GrantTime` records a TSA signed notary *journal* for universal time endorsement (§ 5.2); 4) `GrantRole` assigns roles or permissions to a member of the ledger. A role is a logical authority granted with predefined operating permissions to the ledger (e.g., normal, DBA, regulator, verify-only, read-only, etc).
- **Retrieve** – *get qualified journals from ledger.* We provide three retrieval types to fetch *journals*: 1) a *singleton* retrieval takes a *jsn* as input and returns the corresponding *journal* (via `GetTx`); 2) a *range* retrieval takes a *jsn* range as input and returns all *journals* in that range (via `ListTx` with *ini_jsn* and *limit* specified); 3) a *clue* retrieval takes a clue as input and returns all *journals* containing that clue (via `ListTx` with *clue* specified). In addition, `GetTrustedAnchor` retrieves a trusted anchor, while `GetLastGrantTime` retrieves a time notary (created by `GrantTime`).
- **Verify** - *verify integrity and authenticity of returned journals from journal proofs.* A proof is a non-repudiable evidence signed by an entity. In LedgerDB, user signs the payload of a transaction request; LSP signs the receipt of a transaction commitment; and TSA signs its generated time notary. LedgerDB supports two types (*levels*) of verification (§ 5), i.e., server-side and client-side, suitable for different scenarios. Verification can be accelerated using trusted anchors (§ 5.3).
- **Create** – *create a new ledger with initial roles and members.* The creation of a new ledger involves the initialization of many components, such as a unique ledger identifier, a genesis configuration, system-level *journals*, and third-party registrations (e.g., TSA genesis timestamp). A list of initial members shall be specified during creation, each of which holds at least one role. Role and member configurations are appended as *journals* in genesis block.

- **Purge** – *remove obsolete journals from ledger.* A *purge* is a bulk removal of contiguous *journal*s starting from genesis block (for storage savings). A pseudo genesis will be generated after the purge. We detail it in § 5.4.1.
- **Occult** – *hide journal(s) from ledger.* An *occult* can remove any *journal* (for regulatory purpose) without compromising the integrity of the entire ledger. Since occulted *journal*s cannot be accessed anymore, only regulators are allowed to execute it with care. It can be used in two ways: 1) *occult by sequence number* hides a *journal* given its *jsn*; 2) *occult by clue* hides all *journal*s containing a designated clue. We detail this operator in § 5.4.2.

There are also other operators, e.g.: `Recall` rollbacks the effect of a purge (within a limited time window); *Delete* removes entities in the system, such as a ledger, a role, a member, or a clue. Once a clue is deleted, *journal*s containing that clue will not be returned via clue retrievals, but they can still be reached via range retrievals.

4. JOURNAL MANAGEMENT

In this section, we introduce our *journal* structure and transaction processing, which ensure that committed transactions are tamper-resistant, non-repudiable and persistent.

4.1 Non-repudiable Journal

At client side, a *journal* is initially packed as a payload, which contains operator-related fields (e.g., *ledger_uri*, transaction type, clues) and client-related fields (e.g., client sequence, timestamp and nonce). We call this payload *journal-data* as shown in Figure 2(a). A *request-hash* digest is calculated based on the entire *journal-data* (e.g., using SHA256). The user then signs the request for non-repudiation purpose, which resolves *threat-A* (§ 2.2). Finally, the signed request is submitted to the server.

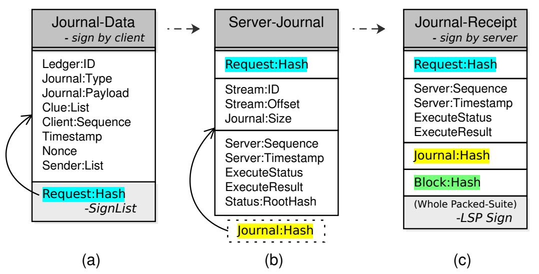


Figure 2: Non-repudiable journal structures.

At server side, a *journal* structure is called *server-journal* as illustrated in Figure 2(b). Apart from the client payload above, additional fields are attached to the *journal* by server, such as *jsn*, stream location (§ 4.2), server timestamp, execution receipt, and membership status. The server then calculates a *journal-hash*, which is the digest of the entire server-side *journal*. The *jsn* (i.e., server sequence) is a monotonically incremented integer assigned to each incoming *journal*.

After a *journal* is committed, the server returns a *journal-receipt* to the client, as shown in Figure 2(c). A *block-hash* refers to the block that contains this *journal*. Besides three digests (i.e., *request-hash*, *journal-hash*, *block-hash*), other fields like *jsn*, timestamp and status are also included in the final receipt. Server then signs the receipt for non-repudiation purpose, before sending it back to the client. The client can keep this receipt as a proof for subsequent verification, which resolves *threat-B*.

4.2 Transaction Processing

In LedgerDB, a *transaction* represents an execution of an operator (listed in Table 2), which may create a *journal* on the ledger. Most blockchain systems adopt an order-and-execute approach to execute transactions serially, leading to extremely low throughput. Hyperledger Fabric [6] proposes an execute-order-validate approach to support concurrent execution of multiple transactions, which however declines significantly when the conflict rate is high during validation. To resolve above issues, LedgerDB adopts a novel *execute-commit-index* transaction management approach. It combines execution and validation where transaction progress will be validated as early as possible before the completion of its entire execution, which better utilizes centralized concurrency control and distributed execution techniques to improve throughput.

Figure 3 shows the overall transaction workflow. In a nutshell, a transaction first enters the execute phase based on its transaction type (e.g., `append`, `retrieve`). It runs on ledger proxy for better scalability. Commit phase collects multiple executed transactions, arranges them in a global order, and persists them to the storage system. It runs on ledger server. Note that the success of a *transaction* only relies on whether it completes the commit phase. After that, index phase starts on ledger server to build indexes for subsequent data retrieval and verification. We explain each phase in detail as follows:

Execute phase. At this phase, the ledger proxy first verifies the validity of the submitted *transaction*, such as its authentication, data integrity, and timeliness. It consults the authorization module to check whether the sender has permission to perform this operation. After that, the verified (non-read) *transaction* is stored to a *Transaction Store* (at storage layer) to reduce communication and storage cost in later commit phase. After that, actual operation logic is performed based on its transaction type, which may involve the modification of the *world state* (which contains metadata of ledger and members). Since the world state is small in size, it is stored in a memory-based key-value store. A transaction might fail at any moment, if it cannot pass any round of the validation. After the execution completes (no matter succeeds or fails), it will be sent to the ledger server for commit phase.

Commit phase. At this phase, multiple *journal*s are collected from the previous phase and processed in batch. Each *journal* is assigned with a unique serial number *jsn* based on its server arrival sequence. The *journal* is then committed sequentially to a *JournalInfo Store* (at storage layer). This *JournalInfo Store* stores committed *journals*, including *Transaction Store* location of the *transaction* (e.g., stream ID and offset in *L-Stream*), return code of the execution, *jsn*, and the timestamp. To ensure data integrity, the hash of the structure is also calculated and stored (i.e., *journal-hash* in § 4.1). For a failed *transaction*, it is not stored in JournalInfo Store. However, we still return a receipt, where the *jsn* is assigned the same as the one from the last succeeded transaction to inform user when it is failed.

