TECHNICAL PAPER — K³LABS RESEARCH

Algebraic Accounting: A Vector-Algebraic Framework for Immutable, Deterministic, and Branchable Financial Ledgers

Rafi Seddiqi

K³Labs • k3labs.ai

Working paper • Version 1.0 • April 2026

ABSTRACT

We present *algebraic accounting*: a formal framework in which a financial ledger is modelled as a state vector in $\mathbb{R}^n$, transactions are modelled as delta vectors, and the evolution of the ledger is expressed as the composition of vector-addition operations drawn from a finite set of typed transformation rules. The framework unifies three strands of prior work: (i) the algebraic formalisation of double-entry bookkeeping originally proposed by Pacioli and later formalised by Ellerman, Mattessich, and Ijiri [1][2][3][4]; (ii) event sourcing and content-addressable immutable storage [7][10][11]; and (iii) the data-version-control systems that apply distributed-version-control semantics (branching, merging, diffing) to structured data [15][16][17]. We specify the algebraic structure (an abelian group of balance vectors under addition with an invariant-preserving subspace), describe immutable source-data capture via SHA-256 content hashing, and formalise ephemeral book management as operations on equivalence classes of ledgers indexed by a directed acyclic graph of commits. We prove determinism, reconstructability, and invariant preservation for the core operations, and we outline a reference implementation using DuckDB and FastAPI. We argue that the combination addresses three long-standing weaknesses of conventional accounting systems — mutable records, ad-hoc reconciliation, and the absence of first-class support for counter-factual analysis — without departing from the

double-entry invariants mandated by generally-accepted accounting principles and regulatory frameworks such as SEC Rule 17a-4 [21].

Keywords: double-entry bookkeeping, algebraic accounting, event sourcing, immutable ledger, content-addressable storage, data version control, REA, triple-entry accounting, formal methods in accounting.

1. Introduction

Double-entry bookkeeping, first codified in the late fifteenth century by Luca Pacioli [1], remains the foundation of virtually all modern financial record-keeping. Its central invariant — that every transaction produces equal and opposite effects across the accounting equation (Assets = Liabilities + Equity) — has survived five centuries essentially unchanged. Yet the systems in which double-entry is implemented have evolved piecemeal: from manual ledgers, through punch-card mainframes, to contemporary relational-database-backed ERP suites. Many of the data-management characteristics of those systems reflect accidents of implementation rather than the mathematical structure of double-entry itself.

Three such characteristics are particularly consequential. First, conventional ledgers are *mutable*: correcting an error typically involves writing a compensating journal entry rather than preserving the original and the correction together in a cryptographically verifiable form. Second, reconciliation across periods, branches, or entities is largely *procedural*, implemented as ad-hoc scripts that reconstruct balances from transaction logs. Third, there is no first-class notion of a *counterfactual* or *experimental* book — alternative accounting treatments, simulation of proposed transactions, or "what-if" analysis at scale must be reconstructed manually, and the relationship between the counterfactual and the production ledger is rarely captured formally.

The converging influence of three modern engineering traditions suggests these characteristics are no longer necessary. The first tradition is the formalisation of double-entry as an algebraic system: Ellerman [2][2a] showed that T-accounts can be understood as the Pareto group of differences over a commutative monoid of quantities, and Mattessich [3] and Ijiri [4] laid axiomatic foundations in which ledger operations are linear-algebraic. The second is event sourcing, articulated by Fowler [7] and Young [8] and operationalised at scale by Kleppmann [13], which replaces mutable state with an append-only log of events from which state is reconstructed by fold. The third is the application of distributed-version-control semantics to structured data, as realised in Dolt [15], TerminusDB [16], and Pachyderm [17], each of which implements branching, merging, and diffing over immutable content-addressed snapshots.

This paper proposes a unification we call *algebraic accounting*. The ledger is a vector in $\mathbb{R}^n$ where each dimension corresponds to a recognised account. Transactions are delta vectors.

