Ubiquitous Verification in Centralized Ledger Database

Xinying Yang[†], Sheng Wang[†], Feifei Li[†], Yuan Zhang[§], Wenyuan Yan[†]
Fangyu Gai[†], Benquan Yu[§], Likai Feng[§], Qun Gao[†], Yize Li[†]
{xinying.yang, sh.wang, lifeifei, yuenzhang.zy, daniel.ywy,
gaifangyu.gfy, benquan.ybq, likai.flk, gaoqun, yize.lyz}@alibaba-inc.com
[†]Alibaba Group [§]Ant Group

***Abstract*—Verifiability is the backbone of most ledger systems to realize credible authentication. However, existing permissioned blockchains and centralized ledger databases lack rigorous verifiability to authenticate all facts (i.e., *what-when-who* validation). Besides, they suffer from high verification cost to a continually growing immutable storage. In this paper, we introduce verification principles behind LedgerDB, a centralized ledger database that achieves both strong external auditability and fast verification. We coin a novel concept called *Dasein Verification* that composes of three validation factors *what-when-who* to formalize ledger auditing. Regarding *what*, LedgerDB devises *fam* (fractal accumulating model) to accelerate existence verification, and *CM-Tree* for efficient lineage verification. Verifiable data mutations are also supported. For *when*, we discuss attacks on existing time pegging protocols that compromise the authenticity of timestamps, and propose a time notary protocol to resolve those threats. Evaluations show that *fam* and *CM-Tree* significantly outperform traditional approaches. Compared to Hyperledger Fabric, LedgerDB achieves 23× higher verification throughput with 500× lower latency in notarization applications, and 3× higher throughput with 300× lower latency in lineage tracking applications. As a public-cloud ledger service, the end-to-end verification latencies of LedgerDB are on average 50× and 1000× lower than that of QLDB in the above applications, respectively.**

I. Introduction

Data verification is essential for both decentralized and centralized ledger technologies. Blockchain is the most in-demand *decentralized ledger technology* (DLT) that provides verifiable tamper-evident objects maintained by mutually distrusting participants [1]–[4]. However, in the latest trend, *centralized ledger database* (CLD) attracts more and more attentions back to *centralized ledger technology* (CLT) [5]–[8]. CLD systems utilize their representative 'trust but verify' methodology to achieve ledger's integrity assurance and non-repudiation clearance, but lack external auditability that needs rigorously verifiable authentication. To solve this problem, we designed LedgerDB [9], a centralized ledger database that achieves both strong external auditability and fast verification.

In our earlier paper [7], we provided an overview of LedgerDB's architecture. The key contribution of [7] is to show that, as a CLD system, LedgerDB not only provides superior write and read performance compared to permissioned blockchains, but also achieves the same level of auditability as in permissionless blockchains. In this paper, we focus on the topic of verification mechanisms in our system. We first introduce a new concept called ***Dasein Verification*** to better formalize verification demands behind a ledger system. We then explain how LedgerDB becomes the first *Dasein*-complete CLD system and achieves high verification efficiency.

The concept *Dasein* is introduced by existential philosopher Martin Heidegger [10], aiming to express the existence of things (e.g., being and time). Similarly, we use *Dasein* to embody the auditability of object existence in LedgerDB. As the perceptive embodiment in ledger systems, the *Dasein* of a piece of electronic data includes three essential elements, which we call a ***what-when-who*** (***3w***) factorization — existence verification relates to ***what***; time verification guarantees ***when***; and non-repudiation answers the question of ***who***. We call a ledger system *Dasein*-complete if all ***3w*** factors of stored data can be rigorously verified.

We argue that *Dasein*-completeness is indispensable to satisfy real-world judicial auditing [11]–[13] and it covers a wide range of use cases. For example, consider a national Grain-Cotton-Oil (GCO) supply chain which involves multiple corporations such as banks, oil manufacturers, cotton retailers, suppliers, and grain warehouses. All the participants need to append their manuscripts, invoice copies, receipts as records on an auditable ledger. With *Dasein*-completeness, any record on the ledger can be auditable by any external parties in terms of ***what-when-who***. Similarly, such a feature is also required by many other applications, such as authorship and royalty notarization [14] and luxury merchandise lineage [15]. However, existing ledger systems fail to reach *Dasein*-complete, though each individual factor has already been extensively considered in many systems. We categorize existing verification mechanisms and their limitations related to ***3w*** factors of *Dasein* verification as follows:

The ***what***. To verify the existence of a transaction, blockchains usually rely on Merkle trees [16]. Bitcoin [1] organizes transaction digests from the same block as a Merkle tree. Hence, a light node can verify transactions using a simplified payment verification (SPV) approach without downloading the entire ledger. Ethereum [2] uses account-based model and adopts Merkle Patricia Tree (MPT) [17] to verify all historical balances. Since each data slot on the MPT path should be maintained, its space overhead is also high. The Diem blockchain [18] blurs block concept in a Merkle

accumulator model. Each transaction becomes an incremental leaf node, which generates corresponding Merkle root hash as its fine-grained tamper proof, rendering high storage and verification overhead. CLD systems such as SQL Ledger [19], QLDB [20], [21] and Oracle blockchain table [22], [23] also use simple Merkle tree models that lead to similar problems above.

The ***when***. Time information is critical for many applications. However, few ledger systems consider rigorous authenticity of timestamps in their designs. In permissionless blockchains, timestamps are credible due to the peer-to-peer architecture, but cannot be used for judicial purposes. Permissioned blockchains like Hyperledger Fabric [24]–[26] fail to provide credible time for external audit, because timestamps on the ledger can be tampered when most peers collude. Regarding CLD systems, QLDB does not consider malicious behaviors from LSP (ledger service provider), who might fabricate timestamps. ProvenDB [27] submits transaction digests to a public blockchain (e.g., Bitcoin) periodically to gain external timestamp evidence. In this case, though LSP cannot directly tamper with a timestamp, it can still infinitely delay its actual effective time (as discussed in § III-B1). SQL Ledger [19] provides *forward integrity* [28], which assumes LSP is trusted until the hash of modification is securely (and immutably) stored outside of the systems. This makes them fail to provide a rigorous ***when*** audit.

The ***who***. It is easy for DLT systems to verify the authenticity and non-repudiation of data from a user (in permissionless blockchain) or an organization (in permissioned blockchain) using digital signatures. In contrast, since CLD systems like QLDB and ProvenDB do not consider the moral hazard of the LSP, the LSP can repudiate for previous request responses by tampering log data to fabricate a user's operation (e.g., delete a *journal*). This makes them fail to provide a rigorous ***who*** audit.

In summary, Existing DLT systems cannot satisfy *Dasein*'s ***when***, while conventional CLD systems cannot satisfy *Dasein*'s ***when*** and ***who***. In addition, all mentioned systems are inefficient in ***what*** verification, especially for data lineage verification, which is a notable real-world application. Towards designing a *Dasein*-complete CLD system with high verification efficiency and low storage overhead, we introduce ubiquitous verification mechanisms adopted in LedgerDB. We summarize our main contributions as follows:

- We coin a concept of *dasein verification* to formalize *what-when-who* validation for ledger databases, which has been adopted in LedgerDB. To meet all *Dasein* factors, we propose an overall framework and specific mechanisms with external ledger auditability. To the best of our knowledge, LedgerDB is the first CLD system that provides native *Dasein* verification (i.e., *Dasein*-complete) for judicial audit [11]–[13].
- To support fast ***what*** verification, we devise an advanced transaction accumulator called *fractal accumulating model* (fam) in § III-A to organize journal digests, and further enhance it with trusted anchors. For ***when*** verification, we first introduce attack models that compromise the authenticity of timestamps in existing time pegging protocols. To achieve globally credible timestamps in action, we then propose a time notary protocol called *Time Ledger* (*T-Ledger*), which provides high write throughput and makes above attacks impractical.
- To enable efficient *N-lineage* verification, we design a two-layer clue merged tree (CM-Tree) that guarantees the authenticity of application-level data lineage (§ IV). The verification throughput of CM-Tree is 33× higher than ccMPT used in [7] and the latency is 24× lower. Advanced mutation verification variants (for *purge* and *occult*) are also provided.
- We conduct empirical evaluation to validate the verification efficiency and completeness of LedgerDB. The results show that LedgerDB provides excellent verification throughput for data notarization and lineage applications, 23× and 3× higher compared to Hyperledger Fabric. At the same time, it achieves 500× and 300× lower latency on average. As a public-cloud ledger service, LedgerDB gets on average 56× and 1000× lower latency in above applications compared to QLDB.

The rest of this paper is organized as follows. We first give our motivation and an overview of LedgerDB in § II. We introduce the concept of *Dasein* verification and our corresponding designs in § III, We describe the *Dasein*-complete audit in § V and the *N-lineage* verification mechanism in § IV. We provide experimental evaluations in § VI. Finally, we discuss related work in § VII and conclude in § VIII.

II. Motivation and Overview

In this section, we first introduce verifiabilities in existing ledger systems and their limitations, which motivate the design of LedgerDB. After that, we present an overview to LedgerDB.

A. Ledger Systems and Limitations

We categorize ledger systems into decentralized ledger techniques DLT (e.g., blockchain) and centralized ledger techniques CLT (e.g., CLD systems) according to their distinct architectures. However, from the perspective of verification mechanisms being used, there is no explicit boundary between DLT and CLD. In particular, all typical ledger systems leverage tree-based (e.g., Merkle tree) models to verify integrity and signature-based models to withstand repudiation. Here we look into ledger verification mechanisms used by both DLT and CLD systems, and compare them from six dimensions as summarized in Table I.