Index phase. At this phase, we build indexes for newly committed transactions to support fast data retrieval and verification. In particular, three types of indexes are built,

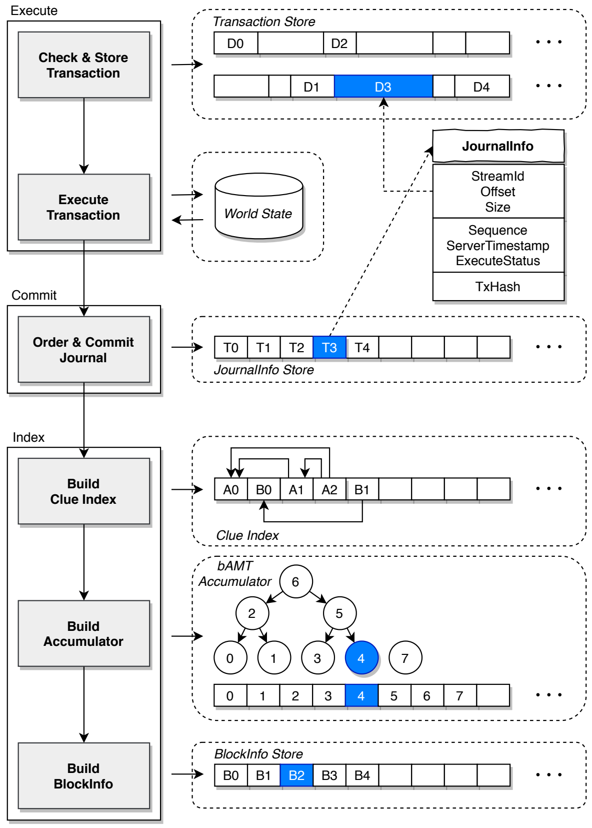


Figure 3: LedgerDB transaction workflow.

i.e., clue index (§ 6.2), *bAMT* accumulator (§ 5.1), and block information. The *clue index* links all historical *journals* through their clues. The clue index serves all singleton, range and clue retrievals (§ 3.2). The *bAMT accumulator* contains all *journals* and provides proofs of journal existence. The *block information* is different from the block in traditional blockchain. It is a summary of the entire ledger, rather than the digest of a single block. With this summary, all ledger information created at a certain time can be globally verified, such as historical *journals*, *clues*, and states. After index phase completes, the *journal* receipt is then signed and replied to client.

4.3 Failure Recovery

LedgerDB handles a variety of failure scenarios, such as ledger server and ledger proxy crashes. In general, a failure recovery requires to handle three components: *JournalInfo Store*, world state, and indexes. Ledger server needs to recover all these components, while ledger proxy only recovers the world state. The recovery is based on the list of *journals* committed in the *JournalInfo store*. For *world state*, it loads the latest checkpoint into memory and then replays subsequent *journals*. Since indexes are stored on *L-Stream*, they only need to read the last indexed *journal* and replay subsequent *journals*.

5. VERIFICATION AND AUDIT

Data verification is essential in ledger databases and blockchains, where it is used to validate data integrity and operation proofs (based on predefined protocols). In LedgerDB, verification can be conducted by different roles in two ways:

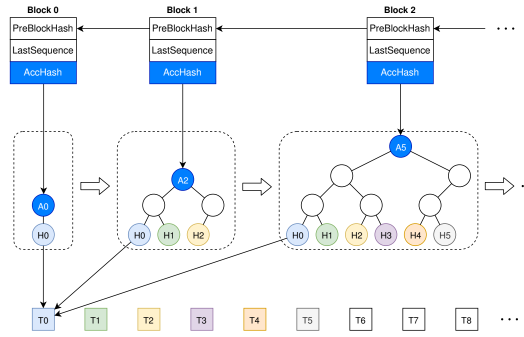


Figure 4: Accumulated Merkle root across blocks.

1) users who demand high efficiency can conduct verification at *server side*, where LSP must be fully trusted. 2) any users and external auditors who have the access to the ledger can conduct verification at *client side*, where LSP is distrusted and external auditability is guaranteed. In this section, we present our major mechanisms designed for various verifications: how to verify the existence of transactions (§ 5.1); how to support universal time notary that can be verified (§ 5.2); how to accelerate verification with trusted anchors (§ 5.3); how to support data removal (i.e., *purge* and *occult*) that can also be verified (§ 5.4).

5.1 Batch Accumulated Merkle Tree

To verify the existence of specified transactions, there are two typical entanglement models used in blockchain-like systems. One is *block-intensive* model (*bim*), where a block is associated with the hash of its previous block to ensure all previous blocks are immutable [17, 43, 32, 12]. Transactions within a same block are kept in a Merkle tree [27] to reduce verification cost to $O(\log n)$ (where n is total transactions in the block). The other is *transaction-intensive* model (*tim*) adopted by QLDB [7, 13] and Libra [16, 5]. In Libra, each transaction is directly associated with its parent transaction in a Merkle tree, which limits the insertion and verification throughput as the tree keeps growing. To benefit from both models above, we propose an advanced batch-accumulated Merkle tree (*bAMT*) that optimized for high transaction-write throughput.

Algorithm 1: Batch calculation of Merkle tree

```
   Data: Merkle-tree $T$, hash set of new journals $H$
   Result: new Merkle-tree roothash $r$
1  $S \leftarrow \varnothing$
2  for all $h$ in $H$ do
3      $n \leftarrow$ Create a new leaf node of $T$ whose hash is $h$
4      Push node $n$ to $S$
5  end
6  while $S \neq \varnothing$ do
7      Pop node $n$ from $S$
8      $p \leftarrow$ $n$'s parent node
9      if node $p$ is valid and never been calculated then
10         Calculate the hash of node $p$
11         Push node $p$ to $S$
12     end
13 end
14 return roothash($T$)
```

The structure of *bAMT* is shown in Figure 4. In *bAMT*, a block contains the hash of its previous block (like in *bim*), as well as a transaction-batch accumulated root hash of a Merkle tree that contains all historical transactions (like in *tim*). This model makes *journal* insertion fast without decreasing verification efficiency. More specifically, for

Merkle trees in *tim*, the computation cost for an insertion is $O(d)$, where d is the tree height. For a batch of m transaction insertions in *tim*, the cost of its root calculation is $O(m \times d)$. As transactions accumulate, the tree becomes deeper, which severely limits overall throughput. In contrast, *bAMT* adopts a coarse-grained batch accumulation approach, instead of a per-transaction accumulation approach, to improve the write throughput. Algorithm 1 shows the procedure of *bAMT* construction. The input takes a batch of transaction hashes. It adds these hashes to the leaf nodes of the original Merkle tree and put the new leaf nodes into a set S (lines 1-5). Then, it takes each node out of S (line 7), calculates the hash of its parent, and adds it back into S (lines 9-12). This process continues iteratively until we reach the root node (lines 6, 13).