Posting a transaction is vector addition. The accounting invariants (e.g. $\Sigma_{\text{assets}} - \Sigma_{\text{liabilities}} - \Sigma_{\text{equity}} = 0$) are linear constraints; posting rules are parameterised linear maps; and the full history of a book is a sequence of delta vectors whose sum is the current state. Source documents are hashed and stored immutably in a content-addressable store, binding every delta vector to its originating evidence. Books may be forked at any commit, mutated independently, and merged through a vector-reconciliation operation that respects invariants. Determinism, auditability, and counterfactual analysis fall out of the mathematics rather than being bolted on.

Contributions. The contributions of this paper are: (1) a precise algebraic specification of a ledger as an element of a commutative group on which transformation rules act as linear maps preserving an invariant subspace (§3); (2) a construction of the immutable source-data layer that binds every delta vector to a SHA-256 hash of its supporting document and exposes a Merkle-tree integrity proof (§4); (3) a specification of ephemeral book management as operations on a commit-DAG of ledger states, with formally defined branch, merge, and release operations (§5); (4) proofs (or proof sketches, where appropriate) of determinism, reconstructability, and invariant preservation (§6); (5) a worked example demonstrating a complete commit-branch-merge cycle on a small book (§7); and (6) a discussion of the relationship of the framework to existing accounting standards, to triple-entry and blockchain-based proposals, and to the data-version-control literature (§9–§10).

2. Background and Related Work

2.1 Algebraic formalisations of double-entry bookkeeping

Pacioli's *Summa* of 1494 [1] is the oldest surviving printed description of double-entry, but Pacioli offered a procedural rather than a mathematical account of the technique. The formal mathematical structure of double-entry was first set out by Ellerman in two papers, "The Mathematics of Double Entry Bookkeeping" (1985) and "Double Entry Multidimensional Accounting" (1986) [2][2a]. Ellerman showed that a T-account is isomorphic to an element of the group of differences over the commutative monoid of non-negative quantities: every debit–credit pair (d,c) is identified with the equivalence class $[(d,c)]$ under $(d_1,c_1) \sim (d_2,c_2)$ iff $d_1 + c_2 = d_2 + c_1$, precisely the Grothendieck construction that produces $\mathbb{Z}$ from $\mathbb{N}$. The group is abelian, and transactions compose by addition. This observation makes the algebraic content of double-entry explicit.

Mattessich's *Accounting and Analytical Methods* (1964) [3] predated Ellerman and argued that accounting should be founded on an axiomatic system, with matrix algebra serving as the natural notation for ledger operations. Mattessich's matrix formulation — where a transaction is a matrix T with $+1$ in the debited account and -1 in the credited account — is essentially equivalent to Ellerman's vector formulation for single-leg transactions; the vector representation generalises it to multi-leg postings. Ijiri's *Foundations of Accounting Measurement* (1967) [4]

developed an independent but complementary mathematical foundation and is frequently cited as introducing the idea of "triple-entry" (an additional axis capturing causal or agential information), a theme taken up much later in Grigg's work on cryptographically-signed triple-entry [18]. More recent contributions by Balzer and Mattessich [5] and Mir and Khan [6] develop the matrix-algebraic treatment further.

Notwithstanding this substantial prior work, the phrase *algebraic accounting* is not yet an established term of art. The closest established labels are *matrix accounting*, *axiomatic accounting*, and *computational accounting*. The present paper adopts *algebraic accounting* because it emphasises the group-theoretic closure and invariant properties that underpin the framework, while signalling the intent to unify the historic algebraic tradition with modern engineering practice.

2.2 Event sourcing and immutable ledgers

Fowler's 2005 note on Event Sourcing [7] articulated the architectural pattern in which application state is derived from an append-only log of domain events, rather than being maintained as a mutable aggregate. Young [8] extended the pattern to CQRS (Command/Query Responsibility Segregation), enabling the reconstruction of arbitrary views of state by replaying events through different projections. Kleppmann [13] and the "turning the database inside-out" argument [14] propose that the immutable event log should be treated as the primary system of record, and that derived stores are best understood as materialised projections.