- **Trusted Dependency.** Permissionless blockchains are open to the public. Anyone can join the network without involving any central authority. Permissioned blockchains' participants are usually pre-registered enterprise-level entities. We call its trusted dependency as 'trust the consortium, but not every single one'. CLD systems follow a centralized architecture. This allows them to adopt a 'trust but verify' approach, which means users have to fully trust the ledger service provider (LSP). This brings in several defects when facing

TABLE I
VERIFIABILITY COMPARISONS BETWEEN LEDGERDB AND OTHER LEDGER SYSTEMS.

System	Trusted Dependency	*Dasein* Support	Verify-Efficiency	Storage Overhead	Verifiable Mutation	Verifiable N-lineage
LedgerDB	TSA(non-LSP)	***what-when-who***	High	Lowest	✓	✓
SQL Ledger	LSP & Storage	***what***-*when*-***who***	High	Medium	✓	✗
QLDB	LSP	***what***	Medium	Medium	✗	✗
ProvenDB	LSP & Bitcoin	***what***-*when*	Medium	Medium	✓	✗
Hyperledger	Consortium	***what-who***	Low	High	✗	✗
Factom	Bitcoin	***what***-*when*-*who*	Medium	Highest	✗	✗

a malicious LSP, where it may collude with users and tamper with the data. To solve the problem, LedgerDB follows a *Dasein* verification framework (§ III) that eliminates the trusted dependency on LSP.

- ***Dasein* Verification.** Compared to traditional database audit that observes user actions and operation trail [29], the essential goal of *Dasein* verification in ledger systems is to further ensure that all the ledger data has not been maliciously tampered with according to predefined rules [7]. Factom [30] is a typical permissionless blockchain broadly used for electronic data notarization. It satisfies rigorous ***what***, non-judicial ***when*** and unrigorous ***who*** (with anonymous mechanism). Since permissioned blockchains do not share their status to open participants, it is easy for them to forge system timestamps if majority members collude, which compromises the ***when*** factor. As mentioned previously, existing CLD systems require full trusted dependency to LSP. Therefore, they are insufficient for rigorous ***when*** and ***who*** verification. To solve the problem, LedgerDB adopts *T-Ledger* (§ III-B) and three-phase signing framework to achieve *Dasein*-complete.
- **Verification Efficiency.** Among these systems, there are two typical data organization models: *block- intensive* model (*bim*) and *transaction-intensive* model (*tim*). The *bim* model (e.g., in Bitcoin, Ethereum) batches transactions into many blocks. These blocks are cryptographically linked as a chain. Theoretically, verifying a transaction in *bim* involves both verifying block links from that block to the genesis and verifying the Merkle tree path from that transaction to the root. In practice, the verification to a block is often done only once when the block is being downloaded from the network. The previously downloaded blocks are regarded as trusted anchors to facilitate the verification of subsequent blocks. Hence, *bim* only needs to verify the Merkle tree path for a transaction at runtime, which is quite fast. In contrast, *tim* (e.g., in Diem, QLDB) abandons the concept of block and adopts transaction-level entanglement to accumulate all transactions in a single Merkle tree. Its verification efficiency is significantly affected when its tree height grows. LedgerDB devises a *fam* tree that combines the benefit of trusted anchors from *bim* and fine-grained transaction verification from *tim*, offering fast transaction existence verification (§ III-A).
- **Storage Overhead.** Both conventional DLT and CLD systems have prohibitive storage overhead due to inherent data immutability. Since no data deletion is allowed, storage overhead is inevitable as data volume grows. To solve the problem, LedgerDB invents a purge operation to remove obsolete data without compromising verifiability (§ III-A2).
- **Regulation Support.** Regulation-violated data (e.g., data that discloses unauthorized personal privacy) should be allowed to delete from a ledger system, but most blockchains and ledger databases cannot support this. To address this problem, LedgerDB implements an occult operation (§ III-A3) that is equipped with *mutation verification* mechanisms to retain verifiability.
- **Verifiable *N-lineage*.** Data lineage is a typical feature in ledger systems. For example, the UTXO model in Bitcoin offers a primitive lineage support. Other systems either build independent lineage engines on top of a ledger [31] or deploy lineage smart contracts inside a blockchain [32]. Nevertheless, they inherently suffer from limited performance and insufficient *Dasein* support as discussed above. LedgerDB implements a native two-layer *CM-Tree* inside its kernel to support fast lineage tracking and verification (§ IV).

B. Threat Model of LedgerDB

The requirement for trusted dependency and *Dasein* verification may vary under different threat models. Here we discuss the threat model of LedgerDB. We start from introducing two common threat scenarios for ledger databases: server-side malicious tampering and LSP-client collusion. In the first scenario, the adversary can be LSP or any attacker who compromises the database server and obtains DBA authority. The adversary might tamper with the incoming data when the server receives a transaction request from the client (*threat-A*). The adversary might also tamper with (e.g., insert, update, delete) an existing *journal*, or forge system timestamps (*threat-B*). In the second scenario, LSP colludes with one or more clients as the adversary. They can collude to modify an existing *journal* to cheat a third-party auditor (*threat-C*).

We assume cryptographic hashes and digital signature algorithms, i.e., SHA256 and ECDSA, are reliable. Based on this, we further assume the identities of all ledger participants are authentic, i.e., they (user, LSP, TSA, and regulator) disclose their public keys certified by a CA. We only trust TSA (Time Stamp Authority) as an authorized third party that can attach a credible and verifiable timestamp to a given piece of data.

C. LedgerDB Overview

LedgerDB [7] is an auditable ledger database with tamper evidence and non-repudiation features. Due to the centralized architecture, its system throughput is significantly higher (exceeding 300,000 TPS) compared to open blockchains. LedgerDB implements a stream file system and an advanced Merkle accumulator to manage *journal*s. Ledger members are registered and authenticated using their public keys (*pk*). They can operate on the ledger via a set of APIs (e.g., `Create`, `Append`, `Verify`) according to their assigned roles.

Figure 1 depicts the two-level *Dasein*-complete framework in LedgerDB: *journal*-level transaction commitment (for ***what*** and ***who*** verification) and ledger-level timestamp entanglement with TSA [33], [34] (for ***when*** verification). For *journal* insertion, the ledger client submits the transaction signed by his/her secret key (*sk*) to the ledger proxy, and provides a proof π_c. The ledger proxy then sends the transaction payload to a shared storage, and sends the proof and payload digest to the ledger server. The server creates a *journal* entry for the transaction and assigns it a unique incremental journal sequence number (*jsn*). At last, the LSP generates a receipt signed by his/her *sk* as a proof π_s to confirm the commitment of that transaction, and sends it to the client. To achieve ledger-level auditability, LedgerDB adopts a two-way time pegging protocol between the ledger and TSA as shown in Figure 1. The digests of the ledger are periodically submitted to TSA who grants a precise universal timestamp and signs on the digest-timestamp pair, which becomes a proof π_t. After that, the proof is recorded as a special *time journal* and is anchored back to the ledger.

Figure 2 illustrates the data model and ledger structures in LedgerDB. A ledger contains multiple *journal*s, each of which has a *LedgerInfo*. Each *LedgerInfo* records its unique *jsn*, *journal* accumulator (whose leaf node stores the digest of *JournalInfo*) root hash, and *state* accumulator root hash, forming a two-layer *CM-Tree* (i.e., state and clue accumulators in Figure 2). In addition, the *world-state* is maintained by a single-layer state accumulator without clue accumulator. We detail both state and clue accumulators in § III-A and § IV, respectively.

In LedgerDB, verification can be conducted in two different manners: 1) verified at server side when LSP can be fully trusted (e.g., by users who demand high efficiency); 2) verified at client side when LSP is distrusted (e.g., by anyone who can directly access the ledger, such as external auditors).

III. DASEIN VERIFICATION

As introduced in § I, *Dasein* verification represents ***3w*** factors. ***what*** means a certain data actually exists (or used to exist). ***when*** proves the data is actually produced at a certain timestamp. ***who*** determines the exact issuer of that operation. A *Dasein*-complete system should be able to rigorously verify all three factors. In this section, We present major verification mechanisms that make LedgerDB *Dasein*-complete: how to verify the existence (***what***) of transactions (§ III-A); how to generate credible timestamps (***when***) that can be verified

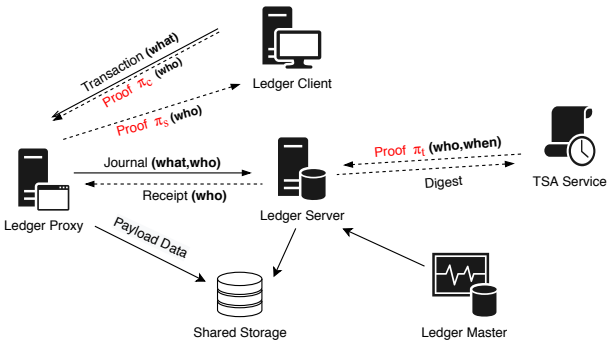


Fig. 1. LedgerDB *Dasein*-complete Framework.