In terms of time complexity, a batch of m transactions, the node hash calculation relates to the number of involved non-leaf nodes. The first non-leaf layer has approximately $\frac{1}{2}m$ nodes, whose upper layer contains approximately $\frac{1}{4}m$ nodes, until halving to the top layer. Hence, it requires $\sum_{n=1}^{d} m(\frac{1}{2})^n \approx m$ hash calculation. Along with the root hash calculation of the d-height tree path, the total computation cost of *bAMT* is $O(m + d)$, which is much smaller than that in *tim*. Compared with *bim*, since the batch size is comparatively smaller than the number of transactions in *bim*'s block, the insertion latency is also lower.

5.2 Universal Time Notary Anchors

Recall that the auditability in existing blockchains and ledger databases are very limited, especially for external audit (§ 2.2). In order to resolve *threat-C*, we introduce a special type of *journal*, called TSA *journal*, endorsed by TSA [10]. TSA is a time notary authority, which can prove that a piece of data exists before a certain time point. A TSA *journal* contains a ledger snapshot (i.e., a ledger digest) and a timestamp, signed by TSA in entirety. These *journals* are mutually entangled between ledger and TSA as time notaries, which provide external auditability for timestamps. Figure 5 depicts this two-way peg TSA entanglement, where a ledger digest is first signed by TSA and then recorded back on ledger as a TSA *journal*. The very first TSA anchor is stored in the genesis block, while latter notary anchors are placed one after another by referencing the hash of its previous TSA *journal*. Compared with one-way peg protocol in ProvenDB (§ 2.1), this two-way protocol can reduce the vulnerable time window to a marginal constant (but not completely eliminated).

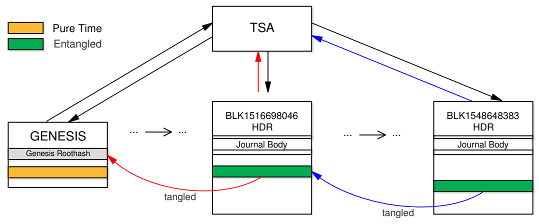


Figure 5: TSA time entangling journals.

Note that a TSA *journal* is generated after the commit of multiple consecutive blocks (we call it *anchor frequency*), which renders a trade-off between cost and security. Frequent TSA anchors reduce the vulnerable time window but require more resources. To further reduce interactions with TSA, a *Time Ledger* (*T-Ledger*) can be maintained by the LSP as a shared time notary anchor service for all other ledgers. It accepts digests submitted from different ledgers and interacts with TSA for TSA *journals*, acting as an agent between ledgers and TSA. A ledger can request for a time anchor from *T-Ledger* by sending its ledger digest, and then append the receipt from *T-Ledger* as a TSA *journal* on its own ledger. Compared to TSA, *T-Ledger* can sustain much higher throughput, where ledgers can request for time anchors at millisecond frequency. This defenses *threat-C* in practice with negligible vulnerable time windows. We offer *T-Ledger* service on our LaaS+, and anyone can download the entire *T-Ledger*, leading to strong external auditability.

5.3 Multi-layer Trusted Anchors

In LedgerDB, a *trusted anchor* is a ledger position, at which its digest has been successfully verified and hence can be exempt from future verification. It is regarded as a credible marker that can accelerate a various number of operators (involving verification). When auditing to a *journal*, we only need to conduct incremental verification from the closest anchor (before the *journal*) to it. We provide two major types of trusted anchors for different usage: server-side anchor (ξ_s) is used when LSP is assumed fully trusted (e.g., when a user prefers efficiency over security); client-side anchor (ξ_c) is used when LSP cannot be trusted, (e.g., when an external auditor fulfill his duty). The choice of verification type and trusted anchor can be specified when calling `Verify` (with parameter `server` or `client` in § 3.2). Note that users can set anchors on their own demands via `SetTrustedAnchor`. In each ledger, its ξ_s is shared with all members, while each member can set her/his own ξ_cs.

5.4 Verifiable Data Removals

LedgerDB supports several data removal operators demanded by real-world applications. It breaks the immutability in most blockchain and ledger databases but still preserves data verifiability. In particular, *purge* removes obsolete records to save storage cost (§ 5.4.1) and *occult* hides violating records to resolve regulatory issues (§ 5.4.2).

5.4.1 *Purge*

A *purge* operation deletes a set of contiguous (obsolete) *journals* starting from genesis to a designated *journal* on ledger, and then generates a pseudo genesis block. The relevant status of the designated point (such as memberships, clue counts) will be recorded to the pseudo genesis block. In addition, original system metadata in genesis (such as creator, initial members, timestamp and block hash) are also migrated. A *purge* operation can be illustrated[2] as follows (where `pur_jsn` is the target purging point):

```
DELETE FROM ledger_uri
 WHERE jsn < pur_jsn;
```

After a purge, a pseudo genesis will be (logically) stored at the purging tail, replacing the last purged block. A purge request also appends a purge *journal* on ledger, which is double-linked with the pseudo genesis as illustrated in Figure 6. This two-way peg enhances their mutual proofs, and provides fast lookups. The *purge* operation also utilizes the server-side trusted anchor ξ_s. An incremental verification

[2]SQL statements are used to help readers to understand the logic more easily. They can be expressed in our APIs otherwise.

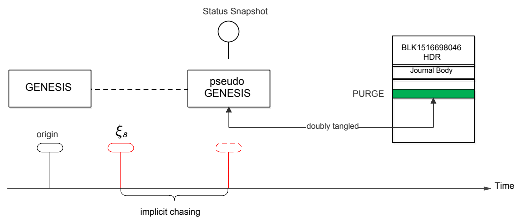


Figure 6: Purge operation with pseudo genesis.

between the closest ξ_s and the purging point will be executed before the purge, which is also sketched in Figure 6. Besides, we also allow the recall of unintentionally purged *journal*s within a recent time window. These *journal*s will be finally erased if no recall is issued.

As purge is a powerful operator, it is only accepted when multi-signatures from the ledger creator (or DBA) and all related members (i.e., whose historical *journal*s are involved in the purge) are gathered. This guarantees the agreement and non-repudiation from all members.

5.4.2 *Occult*

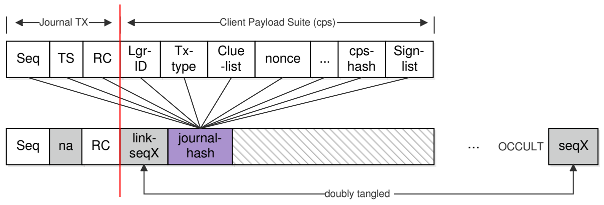


Figure 7: Occult relevant data structure.