McCarthy's REA (Resources, Events, Agents) model, published in *The Accounting Review* in 1982 [9], is an early accounting-specific precursor to event-sourcing thinking: McCarthy argued that accounting systems should represent business activity as a graph of economic events involving resources and agents, rather than as aggregated account balances. The algebraic-accounting framework proposed here treats McCarthy's events as the source of delta vectors and the resulting balance vector as a projection — a view Kleppmann would recognise as a materialised fold over the event log.

2.3 Cryptographic integrity and content-addressable storage

Merkle's 1987 paper introduced the Merkle tree as a structure for compact integrity proofs over large ordered data [10], and Haber and Stornetta's 1991 paper on time-stamping digital documents [11] established that a hash-chain over sequenced documents yields a tamper-evident record without trusted third parties. Nakamoto's 2008 Bitcoin whitepaper [12] popularised these techniques by combining them with proof-of-work to produce a decentralised immutable ledger. For our purposes, decentralisation is not required — the algebraic-accounting framework assumes a single trusted entity maintains the ledger — but the integrity techniques (content hashing, Merkle proofs, hash-chains) transfer directly and provide strong guarantees of tamper-evidence at a cost far lower than blockchain consensus.

2.4 Distributed version control for data

Recent data-infrastructure projects have applied Git-like semantics to structured data. Dolt [15] implements branchable, mergeable, diffable SQL tables using a variant of the Prolly tree, a probabilistic balanced Merkle structure; TerminusDB [16] applies similar semantics to JSON and RDF graphs via immutable storage and succinct data structures; Pachyderm [17] provides Git-like lineage tracking for containerised data pipelines. The algebraic-accounting framework adopts the commit-DAG abstraction from these systems and adapts it to ledger states, where the semantics of *merge* are defined by vector reconciliation rather than by textual or row-level three-way merge.

2.5 Triple-entry and blockchain-based accounting

Grigg's 2005 proposal of triple-entry accounting [18] introduced the idea of a digitally-signed receipt that produces three mutually-consistent ledger entries — one on each counterparty's books and one on a shared receipt register — as a defence against unilateral restatement. The proposal has been revisited extensively in the blockchain-accounting literature; Dai and Vasarhelyi [19] survey the potential for blockchain to support continuous assurance, while Coyne and McMickle [20] provide a critical appraisal arguing that most blockchain implementations fail to solve the underlying verification problem. The algebraic-accounting framework is orthogonal to the question of decentralisation: it is compatible with a triple-entry arrangement in which counterparty-signed deltas are posted to both books' commit-DAGs, but does not require it.

2.6 Regulatory context: WORM and electronic records

SEC Rule 17a-4 [21], as amended in 2022, requires broker-dealers to retain certain records in a write-once-read-many (WORM) format or, alternatively, to maintain an audit-trail system that preserves the complete history of modifications. Similar requirements appear in FINRA 4511 and CFTC 1.31 [22]. The content-addressable immutable store described in §4 satisfies both the WORM and audit-trail variants of the requirement: source documents are written exactly once, and modifications to the ledger take the form of new delta vectors whose hashes are appended to the commit-DAG.

3. The Formal Model

3.1 The ledger as a vector

> **Definition 3.1 (Ledger state).** Let $A = \{a_1, \ldots, a_n\}$ be a finite set of accounts fixed by the chart of accounts. A *ledger state* is a vector $L \in \mathbb{R}^n$ in which the i-th component $L[a_i]$ is the current balance of account a_i, expressed in the unit of the book's reporting currency. Debit-normal accounts are signed positively; credit-normal accounts are signed negatively, so that the accounting equation takes the form of a linear constraint $\Sigma_i\, L[a_i] = 0$ across all accounts whose signs are consistent with the equation.

The sign convention above follows the Ellerman formulation [2]. It is conventional to partition A into the five top-level categories of the accounting equation — assets, liabilities, equity, revenue, expenses — and to extend the chart as necessary (sub-accounts, dimensions, cost centres). Dimensions can be handled by tensor rather than vector representations; we restrict attention here to the vector case.