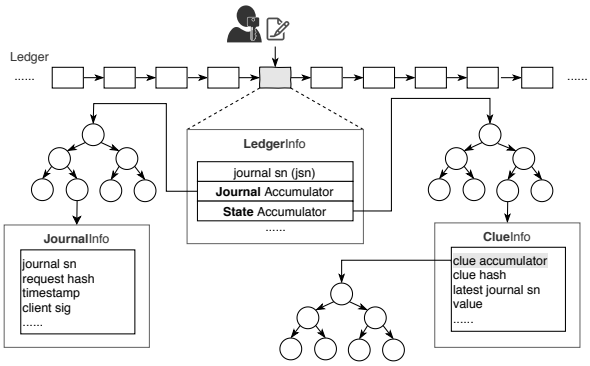


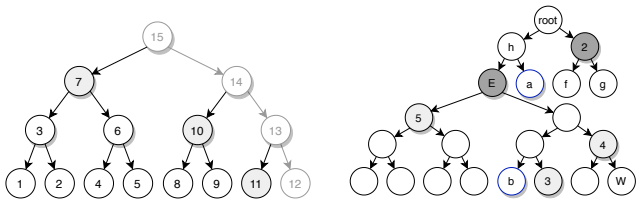
Fig. 2. Ledger Structures in LedgerDB.

(§ III-B); and how to provide non-repudiation (***who***) proofs (§ III-C).

A. Existence Verification (what)

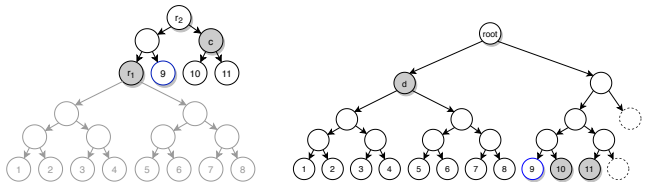
The concept of existence includes *existing* (i.e., a certain data exists on the ledger) and *used to exist* (i.e., a certain digest derived from a certain data used to be on the ledger). Existing verification refers to the integrity check for *journal* and status data. A successful pass of the verification means that the target data exists verbatim on the ledger, while a fail means it is a fake. For example, assume a *journal* with payload `'foobar'` is appended on the ledger, an existing verification for `'foobar'` will pass and for `'foopar'` will fail. Besides the correctness, other design considerations of a good verification mechanism are verification efficiency and storage cost, especially when data keeps growing (§ III-A1). For used-to-exist verification (also called mutation verification), LedgerDB proposes *purge* and *occult* for different scenarios (§ III-A2 and § III-A3). The corresponding verification processes are reliable and fast, without scanning the entire ledger.

1) Fractal Accumulating Model: Recall that *block-oriented* model (*bim*) and *transaction-oriented* model (*tim*) have been discussed in § II-A. Among them, *bim* has relatively high verification efficiency. However, in order to limit the maximum latency between consecutively committed blocks, there are often a large number of blocks each containing a small number of transactions. This leads to large storage overhead, even for a light client that only maintains block headers. In contrast, *tim* offers fine-grained verification for each transaction without



(a) Shrubs tree (b) fam tree

Fig. 3. Shrubs and fam tree verification.



(a) Anchor-based *fam* verification (b) *tim* verification

Fig. 4. Differences between *fam* and *tim*.

storing block headers as in *bim*, but its verification efficiency decreases as data grows. To solve the problem, LedgerDB proposes a *fractal accumulating* model (*fam*) based on an advanced Merkle tree called Shrubs [35]. The *fam* achieves high verification performance as in *bim*, and at the same time has low storage overhead as in *tim*.

The used Shrubs tree has $O(1)$ insertion time complexity, and its prototypical verification cost is the same as in *tim* (e.g., Diem) and *bAMT* [7]. It provides a node-set proof for verification before the binary tree is full, instead of a root-node proof used in conventional Merkle trees. Take Figure 3(a) as an example, the serial number is based on their being sequence. The proof for $cell_1$ is $\{cell_1\}$ itself. When $cell_2$ arrives, the digest of $cell_3$ is calculated, and $\{cell_3\}$ becomes the proof for $cell_2$. When $cell_4$ arrives, its proof becomes $\{cell_3, cell_4\}$. When $cell_5$ comes, the digest of $cell_6$ and $cell_7$ are calculated, and $\{cell_7\}$ becomes the proof for $cell_5$. Similarly, the proof for $cell_8$ is $\{cell_7, cell_8\}$; the proof for $cell_9$ is $\{cell_7, cell_{10}\}$; and the proof for $cell_{11}$ is $\{cell_7, cell_{10}, cell_{11}\}$. Finally, when $cell_{12}$ is accumulated, all digests on the path of $[cell_{13}, cell_{14}, cell_{15}]$ are calculated, and the root hash $\{cell_{15}\}$ becomes the proof for $cell_{12}$. This approach avoids unnecessary accumulation for intermediate nodes, making its insertion extremely fast.

On top of Shrubs tree, *fam* borrows the linked-block entanglement used in blockchain, and refines its equally-linked layout into fractally-organized layout according to the following rule:

Rule 1. When the current tree of a given size is full, its root node becomes the first leaf node of a new tree.

Figure 3(b) illustrates the derivation of a *fam* Merkle tree. Normal leaf nodes store *journal* digests, formalizing a Merkle accumulator. As defined in Rule 1, when the number of cells exceeds a predefined threshold (e.g., $cell_E$), we will initiate a new *fam* that contains the root hash of the current *fam* as its first leaf node. If we treat the traditional Merkle accumulating process as a single accumulation cycle, *fam* divides the accumulating process into many sub-cycles recursively. We can see a watermark cell (e.g., $cell_W$) would regress the *fam* tree accumulation for a brand-new cycle, by resuming its current Merkle root hash as a split cell (e.g., $cell_E$) for the next epoch. This makes *fam*'s watershed skew by the tree figure compared to linear blocks in blockchain. Unlike normal node, the split cell $cell_E$ carries the digest of the entire previous sub-tree, and enters new epoch of *fam* sub-cycle. We call it a merged leaf type compared to the equal block/leaf type in blockchains (e.g., Bitcoin, Diem). The verifying path for $cell_a$ is $\{cell_E, cell_2\}$. The path for $cell_b$ is $\{cell_3, cell_4, cell_5\}$ with its nearest epoch $cell_E$'s root hash as proof.

To further improve the verification efficiency of *fam*, we use the concept of *trusted anchor*s as system or client specified credible checkpoints. They indicate that all data before them have been verified, and this facilitates subsequent verifications. There are two typical scenarios for trusted anchors: *block-oriented* anchor (*boa*) for light node in *bim* and *accumulator-oriented* anchor (*aoa*) in *tim*. Note that Bitcoin has the first prototype of *boa*. In it, a light client downloads block headers with verifying its validation, and stores headers locally. These block headers are used as *boa* trusted anchors, which means that these headers are all proven to be valid. As a result, a transaction existence verification is reduced to an SPV Merkle path verification (§ II-A). The space complexity of *boa* is $O(n)$, which is proportional to the number of blocks.

In contrast, LedgerDB adopts an *aoa* model called *fam-aoa*, where the integrity of data before trusted anchors is trusted as shown in Figure 4(a). Before a new trusted anchor is set, all earlier ledger data must be cryptographically verified. Figure 4(b) illustrates a normal *tim* tree compared with *fam-aoa* in Figure 4(a). As can be seen, for the same $cell_9$, *fam-aoa*'s Merkle path contains two nodes $\{r_1, cell_c\}$, while the normal *tim*'s Merkle path involves three nodes $\{cell_{10}, cell_{11}, cell_d\}$. For the fractal tree with height H, the *fam-aoa* tree limits the verification path to: 1) $O(H)$ fixed cost for all historical epochs and 2) $O(H-1)$ expected cost for the current epoch. For the normal *tim* tree, the cost is $O(\log n)$, increasing with the data volume. Our implementation of *fam* uses a fixed fractal height. Assume that the fixed fractal height is δ, the total number of *journal*s within each sub-tree is 2^δ, and the latest *jsn* is hence $\lambda = 2^{\delta+1}+2$. In this case, normal *aoa* has three levels, which means there are 3 sub-trees and the first two are full (i.e., each with 2^δ leaf nodes). For *journal* λ, the verification cost of *fam-aoa* is $O(2)$, while the cost in normal *aoa* is $O(\delta+2)$.

2) Purge Verification: As discussed in § II-A, high storage overhead in current ledger systems motivates our support of *purge* operation, which provides a type of mutation verification. A purge operation erases consecutive *journal* entries from genesis to a designated *jsn* on ledger. Here we investigate the feasibility for purge in action. Firstly, in ledger applications, the value of historical data mainly lies in its capability to prove the authenticity of the current state, rather than its content itself. For example, we seldom care about our obsolete bank

statements that were ten years ago. But we have to make sure that our current balance is correctly derived from all historical transactions without any mismatching (e.g., missed a roll-in transaction). In addition, it is much easier for all users in a CLD system to reach their agreement of purging existing data, compared to that in permissionless blockchains.

When executing a purge operation, a pseudo genesis is created and stored before the first unpurged block, replacing the role of the last purged block. It replicates the data on genesis, as well as snapshot states of the designated purge point (e.g., clue and membership status). The purge operation itself is recorded on ledger as a purge *journal*, which is doubly linked with the new pseudo genesis for mutual proving and fast locating. The authenticity of a purge *journal* is guaranteed by Prerequisite 1 and can be verified by Protocol 1:

Prerequisite 1. Multi-signatures from DBA and all related members (i.e., who has *journal*s before the purge point) are gathered for the purge *journal*.

Protocol 1. The latest pseudo genesis is viewed as the ledger's genesis block when verifying subsequent *journal*s.