We propose the *occult* operation to hide anti-regulatory *journal*s. An occult converts the original *journal* to a new one that only keeps its metadata (such as *jsn*, timestamp and return code). The occulted journal hash (`journal-hash`) and relevant *jsn* (`seqX`) overwrites original payload's starting offset, and the remaining payload body is formatted as shown in Figure 7. In the figure, the upper and lower portions represent *journal*s before and after occult respectively. The left portion represents the journal's metadata fields, while the right portion represents the *cps* (client payload suite) for payload data. We use a timestamp earlier than the ledger born time to represent *unavailable* (as an *occult indicator*). The occulted *journal* sets its occult flag and then overwrites with *jsn* of the occulting proposer, i.e., requester of `seqX`, at the original *cps* as well as the *cps* digest for proof and future verification. An *occult* operation can be illustrated as follows (where `Seq` is the target *jsn* to hide, and `seqX` relates to the occulting requester's *jsn*):

```
UPDATE ledger_uri
 SET TS = na, cps = CONCAT(
  seqX, journal_hash, blanks)
   WHERE jsn = Seq
      OR cid = des_cid;
```

When occulted *journals* are involved during verification, the protocol is refined as follows: when an occulted flag is detected (from the occulting indicator), we lookup its retained hash digest at *cps*; otherwise, we calculate *journal* hash as normal. To improve lookup efficiency, an occult bitmap index is used to filter out unnecessary checks on normal blocks.

Since the main purpose for occult operator is regulation, we only expect to hide violated or illegal *journal*s, instead of normal ones. Hence, it needs dual signatures from DBA and regulator role holders (e.g., an Internet court) to execute, regardless of its original transaction owner.

6. NATIVE PROVENANCE

Data provenance is important in many blockchain applications, where lineage-orientated retrievals are required to track relationships between different items or actions. To support high-performance lineage management, LedgerDB proposes a native primitive called *clue* that bridges business semantics and stateless ledger databases. It offers a friendly interface to capture lineages in application logic (i.e., clue-based insertion, retrieval and verification).

6.1 Clue Primitive

Existing provenance solutions either suffer from low system performance, i.e., in permissionless blockchains like Everledger [22] (smart contract) and Bitcoin (UTXO), or lack of external auditability, i.e., in permissioned blockchains and ledger databases. For example, UTXO in Bitcoin requires recursive searches that cannot be parallelized; the authenticity of lineage information in QLDB cannot be essentially verified. In LedgerDB, a *clue* is a user-specified label (key) that carries on business logic for data lineage. Together with our centralized architecture and TSA two-way peg protocol, clue realizes provenance management with both high performance and external auditability.

Taking the supply chain as an example, we can specify a clue for a certain merchandise by its unique *id*. All relevant phase traces (e.g., *journals* for packaging, transportation, logistic, stack) are all linked by a same clue defined by the application. Unlike in key-value models where each record has only one key, we can specify multiple clues in `AppendTx`, which allows a single *journal* to be correlated with different application logic. For example:

```
AppendTx(lgid,data,'g0',{'rock','blues','reggae'})
```

represents a certain music `g0` with a mixed style of rock, blues and reggae. A more sophisticated use case is provided in § 7.3.

6.2 Clue Index

Clue-based operators include clue insertion, retrieval and verification. As we target write-intensive scenarios, *clue write* has to cope with high throughput. We design a write-optimized index structure for clues as shown in Figure 8. For each clue, all related *journals* are organized in a reversed *clue Skip List* (*cSL*). A cell (or node) in *cSL* records its own *jsn* and points to its previous cell. All cells are appended to a linear-structured storage when they are created, in which we can locate them using storage offsets directly. Note that cells from a same *cSL* are unnecessary to be adjacent to each other. Hence, we use a fixed number of linear-structured storage instances for clue indexes, where multiple *cSL*s may share an instance. Since the cardinality of clue is usually small, we keep the mapping between *clueid* and corresponding *cSL* in memory. To support fast failover, *cSL* is check-pointed to disk periodically.

Clue Index Write. For a clue insertion, its input is a [*clueid, jsn*] pair. First, we locate corresponding *cSL* and create a cell to record *jsn* (as well as linkages to other cells when needed). The cell is then appended to the storage. This entire process is of $O(1)$ complexity. Given its small size

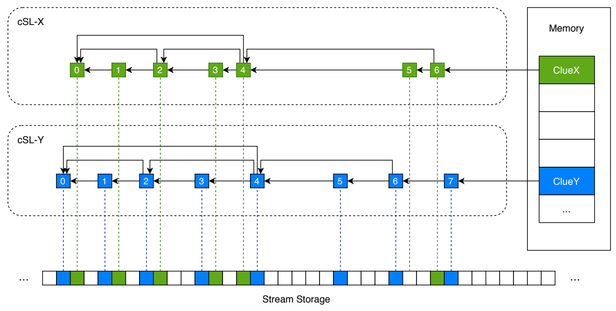


Figure 8: Structure of clue index.

(e.g., tens of bytes) and storage bandwidth (e.g., 100MBps), the write throughout can easily reach one million per second.

Clue Index Read. A clue point retrieval reads the latest *jsn* that shares a specified clue with a given *jsn*. Since all *jsn*s are linked in *cSL*, the lookup process is straightforward. For a clue with n *journal*s, both the computation and storage costs are $O(\log n)$. A clue range retrieval reads a set of *jsn*s for a specified clue with a given *jsn*. First, we perform a point retrieval is to locate the starting cell. For a range size m, we iterate until m cells are obtained. Its total complexity is $O((\log n) + m)$. A range retrieval can be further accelerated by reading subsequent $m - 1$ cells concurrently.

6.3 Clue-Oriented Verification

Clue verification is a discrete batch verification process on all related *journal*s, which is different from sequential linked-block transaction verification [32] and world-state verification [21]. We apply a dedicated verification protocol for this case. A *clue-counter MPT* (*ccMPT*) is proposed to trace the current count (i.e., value) of a certain clue (i.e., key) and anchor its root hash into the latest block header. Given a ledger $\mathcal{L}$ whose *bAMT* is Λ and a specified clue σ: the clue-count trusted anchor for σ is m (where m is 1 for most cases, and will be the latest count before last purge otherwise); $\mathcal{L}$'s *ccMPT* is Δ. A function S (search) takes as input the clue data σ, a tree identifier Δ, and outputs a *journal* amount n. A function V (validate) takes as input a clue σ, a tree roothash r_Δ or r_Λ, a tree identifier Δ or Λ, and outputs a boolean proof π. The clue verification process is defined as follows:

1. $n = \mathsf{S}(\sigma, \Delta)$, which retrieves *journal* amount for clue σ;
2. $\pi_c = \mathsf{V}(\sigma, r_\Delta, \Delta)$, where n is verified from Δ by its current roothash r_Δ from the latest block, and a proof of σ's count n is provided by π_c;
3. Get σ relevant *journal*s by clue index and ledger data $\looparrowright \mathcal{J}_\sigma = \{J_{m+1}, J_{m+2}, ..., J_n\}$
4. Validate each *journal* in $\mathcal{J}_\sigma$ to check if $J_{i\in(m,n]} \in \mathcal{L}$ by $\pi_i = \mathsf{V}(h_{i\in(m,n]}, r_\Lambda, \Lambda)$
5. $\pi = \pi_{m+1} \wedge \pi_{m+2} \wedge ... \wedge \pi_n$, where the whole clue verification proof is calculated by π, and σ is only proved when all $\pi_{i\in(m,n]}$ are proved.