3.2 Transactions as delta vectors

> **Definition 3.2 (Delta vector).** A *transaction* is represented as a delta vector $\Delta T \in \mathbb{R}^n$. The *legs* of the transaction are the non-zero components of ΔT: each non-zero entry $\Delta T[a_i]$ represents a debit (positive) or credit (negative) of the magnitude $|\Delta T[a_i]|$ to account a_i. A delta vector is *balanced* iff it satisfies the signed linear constraint $\Sigma_i\, \Delta T[a_i] = 0$.

Balanced delta vectors form a linear subspace $B \subset \mathbb{R}^n$ of codimension one: all valid postings lie in B. The posting operation is then simply vector addition:

$$L_{k+1} = L_k + \Delta T_{k+1} \qquad \text{where } \Delta T_{k+1} \in B$$

Because $\mathbb{R}^n$ is an abelian group under addition and B is closed under addition, the set of ledger states reachable from a starting state L_0 by applying a sequence of balanced deltas forms an affine subspace through L_0 parallel to B. This is the sense in which algebraic accounting is "closed under invariants": the accounting equation is preserved by construction, not by the runtime enforcement of a check.

3.3 Transformation rules and typed postings

In practice, raw deltas are not entered directly by users; instead, users describe a *business event* (a sale, a cash receipt, a payroll accrual), and the system translates that event into the appropriate delta according to a *posting rule*. We model a posting rule as a parameterised linear map.

> **Definition 3.3 (Posting rule).** A *posting rule* R for an event type e is a function $R: \mathbb{R}^p \to B$ that sends a parameter vector x (the concrete quantities associated with the event — amount, tax rate, cost-of-goods, etc.) to a balanced delta vector. When R is linear, the set of postings it generates is a linear subspace of B; when R is piecewise linear (e.g. if tax brackets apply), the generated set is a union of such subspaces.

The set of posting rules is versioned along with the chart of accounts: changes to the chart or to the rules themselves constitute new commits in the commit-DAG of §5. This is important because it allows the system to replay historic events against historic rules — a property required, for example, for re-statements that must respect the rules in force at the time of the original transaction.

3.4 Projections, views, and derived balances

Many reports (trial balances, sub-ledgers, cost-centre rollups, consolidations) are linear projections of the ledger vector. Let $P \in \mathbb{R}^{m \times n}$ be a projection matrix; the view is $V = PL$. Consolidation across entities, for example, is a projection that sums corresponding accounts across a block-diagonal multi-entity ledger. Because projections commute with addition, the view at commit k is $PL_k = P(L_0 + \Sigma_{j \le k}\, \Delta T_j) = PL_0 + \Sigma_{j \le k}\, P\Delta T_j$, so views can be maintained incrementally and still be guaranteed to agree with a full recomputation.

4. The Immutable Source-Data Layer

4.1 Content addressing of evidence

Every delta vector in the ledger is bound to a *source document*: an invoice, a bank statement line, a signed contract, a counterparty-signed triple-entry receipt. The source document is the evidence for the posting and must be retained in a form that allows independent reverification. We require the document to be stored in a content-addressable store: the document's address is the SHA-256 hash of its canonical byte representation [10]. A delta vector is then represented as a tuple (e, x, R, h), where e is the event type, x the parameter vector, R the posting rule, and h the content hash of the source document. The concrete delta $\Delta T = R(x)$ is derived; it is not stored directly. This ensures that every delta is reconstructable from the evidence and the rule set in force at its commit.

4.2 Hash-chains and Merkle proofs

To make tampering detectable, we sequence the committed deltas into a hash chain in the manner of Haber and Stornetta [11]. The commit hash at position k is defined as

$$c_k = \mathrm{SHA\text{-}256}(c_{k-1} \,\|\, h_k \,\|\, \mathrm{meta}_k)$$

where c_0 is a designated genesis hash, h_k is the content hash of the source document, and meta_k includes event type, parameters, rule identifier, timestamp, and author. A Merkle tree over the c_k values provides compact inclusion proofs for audit. Unlike in a public blockchain, there is no proof-of-work requirement: the ledger maintainer signs the current head of the chain, and signed heads may be lodged with an external notarisation service if required for regulatory purposes.