Combined with *fam*, the latest LedgerDB provides an option for users to customize whether the relevant nodes on *fam* are expected to be erased or not in the latest purge. For the case that the erasure is expected, after aligning trusted anchor to the purging point, the nodes to be retained are all latter nodes of the next node of the purging node's Merkle path, meaning that all left nodes on this path can be erased. For the case that the erasure is not allowed, the *fam* tree can be entirely retained as its space consumption is acceptable (we only need digest but not raw payload). In addition, users can specify some milestone *journal*s that will survive from the purge and store them in a *survival* stream. In this way, these milestone *journal*s can be retrieved and verified to fulfill business demands (e.g., keep historical block trades only).

3) Occult Verification: In practice, uploaded data in the ledger can sometimes contain regulation-violated or illegal information. For example, if personal identities and related information are recorded on the ledger without authorization, it will be a violation of the law in most countries. To handle such regulatory problems, LedgerDB has an *occult* operation that supports another type of mutation verification. An occult operation hides the *journal* with a designated *jsn* and retains its hash digest on the ledger, which does not compromise the ledger's verifiability [36]. The authenticity of an occult operation is guaranteed by Prerequisite 2 and the verification process of an occulted *journal* is defined in Protocol 2:

Prerequisite 2. Multi-signatures from DBA and *regulator* role holder are gathered for the occulted *journal*.

Protocol 2. The retained hash in an occulted *journal* is viewed as the original *journal* when verifying subsequent *journal*s.

The occult verification protocol is straightforward and easy to implement. The verifier identifies the type of each *journal* on the ledger: for a normal *journal*, we calculate its hash; and for an occult *journal*, we read its retained hash instead. In this way, the future retrieval of occulted *journal*s becomes impossible, but the entire ledger remains verifiable.

The latest LedgerDB provides an option to execute occult operation either synchronously or asynchronously. A synchronous occult erases the *journal* immediately during the operation. In contrast, an asynchronous occult delays the physical erasure of the *journal*, in case it will still be used by other operators (e.g., *occult by clue* is a common case). When occulting a *journal*, we first set its occult bit using an occult bitmap index [7] without actually erasing its payload. Till now, it is marked as deleted and can not be retrieved anymore. The data erasing is performed by data reorganization utility during system idle batch from the $occulted$ anchor, which indicates data to be erased.

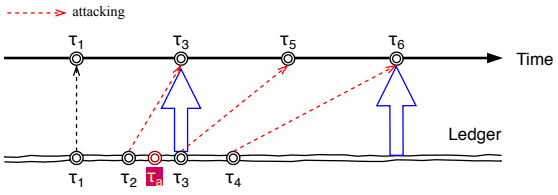
B. Time Verification (when)

Reliable timestamp is important in ledger systems. To allow rigorous third-party auditing, the ledger system needs a reliable mechanism to provide authentic and verifiable timestamps. Recall that the most auditable timestamp mechanism in blockchains is the one used by Bitcoin. Specifically, the winner who solves the PoW puzzle will have its new timestamp validated before others' acceptance. Due to its public accessibility and large ecosystem, this coarse-grained timestamp validation mechanism makes all timestamps on Bitcoin credible for the real world. Hence, it attracts many applications to anchor its timestamp onto Bitcoin. In contrast, traditional database systems most use local system timestamps, which are unreliable. For those CLD systems that provide verifiable timestamps, their timestamp generation protocols are vulnerable to several attacks as we discuss below.

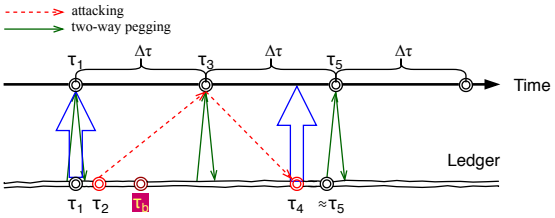
1) Timestamp Attacking Analysis for CLD Systems: ProvenDB discloses its one-way timestamp pegging protocol, which periodically submits digests to Bitcoin [27]. This realizes provable timestamps for ledger data but only to some limited extent. In particular, there is a malicious time magnification defect in this protocol — the timestamp assignment to a sequence of data submitted to the notary can be delayed arbitrarily as long as their relative order remains the same. Figure 5(a) depicts this attack, which we called *infinite time amplification* attack. In this figure, the relative time order is $\tau_1 < \tau_2 < \tau_3 < \tau_4 < \tau_5 < \tau_6$. A τ_2 *journal* is submitted for timestamp, but is anchored at latter time τ_3, while the real τ_3 *journal* is anchored at τ_5. This is same for τ_4 to τ_6. We can see that the τ_a *journal* (whose actual time is in range (τ_2, τ_3) can be tampered during the time range (τ_3, τ_5). Therefore, such a one-way timestamp pegging protocol cannot withstand *threat-B* and *threat-C* (§ II-B).

To solve this infinite time amplification problem in one-way pegging, LedgerDB proposes a two-way timestamp pegging protocol. The protocol relies on Prerequisite 3 and is defined in Protocol 3:

Prerequisite 3. The TSA is trusted as an independent authority, and its *pk* is certified by CA.



(a) Infinite time amplification in one-way pegging



(b) Finite malicious time-window in two-way pegging

Fig. 5. Scenarios for timestamp attacks.

Protocol 3. The TSA 1) assigns the current timestamp to the digest submitted by a ledger and signs the timestamp-digest pair, and then 2) anchors the signed time *journal* back to that ledger.

First, a ledger submits its digest to TSA, and TSA assigns a timestamp and provides its signature on the digest-timestamp pair. This endorsement proves that a certain data (which can be derived from that digest) exists before the assigned timestamp. Second, this TSA-signed information is anchored back to that ledger as a *time journal*, which contains digest's authorized timestamp from TSA along with its signature. To avoid the TSA becoming the single point of failure, we utilize a pool of independent TSA services from different authorized entities [33], [34] to further enhance system availability.

This protocol can reduce the malicious time window from infinite to $2 \times \Delta\tau$ as illustrated in Figure 5(b). Consider an attack that starts from the *time journal* anchoring time τ_1. The adversary holds a *journal* generated at τ_2, and submits its digest at τ_3 for TSA endorsement. After TSA replies with the *time journal*, the adversary holds it until τ_4 before anchoring it to the ledger. We can observe that a *journal* between (τ_2, τ_4) can be tampered, and pretend it is generated at any time in the current TSA-endorsed epoch (τ_1, τ_3), such as at τ_b. In this case, the maximum malicious time window occurs when $\tau_2 \approx \tau_1$ and $\tau_4 \approx \tau_5$, which is approximately $2 \times \Delta\tau$. This mitigates *threat-C* in § II-B with $2 \times \Delta\tau$ time-window confidence.

2) Time Ledger: To have better timestamp protection in practice, it is however costly for each ledger to minimize malicious time window $\Delta\tau$, as this will introduce frequent TSA endorsement.

we propose a *Time Ledger* (*T-Ledger*) that is maintained by the LSP as a public TSA notary anchoring service for all ledgers. The *T-Ledger* records digests submitted from other ledgers and acts as an intermediate agent between all registered ledgers and TSA, which constitutes a two-layer time-notary anchoring architecture. The top layer adopts the two-way pegging protocol (Protocol 3) between TSA and *T-Ledger*. *T-Ledger* commits a digest to TSA for each $\Delta\tau$ time span, which is a periodic time notary finalization. The bottom layer adopts an advanced one-way pegging protocol (Protocol 4) between *T-Ledger* and common ledger, which eliminates the time amplification issue (§ III-B1). This protocol is defined as follows:

Prerequisite 4. *T-Ledger* acts as a public ledger containing regular TSA *journal*s that anyone can download and verify.

Protocol 4. 1) A common ledger issues its request containing its digest and local timestamp τ_c. 2) *T-Ledger* accepts the request only if the delay from τ_c is tolerable within a predefined threshold τ_Δ against its own timestamp τ_t, i.e., $\tau_t < \tau_c + \tau_\Delta$.

Note that we offer a *T-Ledger* service on Alibaba Cloud with public access. Its service availability is guaranteed by its SaaS platform on Alibaba ECS (Elastic Compute Service) [37]. Its verifiable content (e.g., notary digests, TSA authorized timestamps, LSP signatures) and public disclosure together provide strong credibility for ***when*** verification. A normal ledger can submit digests to *T-Ledger* with much higher throughput compared to direct TSA interaction. *T-Ledger* seeks TSA proof every second to control $\Delta\tau$ and helps resolve *threat-B* and *threat-C* in action, because it is impractical for malicious tampering within two seconds. If so, it is more practical for the adversary to change the data before submission, which can never be discovered.

C. Non-Repudiation Proof (who)

Besides existence and time verification, non-repudiation proof is another foundation in *Dasein* verification. LedgerDB leverages digital signatures to associate each ledger operator with a real-world entity (e.g., a natural person or a legal person).

There are three major types of parties in LedgerDB that require non-repudiation proof, i.e., ledger client, LSP, TSA. When a ledger client issues a transaction, it will pack its payload data with metadata (e.g., *ledger_uri*, journal type, nonce), and calculate a *request-hash* based on the entire transaction. The client then signs this digest using its *sk*, and this signature becomes the non-repudiation proof, i.e., π_c in Figure 1. After receiving the transaction, the ledger server calculates a *tx-hash* that is the digest of the server-side *journal*, and grants it a unique incremental *jsn*. Later, when transactions fill up a block, a *block-hash* is calculated during block committing. All three digests (i.e., *request-hash*, *tx-hash*, *block-hash*), along with other fields (e.g., *jsn*, timestamp) are packed in the final receipt. The LSP signs this receipt to enforce its non-repudiation, and this signature is kept by the ledger client externally as a proof, i.e., π_s in Figure 1. For the non-repudiation of TSA, it needs to sign the submitted digest and the timestamp before replying to LSP. This TSA-signed *time journal* is recorded on *T-Ledger*, i.e., π_t in Figure 1, which provides a credible universal timestamp and clarifies the non-repudiation from TSA.