Proof π is only true when the clue count n and all the corresponding n *journal*s are proved. Any unproved status (such as incorrect count, broken *journal*) during the verification process will lead to a false π.

7. LEDGERDB IN PRODUCTION

In real applications, CLD (e.g., LedgerDB) can be leveraged in three styles of user scenarios, and we refer them as mono ledger, *federated* ledger, and *delegated* ledger. *Mono* ledger is the simplest style, in which only one peer whose role is administrator exists. An example is the private memo ledger where everyone keeps his/her own private journals without intersection with others. *Federated and delegated* ledgers are more sophisticated, and we discuss them in § 7.1 and § 7.2 respectively. Some of our customer scenes like supply chain and RegTech (regulatory technology) solutions belong to federated ledger, while other scenes like federated learning and bank auditing suit for delegated ledger.

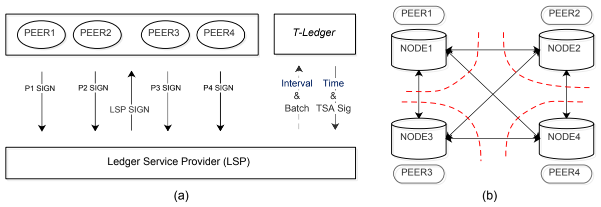


Figure 9: Permissioned chain and federated ledger.

7.1 Federated Ledger

Definition 1. Federated ledger has more than one participants, who have the demand of mutual audit.

Recall that LedgerDB offers external auditability (§ 2), which is not reliant on the trust to LSP. In contrast, permissioned chain is lack of external auditability. Their mutually distrusting participants account the ledger by a delegated consensus, which just offers internal auditability within its consortium. Figure 9 compares major differences between federated ledger (Figure 9(a)) and permissioned blockchain (Figure 9(b)) in terms of architecture. For many use cases of permissioned blockchain, like lineage and notarization, what they want is the stateless tamper-evidence with external auditability, where none of blockchain's decentralized architecture or distributed consensus really matters in these contexts. We believe that such scenarios better fit for a CLD system, on which each permissioned chain node is superseded by a CLD's federated peer, as shown in Figure 9(a).

Here we introduce a typical use case of federated ledger. A certain bank (*Bank-A*) uses LedgerDB for national *Grain-Cotton-Oil* (GCO) supply chain to provide tamper resistance and mutual non-repudiation. As one of the national policy banks with Central bank's bonds, *Bank-A* will never hold an equal blockchain node same as corporations under its regulation. Hence, permissioned blockchain doesn't work here. *Bank-A* holds a centralized GCO ledger on LedgerDB. All regulated corporations, such as oil manufacturers, cotton retailers, suppliers and grain warehouses, are all certified as GCO ledger members who can append their IOUs, manuscripts, invoice copies, receipts as *journal*s on ledger. To achieve external auditability, the batch of GCO ledger digests are submitted to *T-Ledger* publicly, and then TSA *journal*s with credible timestamp and signature are recorded back on GCO ledger. GCO ledger helps *Bank-A* manage all its regulated agricultural transactions in a credible, traceable and transparent manner, which greatly benefits its policy loan, risk control, and macroscopic regulation.

7.2 Delegated Ledger

Definition 2. Delegated ledger only interacts with a centralized agent, who further manages other participants with the demand of mutual audit.

Delegated ledger is more like an initiative meta-framework rather than a deployment style. Most of real customer cases have their existing centralized system for their businesses, and would like to further enhance their solution with rigorous auditability. Delegated ledger leverages the centralized system itself and TEE (Trusted Execution Environment) to offer a low-intrusive solution. In particular, we propose a separation of (four) powers, i.e., *journal* recording (client), *journal* keeping (LSP), universal timing (TSA) and delegated signing (TEE) on delegated ledger:

- **Journal recording.** Terminal public key holders (i.e., client or called peer) only communicate with the centralized system (CS). Peer's public key is previously mapped, deployed and stored in a TEE enclave by CS's open logic. The ledger is transparent to peers, who do not explicitly make a signing action.
- **Delegated signing.** After CS generates *journal* from peer and submits the delegated process to TEE, TEE then signs the *journal* transaction using the corresponding secret key as delegation, which is homologous to the public key of the peer. The delegated signature is credible because of the endorsement of hardware vendor's authority.
- **Universal timing.** To prevent the potential hazard that LSP forges local timestamps (and conspires with users), the two-way peg TSA *journal* is endorsed on ledger. This ensures a judicial level auditability for trusted universal timestamps.
- **Journal keeping.** Once a *journal* is flushed on ledger upon user submission, it will be immutable afterwards. LSP then responds its own signature together within the transaction receipt. This finalizes LSP's non-repudiation as discussed in § 4.1.

LedgerDB's use cases in Alibaba MPC (Secure Multi-Party Computation) represent a typical paradigm of delegated ledger. The multi-party mutual data exchange and SQL manipulation are all tracked and stored on a single ledger maintained by CS. It manages requests from all the league members and sends delegated signature tasks to TEE. In this case, both record tamper-proof and league's mutual non-repudiation are guaranteed.

7.3 Provenance in Action

Here we discuss how LedgerDB clue (native provenance) is used in real applications. An example use case is from the authorship and royalty copyrights ledger of National Copyrights Administration of China (NCAC) [34]. NCAC uses LedgerDB to record authorship and royalty copyrights information for all cyber literatures and national papery publications from 400+ publishing houses.

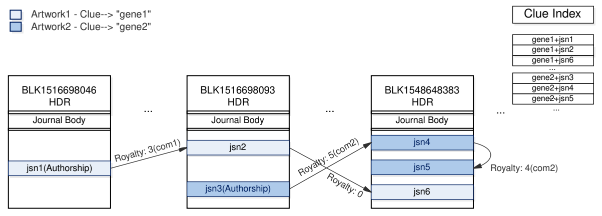


Figure 10: Copyrights ledger with clue.

Figure 10 illustrates the clue on NCAC copyrights ledger for two pieces of artwork. Artwork1's unique clue is `gene1`, and artwork2's unique clue is `gene2`. The first *journal* for a certain artwork is always an authorship right's claim. Here we can see that both artwork1 and artwork2's first relevant *journal*s are the authorship *journal*s. For artwork1, royalty have a reassignment at `blk1516698093` with 30% percentage royalty (arrows illustrate this trade detail) sold to `com1` (recorded in *jsn2*). The *jsn6* is an invalidation *journal*, from which the entire royalty returns to the author again. Artwork2 follows a similar logic. Via `ListTx`, taking *clueid* `gene1` as input, corresponding *journal*s (artwork1's *jsn1, jsn2, jsn6*) are then fetched as clue provenance details.