4.3 WORM compatibility and regulatory defensibility

The content-addressable store is trivially WORM-compliant in the sense of SEC 17a-4(f)(2) [21]: writing the same document a second time is idempotent (same hash, same address), and deletion is not exposed at the storage interface. Modifications to the ledger are represented as new deltas — typically a compensating delta plus a corrected delta, each with its own source document — rather than as mutations of existing records. The full history of corrections is thus preserved and cryptographically bound. This satisfies the audit-trail alternative under Rule 17a-4(f)(2)(i)(A)(2) and, via external notarisation of signed heads, the WORM alternative under (f)(2)(i)(A)(1) [22].

5. Ephemeral Book Management

5.1 The commit-DAG

Conceptually, a *book* is a directed acyclic graph of commits. Each commit k is a tuple $(c_k, c_{\mathrm{parent}(k)}, \Delta T_k, \mathrm{meta}_k)$ where c_k is the commit hash as defined in Equation (2), $c_{\mathrm{parent}(k)}$ is the hash of its parent (or the set of hashes of its parents, in the case of a merge commit), and the remaining fields are as above. The state of the book at any commit is computed by folding vector addition along the path from the genesis commit:

$$L(c_k) = L_0 + \Sigma_{j \in \mathrm{path}(c_k)} \Delta T_j$$

Branches are labelled references to commit hashes, equivalent to branches in Git [15]. A *release* is an immutable tagged reference, typically used to identify period-end closes. The head of the production branch is the canonical ledger at any point in time.

5.2 Branching

> **Definition 5.1 (Branch).** Given a book B with head c_{h}, a *branch from* c_h is a new labelled reference whose initial head is c_{h} and to which new commits may be appended. The branch shares all ancestor commits with the original; only the commits appended on the branch are unique to it.

Branches are typically created for three purposes: (i) to prepare a proposed posting for review before promotion to production; (ii) to prepare a "what-if" analysis, such as the effect of an accounting policy change, without disturbing the production ledger; and (iii) to isolate a period-end close from ongoing activity during the close process. Because branches share history, they cost O(1) in storage.

5.3 Merging

The semantics of merge are where algebraic accounting diverges most visibly from Git-like version control. In a textual VCS, merge is three-way text reconciliation; in a ledger, merge is vector reconciliation. Given two branches a and b with common ancestor o, define

$$\Delta a = L(a) - L(o) \quad \Delta b = L(b) - L(o)$$

Both Δa and Δb lie in B by closure under addition. The merged state is

$$L(\text{merge}(a, b)) = L(o) + \Delta a + \Delta b = L(a) + \Delta b = L(b) + \Delta a$$

which is symmetric by abelianness. *Merge conflicts* arise when the same source document (same content hash) has been incorporated on both branches with divergent parameters or rules, or when the two branches have incompatible chart-of-accounts changes. In the former case the conflict is resolved by deduplication (the delta is applied once); in the latter, the conflict must be resolved by a human before the merge commit is written. We note that the absence of cell-level conflicts — which is the dominant source of pain in textual three-way merges — is a direct consequence of the algebraic structure: additions commute.

5.4 Releases and immutability of closed periods

A *release* is a signed tag pointing to a specific commit, typically representing a period close (month-end, quarter-end, year-end). Once a release is signed by the appropriate officer, the

commit is immutable in the audit sense: any subsequent correction must be posted as a new delta on a successor commit, not as a mutation of the released state. This implements the standard accounting practice of keeping closed periods untouched while retaining full restatement ability.

6. Formal Properties

6.1 Determinism

> **Proposition 6.1 (Determinism).** Given the genesis state L_0, the ordered sequence of commits (c_1, …, c_k), and the rule-set versions referenced by each commit's metadata, the state $L(c_k)$ is uniquely determined. Two independent reconstructions over the same inputs yield identical vectors.

Proof sketch. Each commit's delta ΔT_j is derived as $R_j(x_j)$ from the parameters x_j stored in the commit and the posting rule R_j identified by the rule-version reference. Vector addition is associative and commutative. The fold over the ordered sequence therefore produces the same result regardless of grouping. Because the rule-set version is recorded and the rules themselves are immutable artefacts in the content-addressable store, every historic delta is reproducible. ∎

6.2 Reconstructability from immutable sources

> **Proposition 6.2 (Reconstructability).** For any commit c_k, the state $L(c_k)$ can be reconstructed from the content-addressable store of source documents, the rule-set store, and the commit metadata alone, without reference to any precomputed balance.