IV. VERIFIABLE N-LINEAGE

A. Native lineage (N-lineage)

Lineage is a typical and widely discussed application branch in ledger technologies [32], [38]. Business-level lineage traces all relevant records whose integrity should be validated. A lineage application should provide a validated batch of records of its representative entity's origin and journey lifecycle on ledger. As an example of copyright protection, an artwork is produced in `2005`, whose first royalty is transferred at `2010`. Another transferring happened at `2015`. The lineage verification should track all these `3` records for this artwork, and should verify all their integrities, including the number of records.

UTXO is the first blockchain data lineage model as discussed, it is a kind of *N-lineage* prototype with performance limitation. Existing permissioned blockchain lineage solutions often leverage database systems to manage logic data and its blockchain reference to ensure integrity validating [31], [39], which makes a gap between applications and blockchains. Systematic or specific lineage smart contracts in permissioned chain is another approach, which still remains the *Dasein*-incomplete auditability issue.

Clue is a new concept introduced in our previous paper [7] to support native business-level lineage with a friendly user-defined (KV-like) interface. Clue tracking is a fine-grained route on ledger that represents business logic, and facilitates many real-world best practices. A clue (i.e., a label) is a business logic carries on lineage which is natively supported in LedgerDB. In the above copyright example, a specific clue (e.g., `DCI001`) is assigned for the artwork by client. Each journal appending related will take `DCI001`as an input parameter through API `AppendTx(lg_id, payload, 'DCI001').DCI001` oriented clue verification will retrieve and verify all the 3 relevant journals by `ListTx` and `Verify` [7]. We implemented a write-optimized clue SkipList (cSL) index to perform fast $O(1)$ insertion and $O(log(n))$ read (a clue underlies n journals [7]) in our earlier paper, whose clue-oriented verification is not fully optimized.

B. Clue Merged Tree

As a user-defined key that carries on lineage application logic, clue traces various status according to the key and records corresponding *journal*s on ledger. This status variation is also called the change of world-state in blockchain.

1) State Tree: A typical state tree is the MPT (Merkle-Patricia Trie) conducted in Ethereum [2] for account balance and contract storing and fast historical status verification. We also proposed a dedicated *clue-counter* MPT (*ccMPT*) for a write-intensive design that avoids additional clue-orientated data insertion in our previous design [7]. However, it is not well optimized for clue-oriented verification.

The main procedure in *ccMPT* clue verification is to verify the integrity of the specified $clue$'s counter m, and then verify all the m *journal*'s existences. The m *journal*s existence verification makes a linear expansion of the total cost. As the significant improvement from Shrubs algorithm introduced in § III-A1, its Merkle accumulator insertion cost is $O(1)$, so it is efficient to add respective $clue$'s own sub-accumulator who will increase verification performance without regressing insertion throughput. The low insertion cost in Shrubs model is the backbone of our *CM-Tree* below.

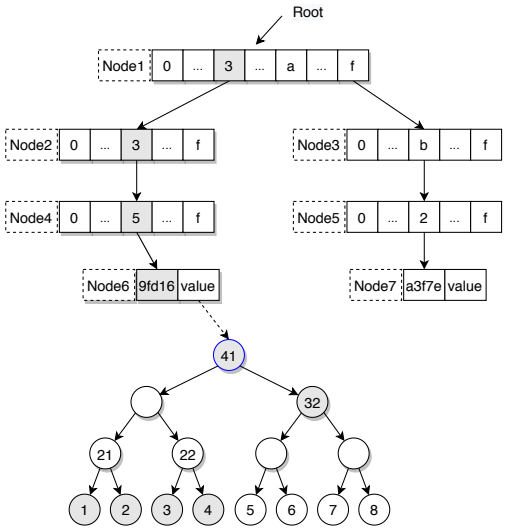


Fig. 6. Verifiable Clue Merged Tree.

2) Merkle and Merkle-Patricia merged tree: Figure 6 depicts our initiative design of the Merkle and Merkle-Patricia merged tree, i.e., clue merged tree (*CM-Tree*), which consists of an MPT (*CM-Tree1*) and many subtrees for clue Merkle accumulating (*CM-Tree2*). The *CM-Tree* performs significantly faster for verification with similar insertion cost compared to *ccMPT*.

CM-Tree1 records all clue's historical and current status. The root hash of the clue MPT is calculated and recorded in every block to capture the verifiable snapshot according to its block version. We use SHA-3 algorithm to scatter a 32-bytes length clue key based on its client specified string to avoid excessive compression and keep the tree balanced. Each *CM-Tree1*'s non-leaf node holds 16 branches, and we keep a configurable top layers cache in memory (e.g., top 6-layers caching cost is around 512MB). Bottom layers including the leaf nodes are stored on disk persistently.

As illustrated in Figure 6 (a shortened clue key and *CM-Tree1* layers are depicted here), $clue$ `3359fd16` is stored on the MPT with its non-leaf nodes and long-tail leaf node for residual `'9fd16'`. The prefix in each node keeps its storing index, which records the position of its child node in the stream storage. The value of $clue$ `3359fd16` leaf node stores its *CM-Tree2* root proof set as described of Shrubs tree in § III-A1. There are 8 *journal*s related to $clue$ `3359fd16` as shown in the bottom *CM-Tree2* in Figure 6.

3) CM-Tree Insertion: A complete *CM-Tree* insertion involves 2 main steps: the top-down $clue$ relevant *CM-Tree2* insertion and the bottom-up *CM-Tree1* root hash calculation.

Regarding the first step, the $clue$ key is searched among

the MPT (up layers from memory and bottom layers from disk) until to find the key position or insert a new leaf node if not found. Then, search its accumulator based on `value` and append the new cell at tail (for a new clue just insert this first cell). In terms of the second step, the new *CM-Tree2* root proof is firstly calculated, so as to update the latest version of the $clue$'s 'value' in *CM-Tree1*. Then, a normal process for MPT hash path calculation is performed up to its root.

C. Clue-oriented Verification

The goal of clue-oriented verification is to ensure the data integrity of its journal entries (i.e., has not been tampered). All relevant *journal*s should be validated in a clue-oriented verification, meaning that any unauthenticated proof will fail the entire verification.

Typical clue verification has two scenes: 1) verify the entire clue so far; 2) verify within a range specified by version (or timestamp) boundaries. They are conducted by our `Verify` API presented as:

```
Verify(lgid,CLUE,*{key,txdata,ρ,root},level)
```

, where `lgid` is the ledger identifier; `CLUE` is an enumeration for $clue$; `key` is the specified clue; `txdata` are the related *journal* set or *CM-Tree2* digest set (for semi-verification) to be verified; ρ is the verifiable path to $root$, which is the credible datum; and `level` differentiates it is operated from client side or server side.

We apply a fast clue-oriented verification algorithm based on *CM-Tree*. Given a ledger $\mathcal{L}$ whose *CM-Tree* is Λ and a specified clue σ: *CM-Tree1* of Λ is Δ_1 and *CM-Tree2* of Λ is Δ_2. A function S (search) takes as input the clue data σ, the version boundaries v_1 and v_2 to be verified, a tree identifier Λ, and outputs a number set $\mathbb{N}$. A function R (retrieve) take as input an enumerated value, a tree identifier Δ (e.g., Δ_1 or Δ_2), a number set $\mathbb{N}$ (or a clue σ for MPT), and outputs a cell entry set $\mathbb{C}$. A function V (validate) takes as input a clue σ, a proving path of cell set $\mathbb{C}$, a *CM-Tree* tree identifier Λ, and outputs a boolean proof π. Both path calculation functions $\mathsf{P_1}$ and $\mathsf{P_2}$ take a number set $\mathbb{N}_i$ as input, and also output a number set $\mathbb{N}_o$. The client-side clue verification (i.e., the `level` parameter in API is `client`) process is defined as follows:

1) $\mathbb{N}_1 = \mathsf{S}(\sigma, v_1, v_2, \Lambda)$, which computes destination leaf cell number set $\mathbb{N}_1$ to be verified for clue σ.
2) $\mathbb{N}_2 = \mathsf{P_1}(\mathbb{N}_1)$, which calculates all needed *CM-Tree2* Merkle proof paths; then, $\mathbb{N}_3 = \mathsf{P_2}(\mathbb{N}_1)$, which calculates all the non-leaf cells that can be calculated using $\mathbb{N}_1$.
3) $\mathbb{N} = \mathbb{N}_2 - (\mathbb{N}_2 \cap \mathbb{N}_3)$, which computes the non-leaf proofs' positions to be retrieved for verification.
4) Fetch all the proof cells $\mathbb{C}_a$ for σ's *value* on *CM-Tree1* according to *CM-Tree2*'s non-leaf proof cell positions $\mathbb{N}$, and leaf cells numbers $\mathbb{N}_1$ by $\mathbb{C}_a = \mathsf{R}(MT, \Delta_2, \mathbb{N} \cup \mathbb{N}_1)$.
5) *CM-Tree1* proof nodes $\mathbb{C}_s$ across layers from the bottom-up are fetched by $\mathbb{C}_s = \mathsf{R}(MPT, \Delta_1, \sigma)$, and replied together with $\mathbb{C}_a$ as an entire proof set for client to verify.
6) $\pi = \mathsf{V}(\sigma, \mathbb{C}_a \cup \mathbb{C}_s, \Lambda)$. Client verifier first verifies the integrity of σ based on 1) $\mathbb{C}_a$ towards its *CM-Tree2*, and then 2) verifies $\mathbb{C}_s$ towards its *CM-Tree1* route.