8. EVALUATION

We deploy LedgerDB on Alibaba Cloud and evaluate its performance under different workload configurations (e.g., payload size, job number, access pattern). In particular, we compare *bAMT* with Libra's transaction accumulator, and compare *cSL* with RocksDB for different clue operations. An application-level evaluation is provided that compares with a state-of-the-art permissioned blockchain, i.e., Hyperledger Fabric. All experiments were conducted in a 2-node cluster, where each node runs CentOS 7.2.1511 and is equipped with Intel(R) Xeon(R) Platinum 2.5GHz CPU, 32GB RAM, and 1TB ESSD storage (with throughput at 632MBps and 53K IOPS) from a public storage service on Alibaba Cloud. All nodes are connected via 25Gb Ethernet.

8.1 Transaction Throughput

We first evaluate the overall transaction throughput of LedgerDB. Four typical different access patterns are used, which are derived from common business scenarios: append-only writes, sequential reads, random reads, and random reads on recent data. In real applications, the size of *journal* payload is usually limited: small items (e.g., operation log, audit trail) or digests of large items (e.g., audio and video notarization). Hence, we evaluate with KB-level *journal* sizes. We vary the number of client jobs up to 128 to demonstrate the end-to-end throughput and the trends. Figure 11(a) shows the throughput for append-only writes. For 1KB-length *journal* writes, a single client can reach 6K tps. As the job number increases, the throughput rises almost linearly until the number reaches 64. After that, the server is saturated and the throughput converges to 280K tps. We can observe that the throughputs for writing 128B and 1KB *journal*s do not differ much. It is because the network and disk cost (which is proportional to the journal size) is not the bottleneck, while the CPU cost (which varies little for different journal sizes) dominates. However, for 4KB *journal*s, the throughput is dominated by the disk bandwidth and can only reach 153K tps as shown in Figure 11(a).

Figure 11(b) shows the throughput for sequential reads. Its throughput is higher than append-only writes (with the same payload) as shown in Figure 11(b). Due to smaller process overhead, read throughput (for 1KB *journal*s) can exceed 300K tps. For 4KB and 16KB *journal*s, the throughput is also limited by the disk bandwidth as in append-only writes, which can only reach 155K and 40K tps. We observe that the sequential read throughput is not significantly better compared to the write throughput in Figure 11(a), which is because writes in *L-Stream* are extremely lightweight that avoid data compaction.

Figure 11(c) shows the throughput for random reads. Because of low hit ratio in buffer pool, random reads dramatically increase I/O cost. The overall performance is limited by disk random-read capability, and can only reach 54K tps. Random reads on recent data is also a typical user workload.

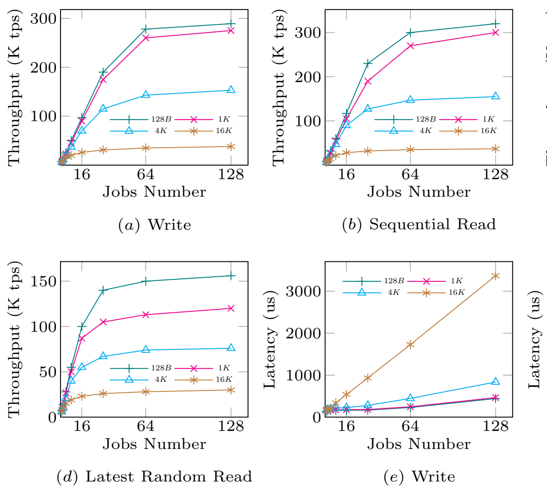


Figure 11: End-to-end throughput and average latency with different request sizes under different access patterns in LedgerDB.

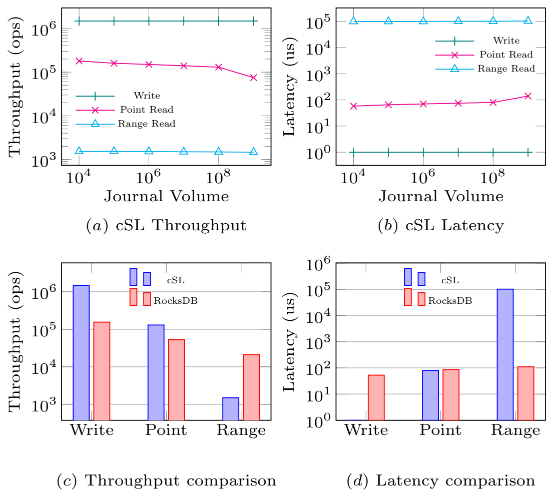


Figure 12: Comparison of performance between cSL and RocksDB clue index in different access modes (fixed journal volume of 10^8).

Due to the relatively small amount of accessed transactions, the hit ratio in buffer pool is much higher. Hence, its performance is better than random reads and can reach 100K tps as shown in Figure 11(d). The smaller the transaction data, the higher the hit ratio in buffer pool can be.

Figure 11(e) shows the overall end-to-end write latency. The latency of transactions is similar to the throughput, which is affected by the degree of concurrency and transaction data size. As can be seen, the latency is less than 1ms for any payload size under 4KB. Figure 11(f) measures the latency for random read, which is 2ms to 3ms for transaction size between 128B and 4KB.

8.2 Performance of Clue

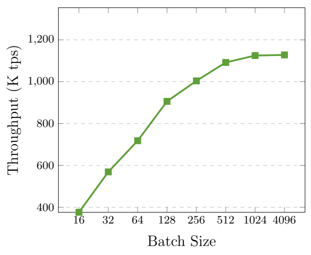
Clue is our subtle design to support native provenance in LedgerDB. We evaluate the performance of *cSL* using the same experimental environment as above. The performance of clue-related operations mainly depends on the total numbers of *journal*s. Hence, we evaluate it with *journal* number varying from 10^4 to 10^9, as shown in Figure 12(a, b). Three typical operation types are used: clue write, point retrieval, and range retrieval. Due to the *cSL* structure, the time complexity of each write is stable (regardless of its *cSL* size). We can observe that the write throughput is almost flat at about 1.5M ops. This high performance of clue index insertion makes the bottleneck of clue writing reside on ledger recording process itself. For point retrieval, due to the cost of *cSL* lookup, the time complexity is directly related to the size of *cSL*. As *cSL* size grows, both read throughput and latency degrade logarithmically. When *journal* number increases from 10^4 to 10^9, *cSL* throughput decreases from around 180K to 75K ops and all latencies are within 1ms. The main bottleneck is disk read bandwidth for random read. Though range retrieval is also related to *cSL* size, it is more sensitive to the size of the reading range, as each *journal* in the range needs to be fetched from storage. A range retrieval of 100 *journal*s reaches around 1500 ops with 100ms latency.