A proof π is only true when the two layers of *CM-Tree* are all proved. Any invalidation during the process will lead to a false π. The server-side clue-oriented verification is easier compared to above client-side algorithm, it also leverages the first three steps above and doesn't need to retrieve the cell proof set back to client for his own validation (i.e., skip the 4^{th} and 5^{th} step). Instead, the last verifying 6^{th} step is done at the server side.

To better illustrate the proof cell number to be retrieved in the 3^{rd} step, we number the non-leaf cells for demonstration. For a first 4 items' verification of $clue$ `3359fd16`, It is easy to locate all the needed non-leaf proof is $\{cell_{21}, cell_{22}, cell_{32}\}$, while $\{cell_{21}, cell_{22}\}$ is the result-set of $(\mathbb{N}_2 \cap \mathbb{N}_3)$. So only $\{cell_{32}\}$ will be replied to verifier in this case.

V. DASEIN-COMPLETE AUDIT

Audit is a serial of verifications to observe user actions and operation trails based on predefined rules. To better formalize the completeness of ***3w*** verification in ledger systems, we propose a notion called *Dasein*-complete, by fulfilling which the system should be able to provide rigorous auditability to external audit. The goal of this notion is to guarantee that the entire ledger cannot be maliciously tampered by clients, LSP, TSP, as well as their collusion. The definition of *Dasein*-complete is as follows:

Definition 1. A *Dasein*-complete ledger audit passes the entire verification for all *Dasein* dimensions, i.e., ***what***, ***when***, ***who*** verifications.

The audit of the entire ledger takes all *journal*s (including purge, occult, and time *journal*s), as well as the latest receipt returned from LSP as input. Function $\mathcal{V}$ verifies the integrity and signature of all relevant *journal*s. It takes a set of block $\mathbb{B}$ to be verified as input and outputs a proof π. Function $\mathcal{V}'$ verifies the digest consistency between adjacent blocks. It takes two adjacent blocks as input and outputs a proof π'. Function $\mathcal{P}$ verifies the signature. It takes the signed object $\mathbb{O}$ as input and outputs a proof Π. Note that any failure of a verification sub-task (i.e., *false* from function $\mathcal{V}$, $\mathcal{V}'$ or $\mathcal{P}$) will early terminate the entire audit process and return a *failed* status.

1) Prove all $purge$ *journal*s' validity: $\Pi_1 = \mathcal{P}(\mathbb{O}_p)$, where $\mathbb{O}_p$ contains all relevant members. Prove all $occult$ *journal*s' validity: $\Pi_2 = \mathcal{P}(\mathbb{O}_o)$, where $\mathbb{O}_o$ contains regulator and DBA.
2) Locate all *time journal*s n within the specified temporal range. Prove signatures of n *time journal*s. Retrieve n blocks containing *time journal*s, where we denote m_i as the block number of the i^{th} *time journal*. Locate n block range sets to be verified: $\mathbb{B}_1 = \{\mathrm{B}_i | i \in [1, m_1]\}$, $\mathbb{B}_2 = \{\mathrm{B}_i | i \in (m_1, m_2]\}$, $\ldots \mathbb{B}_n = \{\mathrm{B}_i | i \in (m_{(n-1)}, m_n]\}$.

3) Verify each range set of $\mathbb{B}_i$ by sequentially replaying from start to end: $\pi_i = \mathcal{V}(\mathbb{B}_i)$. Each $\mathbb{B}_i$'s temporal range is successfully audited after fully validating π_i.
4) Conduct block boundary verification across adjacent sets (e.g., verify block $m+1$ via block m): $\pi'_i = \mathcal{V}'(\mathrm{B}_i, \mathrm{B}_{i+1})$.
5) Verify the digest and signature from LSP's latest receipt ($\mathbb{O}_l$): $\Pi_3 = \mathcal{P}(\mathbb{O}_l)$.
6) The full audit is completed by computing: $\pi = \pi_1 \wedge ... \wedge \pi_n \wedge \pi'_1 \wedge ... \wedge \pi'_n \wedge \Pi_1 \wedge \Pi_2 \wedge \Pi_3$, where π is only *true* when all other proofs have successfully verified.

The above *Dasein*-complete audit process requires temporal validation and non-repudiation proofs from all participants, which should fulfill electronic data auditing requirements in real-world scenarios. In addition, this process can further take a temporal predicate to limit the scope of an audit (e.g., audit all transactions committed before `2018-12-31`).

VI. EVALUATION

We deploy LedgerDB on Alibaba Cloud and evaluate its various verification mechanisms including *Dasein*, *fam*, and *CM-Tree*. In addition, we compare LedgerDB's end-to-end performance with QLDB and Hyperledger Fabric in real-world data notarization and data lineage applications. Our experiments were conducted in an in-house cluster with two nodes, each of which runs CentOS 7.2.1511 and is equipped with Intel(R) Xeon(R) Platinum 2.5GHz CPU, 32GB RAM, and 1 TB of ESSD storage (a public storage service on Alibaba Cloud). All nodes are connected via 25Gb Ethernet.

A. Dasein Verification Breakdown

We use a workload that conducts a single audit operation on 1000 sequential *journal*s, in order to measure the breakdown cost of *Dasein* verification, i.e., existence verification (***what***), time validation (***when***), and non-repudiation proof (***who***). Figure 7 shows the verification latency of randomly picked 1000 testing sequential *journal*s by varying timestamp-related options, *journal* size, and signatures.

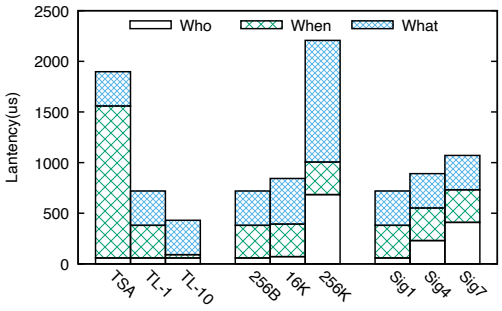
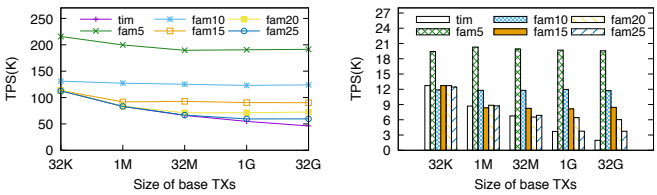


Fig. 7. Latency breakdown for *Dasein* verification factors of what-when-who.

The left three bars focus on the ***when*** factor, which depict three ***when*** scenarios for verifying credible timestamps. In all cases, each *journal* has a payload fixed to 256B and is single-signed (*Sig-1*), and the operated ledger's anchoring interval $\Delta\tau$ to TSA or *T-Ledger* is set to one second. *TSA* represents the case that the operated ledger directly interacts with TSA, which is inherently costly. *TL-1* represents the case that the operated ledger is being appended with a TPS of 1, while each *journal* is anchored to *T-Ledger*. Similarly, *TL-10* represents the case that the operated ledger increases its TPS to 10 instead. As can be seen, with the help of *T-Ledger*, the verification latency of ***when*** can be reduced by 50× in *TL-10* compared to using TSA pegging directly. The middle bars focus on the ***what*** factor, which depict the verification with different *journal* payload sizes under *TL-1* and *Sig-1* setting. The result shows that the ***who*** and ***what*** verification latencies increase significantly when payload size increases from 256B to 256KB (12× for ***who*** and 4× for ***what***). The right bars focus on the ***who*** factor, which depict the verification for multiple-signed *journal*s on *TL-1*. We vary the number of signers from 1 to 7, and the results show that the ***who*** verification latency scales linearly with the number of signatures.



(a) Append TPS comparison (b) GetProof TPS comparison

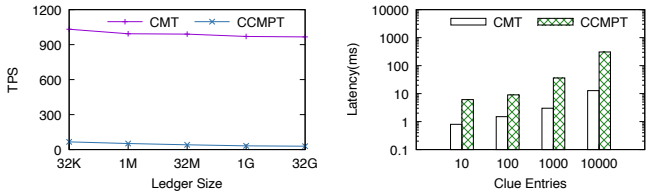
Fig. 8. Write and verification performance evaluation and comparison between *tim* and *fam*.

B. fam Tree Evaluation

Recall that the design of *fam* benefits from the advantage of both *bim* (i.e., linked block model of Bitcoin) and *tim* (i.e., accumulator tree model of Diem). We evaluate write and existence verification (i.e., `GetProof`) performance among different accumulator models with the same experimental setting above.

A *fam-n* represents a *fam* tree whose fractal height is n. We vary the fractal heights from *fam-5*, *fam-10*, *fam-15*, *fam-20* to *fam-25*, whose relative epoch threshold are 2^5, 2^{10}, 2^{15}, 2^{20}, and 2^{25} respectively. Figure 8(a) shows the throughput of `Append` operation. As can be seen, *fam-5* has more than 200K TPS and *fam-15* gets 100K TPS. We can observe that *tim*'s throughput decreases almost linearly as ledger size increases. When the ledger data volume grows to 32GB, *tim* falls to the worst among all six models. For *fam* models, their append throughput decline at first, but get stable after the data fills up at least one full epoch. Once an epoch is filled up, *fam* will setup a new epoch, so that the append cost is bounded by the fractal threshold of a *tim* tree. In particular, The throughput of *fam-5* is 4× higher than that of *tim* on average, while *fam-15* is 2× higher than *tim*.

Figure 8(b) shows the throughput of `GetProof` operation. The test is conducted on randomly generated transaction *jsn*s. As can be seen, *fam-5* and *fam-10* get relatively stable 20K and 12K TPS when the ledger size varies from 32K to 32G (i.e., all five bars in Figure 8(b)). In contrast, *fam-15*, *fam-20*, and *fam-25* only have stable verification throughput from the



(a) Verification throughput (b) Verification latency

Fig. 9. Clue verification performance.

last four, three, and two bars, respectively. This is because the volumes of an epoch in these models are different, and they only get stable performance when accumulated *journal*s reach their own thresholds. As expected, *tim*'s throughput decreases linearly when ledger data volume grows. For the 32KB-size ledger, only *fam-5* fills up its epoch with 32 leaves each. Therefore, *fam-5* performs best and others have similar throughput for around 10 layers calculation in this case. Note that in real applications, the ledger data can easily reach several Gigabytes. A commonly used *fam-15* (e.g., several Megabytes per epoch) and *fam-20* can get 8K and 6K TPS, relatively 5× and 4 × higher than that in *tim*.

C. CM-Tree Evaluation

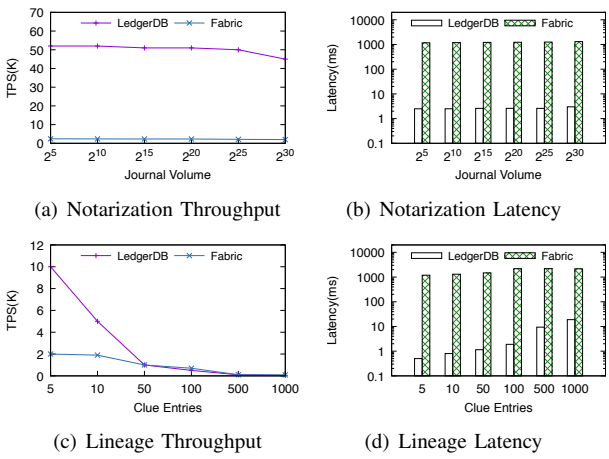
Clue is a subtle primitive to realize *N-lineage*. In this test, we generate multiple clue keys and randomly assign 1 to 100 *journal*s to each clue. The average *journal* size is 1KB. We measure both *CM-Tree* and *ccMPT* verification throughput on a randomly selected clue.

Figure 9(a) shows the verification throughput of both approaches. As can be seen, *CM-Tree* sustains a high and stable throughput at around 1000 TPS. This is because each *CM-Tree2* in clue *CM-Tree* is an independent accumulator separated from the ledger accumulator, and hence its verification cost will not increase as ledger size grows. For *ccMPT*, the existence of all m *journal*s has to be verified using its ledger accumulator. This makes its cost complexity as $O(m \times log(n))$ (where n is the total number of *journal*s), which is much higher than *CM-Tree2*'s $O(m)$. For the 32KB-size ledger, *CM-Tree* is 16× faster than *ccMPT*. When the ledger volume grows to 32GB, *CM-Tree* performs 33× faster than *ccMPT*.

Figure 9(b) shows the verification latency of both approaches on a fixed 1GB ledger accumulator. As can be seen, *CM-Tree* takes $0.8ms$ to verify a *10-entries* clue compared to $6.1ms$ in *ccMPT*, and takes $3ms$ for a *1000-entries* clue compared to $36.1ms$ in *ccMPT*. We observe that the more the number of entries in the clue is, the faster the *CM-Tree* performs. This is because the linear cost is proportional to the entry number in *ccMPT* compared to the logarithmic cost in *CM-Tree*. For the *10000-entries* case, *CM-Tree*'s latency is 24× better than that in *ccMPT*.

D. Evaluation in Applications

In this test, we evaluate application-level write and verification performance in both CLD (LedgerDB and QLDB) and blockchain (Hyperledger Fabric) systems. We choose



(a) Notarization Throughput (b) Notarization Latency

(c) Lineage Throughput (d) Lineage Latency

Fig. 10. Application-level comparison between LedgerDB and Hyperledger Fabric.

data notarization and data lineage, two typical real-world applications of both blockchain and CLD. A data notarization system stores various kinds of evidentiary records (i.e., blob proofs), each of which is identified by a unique *id* for latter retrieval and verification. A data lineage system stores and entangles relevant items under corresponding keys to track item provenance.

QLDB. Since QLDB only provides its service on public cloud [5], we deploy both applications in QLDB [40] on AWS and in LedgerDB on Alibaba Cloud. We ensure that the end-to-end deployment of each application has all its clients and servers hosted in the same region, in order to get a relatively fair comparison from a service offering perspective.

For data notarization application, a document is a [*index, data*] pair, where $data$ is a randomly generated 32KB string. To verify a document, we first retrieve the document containing the specified index, and then conduct the verification via QLDB's `GetRevision` API [41]. In QLDB, the latencies for write, read, and verify are $65ms$, $36ms$, and $1.56s$ respectively, as shown in Table II. They are 2.4×, 1.3×, 56× higher than that in LedgerDB.

For data lineage application, we design a [*key, data, prehash, sig*] schema to realize lineage logic in QLDB. The $prehash$ is a SHA-256 hash digest of the previous document, and sig is an ECDSA signature on the current document's digest. All documents have a sig signed by the same *sk*. We list two cases of a same key with different 5 and 100 versions as shown in Table II. LedgerDB is 278× and 5197 × faster than QLDB for above two cases.

Note that QLDB only offers public cloud service, and the above comparison is only for the completeness of our evaluation. It is difficult to fairly measure both systems using identical deployment environments. In addition, we omit the throughput comparison due to the transmission bandwidth limit in QLDB.

Hyperledger Fabric. We also compare verification performance in both applications between LedgerDB and Hyperledger Fabric. Hyperledger Fabric 2.2.0 is deployed in

the same setting as above. A single-channel ordering service runs a typical Kafka orderer: 3 ZooKeeper nodes, 4 Kafka brokers, 5 Fabric endorsers, and 3 orderers. Although there is no explicit data verification interface in Fabric, its implicit verification logic works when data is retrieved, i.e., in its workflow of gathering all peer signatures of consensus. Hence, we implement it within a smart contract using $GetState$ in Fabric. In LedgerDB, we implement these applications with clues.

For data notarization application, we fix each *journal* payload to 256B in both systems when evaluating their throughput (Figure 10(a)), and then measure their latencies using a real workload of 4KB payload size in average (Figure 10(b)). Figure 10(a) shows that LedgerDB's throughput decreases from 52K to 50K TPS when the *journal* volume increase from 2^5B to 2^{30}B. In Fabric, this number decreases from 2386 to 1978 TPS. Figure 10(b) shows that LedgerDB and Fabric retain their latency around $2.5ms$ and $1.2s$ respectively when *journal* volume grows. The results show that LedgerDB achieves 23× higher TPS than Fabric and gets around 500× lower latency at the same time.

For data lineage application, we test the scenario of an entire verification of a specified data key. We vary the size of *journal* entries when testing their throughput and end-to-end latency. Figure 10(c) depicts that LedgerDB reaches much higher throughput than Fabric when clue entry number is small, but converges with Fabric when the entry number exceeds 50. This is because LedgerDB performs random I/O for each entry, while Fabric has nearly a single random I/O for the entire clue. In terms of verification latency, Both LedgerDB and Fabric grow when the number of clue entries increases, as depicted in Figure 10(d). LedgerDB's latency is 300× lower than that of Hyperledger Fabric on average.

VII. RELATED WORK

Blockchain. Blockchain is the on-stage DLT that maintains decentralized immutable ledgers among mutually distrusting participants using cryptographic primitives and verifiable tree models [42]–[47]. Bitcoin [1] is the first blockchain system, whose data integrity is guaranteed by Merkle tree and mutually entangled block model (*bim*). Ethereum [2] implements MPT to store and verify states, whose address keys are mapped by original business key using a hash function. Diem [18] presents a pure *tim* based Merkle accumulator. It is a fine-grained transaction-level tree. Hyperledger Fabric [24]–[26], [48] is a widely used permissioned blockchain. Compared to permissionless blockchain whose open ecosystem brings in strong auditability, majority participants in permissioned blockchain can collude to forge timestamps and fork new chains. Corda [49] solves such auditability issue using a secure hardware (e.g., Intel SGX) as a neutral notary.

Ledger database. More and more innovations blur the boundary between blockchain and database [39], [50]. BigchainDB [51] implements Tendermint to achieve Byzantine fault tolerance on top of MongoDB [52]. BlockchainDB [53] builds database functionality using blockchain in its storage layer. The thriving of ledger databases in recent years brings in more and more blockchain-like applications back to a centralized architecture [21], [54]. QLDB [5], [20] discloses its transaction verification approach for an entire Merkle tree, which limits verification efficiency when data volume grows. SQL Ledger [19] offers forward integrity of relational data by relying on a trusted storage outside of the system. Oracle introduces a new database feature called blockchain table [8], [22] recently, as well as its corresponding SQL syntax and systematic packages. ProvenDB [27] invokes external entanglement by submitting digests to Bitcoin periodically. However, existing ledger databases also have significant space overhead, as no obsolete transactions are allowed to delete.

TABLE II
APPLICATION-LEVEL COMPARISON BETWEEN LEDGERDB AND QLDB.