We also evaluate the performance for using RocksDB as the clue index. To obtain the best performance of RocksDB, we turn off its WAL, non-sync write and multi-thread read/write. Compared to RocksDB, *cSL* gets up to 1/50 write latency and 10× write throughput, as well as 2.5× point read throughput, as shown in Figure 12(c, d). For index writes, *cSL* only needs to append the new cell at its tail with $O(1)$ cost. In contrast, RocksDB has to search and insert a new cell in memtable with $O(\log n)$ complexity. In addition, *cSL* also eliminates costly compaction process that results in significant write amplification in RocksDB. For point reads, *cSL* only needs a binary search on one skip list, while RocksDB needs to search among multiple SST files. For range reads, RocksDB's throughput is more than 10K ops and up to 14× of *cSL*. It is because that the range retrieval of related *journal*s requires random I/O in *cSL*, but sequential I/O in RocksDB. In summary, *cSL* is a write-optimized structure as expected, and friendly to random reads.

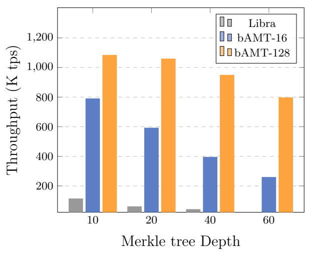
8.3 Performance of bAMT

Recall that the time complexity of *bAMT* is discussed in § 5.1. We use SHA256 hash as a primitive in this evaluation. Figure 13(a) shows the throughput of a fixed 40-height *bAMT* with different sizes. As can be seen, the throughput peak appears when the batch size reaches 1000. After that, the throughput converges due to the cost of frequent hash calculation. In our test environment, SHA256 has around 1000K ops. Figure 13(b) shows the throughput comparison between Merkle accumulator in Libra and *bAMT* with two sizes, i.e., 16 and 128. The throughput of bAMT-16 is 10× of that in Libra on average, because of its batch aggregation of root calculation. From a closer look, it is 7× for 10-height trees, and 12× for 60-height trees. bAMT-128 gets an average 20× of that in Libra (and 36× for 60-height trees).

The choice of *bAMT* size is a trade-off between performance and proof granularity. Large-sized *bAMT* gains a high insertion throughput with coarse-grained proofs for the *journal*s. In contrast, small-sized *bAMT* sustains lower performance, but supports fine-grained proofs. LedgerDB provides the interface of *bAMT* size configuration for users to tune for their specific scenarios. Users can specify their interval parameter to enforce the proof finalization, which enables their own trade-off.

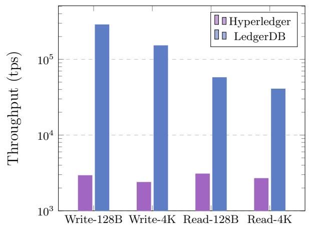


(*a*) bAMT root calculation

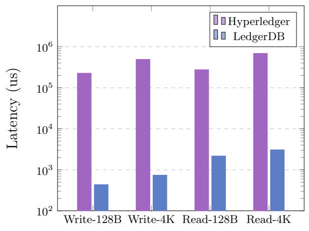


(*b*) bAMT vs. Libra

Figure 13: bAMT performance evaluation and comparison with Libra Merkle tree accumulator.



(*a*) Throughput comparison



(*b*) Latency comparison

Figure 14: Sytem performance comparison between LedgerDB and Hyperledger Fabric in the same notarization application.

8.4 Application Evaluation

Data notarization is a typical application for both ledger database and blockchain. We hence compare the performance in this application between LedgerDB and Hyperledger Fabric. Data notarization system stores various kinds of evidentiary records (proofs), and each piece of record is identified by an *id*. A new record is inserted via a put operation (`AppendTx` in LedgerDB), and can be later retrieved using its *id*. In LedgerDB, we implement this with clues. In Fabric, we conduct it within a smart contract using *PutState* and *GetState*.

Hyperledger Fabric is deployed on the same experimental environments as mentioned. A single-channel ordering service runs a typical Kafka orderer setup: 3 ZooKeeper nodes, 4 Kafka brokers, 5 peers as endorsers and 3 Fabric orderers (all on distinct VMs). We randomly generate a large amount of evidences (more than 20 million), and implement two benchmark tools *PutBench* and *GetBench* for both systems to evaluate their write and read throughput. The experimental results are shown in Figure 14. Fabric gets 2700 write tps and 2900 read tps on average, with 365ms write latency and 490ms read latency. LedgerDB reaches around 83×, 17× for write and read throughput compared to Fabric, with 1/600, 1/200 write and read latency respectively. In summary, LedgerDB significantly outperforms permissioned blockchains in all aspects due to its centralized architecture and various optimizations.

9. RELATED WORK

In database community, there are a wide range of related functionalities such as temporal table [24, 39, 3, 9], audit table [1, 2, 45], time series database [41, 33, 19, 44], and tense database [38]. Both temporal and audit table have a certain level of auditability. However, they are lack of integrity guarantee and non-repudiation, where malicious operations can be done easily. Oracle introduces the blockchain table in DBMSes to prevent such user and administrator fraud behaviors in a trust-but-verify way [35], but it still lacks external auditability and verifiable lineage. Time series database optimizes temporal data manipulation but lacks data auditability. Tense database claims the importance of universal time with the consideration of history slots, transitional and current states [15]. Its intrinsic design is still the usage of temporal tables.

Among those products and researches that blur blockchain and database boundaries [42, 46, 20], QLDB [7, 13] aims for immutable and verifiable ledger applications, who brings DLT back to a centralized architecture. However, it has own limitations: the LSP can tamper data in the backend; and the ledger user can also collude with LSP to forge new ledgers to deceive a third-party. That is to say, it lacks auditability in practice. In addition, its ledger-level verification without trusted anchors and purging functionality also inherits blockchain's heavy CPU and storage costs. Last but not least, its built-in transaction support complicates the system and limits its throughput. Similar CLT idea also appears in ProvenDB [40], which is compatible with MongoDB and keeps database changes by maintaining logical versions whose proofs are tangled with public blockchain evidence. Libra [16, 5] uses transaction-granularity ethics whose world states is at the transaction level, i.e., each transaction has its own path to the digest of Merkle root. BigchainDB [26] and OrbitDB [36] have certain levels of database capacities, but still belong to the DLT category.

10. CONCLUSION

In this paper, we introduce LedgerDB, a centralized ledger database that supports universal audit and verification for applications from mutually distrusting parties. It provides strong auditability with the help of TSA time notary anchors generated by a two-way peg protocol. LedgerDB offers excellent system performance, including high write throughput optimized by a *bAMT* model, and high verification efficiency optimized by trusted anchors. It supports verifiable data removals, i.e., *purge* and *occult*, to meet real-world requirements, which break immutability in blockchains but still preserve data verifiability. In addition, native provenance primitives are provided to ease the application development. The experimental results demonstrate that LedgerDB achieves superior throughput (i.e., 300K TPS), which is more than 80× higher than that in the state-of-the-art permissioned blockchain (i.e., Hyperledger Fabric). LedgerDB has been widely used in real customer businesses on Alibaba Cloud, such as copyrights (with clue) and supply chain (with federated ledger), and it becomes a fascinating alternative to permissioned blockchains.

11. ACKNOWLEDGMENTS

We thank reviewers for their insightful feedback. We appreciate Haizhen Zhuo, Zhonghao Lu and Shikun Tian for their development on LedgerDB storage kernel, and Tianyi Ma, Zhenhua Dong, Zhihui Zhu and Kan Chen for their development on Alibaba LaaS platform. We also express our gratitude to Yayang Guan and Zhengsheng Ye who offered significant help during LedgerDB's productization process on Alibaba Cloud.

12. REFERENCES

[1] R. Agrawal, R. Bayardo, C. Faloutsos, J. Kiernan, R. Rantzau, and R. Srikant. Auditing compliance with a hippocratic database. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 516–527. VLDB Endowment, 2004.

[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 143–154. Elsevier, 2002.

[3] M. Al-Kateb, A. Ghazal, A. Crolotte, R. Bhashyam, J. Chimanchode, and S. P. Pakala. Temporal query processing in teradata. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 573–578. ACM, 2013.

[4] Alibaba. Alibaba cloud baas(blockchain as a service). https://www.aliyun.com/product/baas, 2018.

[5] Z. Amsden et al. The libra blockchain. *White Paper, From the Libra Association members*, 2019.

[6] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

[7] AWS. Amazon quantum ledger database (qldb). https://aws.amazon.com/qldb, 2018.

[8] M. Azure. Microsoft azure blockchain service. https://azure.microsoft.com/services/blockchain-service, 2018.

[9] M. H. Böhlen. Temporal database system implementations. *ACM Sigmod Record*, 24(4):53–60, 1995.

[10] T. Bouchard and G. Benson. Electronically verified digital signature and document delivery system and method, July 25 2006. US Patent 7,082,538.

[11] R. G. Brown, J. Carlyle, I. Grigg, and M. Hearn. Corda: an introduction. *R3 CEV, August*, 1:15, 2016.

[12] V. Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3:37, 2014.

[13] A. C. C. D. Kadt. Introduction to amazon quantum ledger database (qldb). https://www.youtube.com/watch?v=7G9epn3BfqE, 2018.

[14] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[15] T. Cloud. A survey for historical data computing and management of aws qldb and tencent tdsql. https://cloud.tencent.com/developer/article/1374068, 2018.

[16] L. Consortium. An introduction to libra. *White Paper, From the Libra Association members*, 2019.

[17] T. T. A. Dinh, R. Liu, M. Zhang, G. Chen, B. C. Ooi, and J. Wang. Untangling blockchain: A data processing view of blockchain systems. *IEEE Transactions on Knowledge and Data Engineering*, 30(7):1366–1385, 2018.

[18] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1085–1100. ACM, 2017.

[19] T. Dunning and E. Friedman. Time series databases new ways to store and access data, volume 1st edit. *Oreilly and Associates*, 2014.

[20] M. El-Hindi, M. Heyden, C. Binnig, R. Ramamurthy, A. Arasu, and D. Kossmann. Blockchaindb-towards a shared database on blockchains. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1905–1908. ACM, 2019.

[21] Ethereum. https://www.ethereum.org, 2014.

[22] Everledger. https://www.everledger.io/, 2018.

[23] Gartner. Amazon qldb challenges permissioned blockchains. https://www.gartner.com/en/documents/3898488/amazon-qldb-challenges-permissioned-blockchains, 2019.

[24] K. Kulkarni and J.-E. Michels. Temporal features in sql: 2011. *ACM Sigmod Record*, 41(3):34–43, 2012.

[25] S. Matthew. Merkle patricia trie specification. *Ethereum, October*, 2017.

[26] T. McConaghy, R. Marques, A. Müller, D. De Jonghe, T. McConaghy, G. McMullen, R. Henderson, S. Bellemare, and A. Granzotto. Bigchaindb: a scalable blockchain database. *white paper, BigChainDB*, 2016.

[27] R. C. Merkle. Protocols for public key cryptosystems. In *1980 IEEE Symposium on Security and Privacy*, pages 122–122. IEEE, 1980.

[28] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42, 2016.

[29] C. Mohan. Tutorial: blockchains and databases. *PVLDB*, 10(12):2000–2001, 2017.

[30] C. Mohan. State of public and private blockchains: Myths and reality. In *Proceedings of the 2019 International Conference on Management of Data*, pages 404–411. ACM, 2019.

[31] J. Morgan. Quorum whitepaper. *New York: JP Morgan Chase*, 2016.

[32] S. Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[33] S. N. Z. Naqvi, S. Yfantidou, and E. Zimányi. Time series databases and influxdb. *Studienarbeit, Université Libre de Bruxelles*, 2017.

[34] NCAC. National copyrights administration of the people's republic of china. http://en.ncac.gov.cn, 2020.

[35] Oracle. Oracle blockchain table. https://blogs.oracle.com/blockchain/, 2019.

[36] OrbitDB. Orbit db. https://orbitdb.org, 2018.

[37] M. C. Osborne, J. W. Sweeny, and T. Visegrady. Timestamp systems and methods, Dec. 29 2015. US Patent 9,225,746.

[38] A. Pan, X. Wang, and H. Li. Conceptual modeling on tencent's distributed database systems. In *International Conference on Conceptual Modeling*, pages 12–24. Springer, 2018.

[39] D. Petkovic. Temporal data in relational database systems: a comparison. In *New Advances in*

Information Systems and Technologies, pages 13–23. Springer, 2016.

[40] ProvenDB. Provendb: A blockchain enabled database service. https://provendb.com/litepaper/, 2019.

[41] S. Rhea, E. Wang, E. Wong, E. Atkins, and N. Storer. Littletable: A time-series database and its uses. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 125–138. ACM, 2017.

[42] A. Sharma, F. M. Schuhknecht, D. Agrawal, and J. Dittrich. Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In *Proceedings of the 2019 International Conference on Management of Data*, pages 105–122. ACM, 2019.

[43] P. Snow, B. Deery, J. Lu, D. Johnston, and P. Kirby. Factom: Business processes secured by immutable audit trails on the blockchain. *Whitepaper, Factom, November*, 2014.

[44] Timescale. Timescale db. https://docs.timescale.com/latest/main, 2018.

[45] N. Waraporn. Database auditing design on historical data. In *Proceedings of the Second International Symposium on Networking and Network Security (ISNNS'10). Jinggangshan, China*, pages 275–281, 2010.

[46] C. Xu, C. Zhang, and J. Xu. vchain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 International Conference on Management of Data*, pages 141–158. ACM, 2019.