Operation			Latency(s)	
			QLDB	LedgerDB
Notarization		Insert	0.065	0.027
		Retrieve	0.036	0.028
		Verify	1.557	0.028
Lineage	5-versions	Verify	7.786	0.028
	100-versions	Verify	155.9	0.030

Verifiable outsourced database. Query authentication in outsourced database (ODB) is another area of data verification [55]–[61]. It ensures data correctness of query performed by untrusted servers (e.g., delegated servers, cloud services). The correctness and completeness of result sets returned to clients can be verified using authenticated data structures (ADS) or zero-knowledge proof (ZKP) [62]. ADS [63]–[65] usually implement Merkle tree and signature-based structures to make data integrity verifiable by small verifiable objects (VO). However, ADS supports only limited query patterns, while ZKP can verify a broad range of queries with higher computation complexity. Some emerging verifiable query schemes rely on trusted hardware [66] (e.g., Intel SGX) and blockchain [67].

VIII. CONCLUSION

In this paper, we introduced ubiquitous verifiability and *Dasein*-complete auditability achieved in LedgerDB, a high-performance centralized ledger database. An external audit framework is introduced that supports *Dasein* (i.e., ***what***, ***when***, ***who***) verification of data stored on ledger. LedgerDB devises a *fam* tree based on Merkle accumulator to accelerate existence verification. It implements *T-Ledger* and a two-way timestamp pegging protocol to eliminate the chance of time-based attacks in real-world applications. In addition, it implements a two-layer *CM-Tree* to support fast *N-lineage* verification. Experimental results show that LedgerDB verification performance is significantly efficient compared to existing ledger systems, i.e., Hyperledger Fabric and QLDB.

REFERENCES

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

[2] Ethereum. Ethereum is a global, open-source platform for decentralized applications. https://www.ethereum.org, 2014.

[3] David Schwartz, Noah Youngs, Arthur Britto, et al. The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, 5(8), 2014.

[4] B Kusmierz. The first glance at the simulation of the tangle: discrete model. *IOTA Found. WhitePaper*, pages 1–10, 2017.

[5] Amazon Web Services. Amazon quantum ledger database (qldb). https://aws.amazon.com/qldb, 2018.

[6] Gartner. Top 10 trends in data and analytics for 2020. https://www.gartner.com/smarterwithgartner/gartner-top-10-trends-in-data-and-analytics-for-2020/, 2020.

[7] 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. *Proceedings of the VLDB Endowment*, 13(12):3138–3151, 2020.

[8] Oracle. Oracle blockchain blog. https://blogs.oracle.com/blockchain/, 2019.

[9] Alibaba Cloud LedgerDB. A ledger database that provides powerful data audit capabilities. https://www.alibabacloud.com/product/ledgerdb, 2019.

[10] Wikipedia. Dasein. https://www.wikipedia.org/wiki/Dasein, 2020.

[11] Government of Beijing Municipality. Beijing internet court. https://english.bjinternetcourt.gov.cn/, 2019.

[12] Government of Hangzhou Municipality. Hangzhou court of the internet. https://www.netcourt.gov.cn/?lang=En, 2019.

[13] Government of Guangzhou Municipality. Guangzhou court of the internet. https://en.gzinternetcourt.gov.cn/en/index.html, 2019.

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

[15] Chou Tai Fook. Chow tai fook jewellery group. https://www.chowtaifook.com/en/, 2021.

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

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

[18] Diem Association. Diem blockchain. https://www.diem.com/en-us/, 2020.

[19] Panagiotis Antonopoulos, Raghav Kaushik, Hanuma Kodavalla, Sergio Rosales Aceves, Reilly Wong, Jason Anderson, and Jakub Szymaszek. SQL ledger: Cryptographically verifiable data in azure SQL database. In *Proceedings of the 2021 ACM SIGMOD international conference on Management of data*, pages 2437–2449, 2021.

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

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

[22] Oracle. Oracle blockchain table. https://docs.oracle.com/en/database/oracle/oracle-database/20/newft/oracle-blockchain-table.html, 2020.

[23] Oracle. Blockchain tables in oracle database: Technology convergence. https://blogs.oracle.com/blockchain/blockchain-tables-in-oracle-database:-technology-convergence, 2021.

[24] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

[25] Hyperledger. The linux foundation. https://www.hyperledger.org/, 2019.

[26] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. Performance benchmarking and optimizing hyperledger fabric blockchain platform. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 264–276. IEEE, 2018.

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

[28] Mihir Bellare and Bennet Yee. Forward integrity for secure audit logs. Technical report, Citeseer, 1997.

[29] Narongrit 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.

[30] Paul Snow, Brian Deery, Jack Lu, David Johnston, and Peter Kirby. Factom: Business processes secured by immutable audit trails on the blockchain. *Whitepaper, Factom, November*, 2014.

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

[32] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. Fine-grained, secure and efficient data provenance on blockchain systems. *Proceedings of the VLDB Endowment*, 12(9):975–988, 2019.

[33] National Time Service Center. https://ttas.ntsc.ac.cn/, 2020.

[34] Xi'an Trusted Time Authentication Service. https://www.chinattas.com/, 2020.

[35] Ethereum Foundation. Shrubs - a new gas efficient privacy protocol. https://www.youtube.com/watch?v=_tqwCBrw1Xc, 2019.

[36] Yize Li, Benquan Yu, Xinying Yang, Wenyuan Yan, and Yuan Zhang. Managing blockchain-based centralized ledger systems, January 1 2021. US Patent 10,904,013.

[37] Alibaba Elastic Cloud Service. Elastic and secure virtual cloud servers to cater all your cloud hosting needs. https://www.alibabacloud.com/product/ecs, 2019.

[38] Xueping Liang, Sachin Shetty, Deepak Tosh, Charles Kamhoua, Kevin Kwiat, and Laurent Njilla. Provchain: A blockchain-based data provenance architecture in cloud environment with enhanced privacy and availability. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 468–477. IEEE, 2017.

[39] Lindsey Allen, Panagiotis Antonopoulos, Arvind Arasu, Johannes Gehrke, Joachim Hammer, James Hunter, Raghav Kaushik, Donald Kossmann, Jonathan Lee, Ravi Ramamurthy, et al. Veritas: Shared verifiable databases and tables in the cloud. In *9th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[40] Amazon. Sql-compatible access to relational, semi-structured, and nested data. https://partiql.org, 2016.

[41] Amazon Web Services. Amazon quantum ledger database (amazon qldb) - developer guide. https://docs.aws.amazon.com/qldb/latest/developerguide/API_GetRevision.html, 2020.

[42] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Caper: a cross-application permissioned blockchain. *Proceedings of the VLDB Endowment*, 12(11):1385–1398, 2019.

[43] 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.

[44] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. On scaling decentralized blockchains. In *International conference on financial cryptography and data security*, pages 106–125. Springer, 2016.

[45] Sheng Wang, Tien Tuan Anh Dinh, Qian Lin, Zhongle Xie, Meihui Zhang, Qingchao Cai, Gang Chen, Beng Chin Ooi, and Pingcheng Ruan. Forkbase: An efficient storage engine for blockchain and forkable applications. *Proceedings of the VLDB Endowment*, 11(10), 2018.

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

[47] Amazon Web Services. Amazon managed blockchain. https://aws.amazon.com/blockchain/, 2018.

[48] Oracle. Oracle blockchain enterprise edition. https://www.oracle.com/blockchain/blockchain-platform-enterprise-edition/, 2021.

[49] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: an introduction. *R3 CEV, August*, 1:15, 2016.

[50] Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, and Jens 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.

[51] Trent McConaghy, Rodolphe Marques, Andreas Müller, Dimitri De Jonghe, Troy McConaghy, Greg McMullen, Ryan Henderson, Sylvain Bellemare, and Alberto Granzotto. Bigchaindb: a scalable blockchain database. *white paper, BigChainDB*, 2016.

[52] MongoDB. The database for modern applications. https://www.mongodb.com/, 2016.

[53] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. Blockchaindb: a shared database on blockchains. *Proceedings of the VLDB Endowment*, 12(11):1597–1609, 2019.

[54] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. Blockchain meets database: design and

implementation of a blockchain relational database. *Proceedings of the VLDB Endowment*, 12(11):1539–1552, 2019.

[55] Michael Backes, Dario Fiore, and Raphael M Reischuk. Verifiable delegation of computation on outsourced data. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 863–874, 2013.

[56] Sumeet Bajaj and Radu Sion. Correctdb: Sql engine with practical query authentication. *Proceedings of the VLDB Endowment*, 6(7):529–540, 2013.

[57] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. Concerto: A high concurrency key-value store with integrity. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 251–266, 2017.

[58] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 121–132, 2006.

[59] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Authenticated index structures for aggregation queries. *ACM Transactions on Information and System Security (TISSEC)*, 13(4):1–35, 2010.

[60] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. Integridb: Verifiable sql for outsourced databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1480–1491, 2015.

[61] Cheng Xu, Ce Zhang, and Jianliang 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.

[62] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. vsql: Verifying arbitrary sql queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 863–880. IEEE, 2017.

[63] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. *ACM SIGPLAN Notices*, 49(1):411–423, 2014.

[64] Roberto Tamassia. Authenticated data structures. In *European symposium on algorithms*, pages 2–5. Springer, 2003.

[65] Charles Martel, Glen Nuckolls, Premkumar Devanbu, Michael Gertz, April Kwong, and Stuart G Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.

[66] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 264–278. IEEE, 2018.

[67] Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song. Falcondb: Blockchain-based collaborative database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 637–652, 2020.