This is the *event sourcing* property expressed in the algebraic setting; it is a direct corollary of Proposition 6.1 together with the observation that every commit's data is stored by content hash in an append-only store, not as a mutation of derived state. This is the property that distinguishes the framework from conventional ledgers, in which the current balance is the primary datum and the transaction log is a secondary artefact.

6.3 Invariant preservation

> **Proposition 6.3 (Invariant preservation).** If L_0 satisfies the accounting equation and every delta ΔT_j lies in the balanced subspace B, then every reachable state $L(c_k)$ satisfies the accounting equation.

Proof. Let e denote the linear functional implementing the signed sum that constitutes the accounting equation; then $e(L_0) = 0$ and $e(\Delta T_j) = 0$ for all j. By linearity, $e(L(c_k)) = e(L_0) + \Sigma_j e(\Delta T_j) = 0$. ∎

This closure under invariants is the mathematical content of "accounts always balance": it is not an obligation imposed on the user or a runtime check, it is a property of the algebra.

6.4 Merge correctness

> **Theorem 6.4 (Merge symmetry and invariant preservation).** The merge operation defined in §5.3 is symmetric in its arguments and, provided both input branches preserve the accounting equation, produces a merged state that also preserves the accounting equation.

Proof sketch. Symmetry follows from the commutativity of vector addition. Invariant preservation follows from Proposition 6.3 applied to the concatenated sequence of deltas from both branches. ∎

7. Worked Example

Consider a minimal chart of six accounts: Cash, Accounts Receivable, Inventory, Revenue, COGS, Equity. The ledger state vector is L = [Cash, AR, Inventory, Revenue, COGS, Equity]. The genesis state is L_0 = [0, 0, 0, 0, 0, 0].

Commit c_1: opening capital contribution of $1,000. The posting rule for "capital contribution" with parameter $x = 1000$ produces ΔT_1 = [+1000, 0, 0, 0, 0, −1000]. The state after commit is $L(c_1)$ = [1000, 0, 0, 0, 0, −1000]; the accounting equation sums to zero.

Commit c_2: inventory purchase of $400 on credit. The rule for "credit purchase of inventory" produces ΔT_2 = [0, 0, +400, 0, 0, 0] on the asset side and a corresponding credit to Accounts Payable; in this minimal chart without an AP account, we extend to seven dimensions or fold AP into Equity with a negative sign for illustration. Using the extended chart ΔT_2 = [0, 0, +400, 0, 0, 0, −400]. The state after commit c_2 is [1000, 0, 400, 0, 0, −1000, −400].

Commit c_3: cash sale of $100 with inventory cost $60. The posting rule for "cash sale with cost of goods" is multi-leg:

```
ΔT₃ = [+100, 0, −60, −100, +60, 0, 0]
```

Net effect: Cash +100, Inventory −60, Revenue −100 (credit-normal), COGS +60. Sum = 0: invariant preserved. The state after commit c_3 is [1100, 0, 340, −100, 60, −1000, −400].

Branching. Suppose at this point we want to model the effect of writing down the remaining inventory by $50, without committing to the write-down on the production branch. We branch from c_3 as branch `scenario-writedown` and commit c_4^{scenario} with $\Delta T_4^{\text{scenario}}$ = [0, 0, −50, 0, +50, 0, 0] (impairment to COGS). On the scenario branch the state becomes [1100, 0, 290, −100, 110, −1000, −400].

Merge. Meanwhile, on the production branch, suppose a $200 cash customer payment on account is received (c_4^{prod}, ΔT_4^{prod} = [+200, −200, 0, 0, 0, 0, 0]). The production state becomes [1300, −200, 340, −100, 60, −1000, −400]. If we then decide to adopt the scenario's write-down, we merge the scenario branch into production at common ancestor c_3: $\Delta a = \Delta T_4^{\text{prod}}$, $\Delta b = \Delta T_4^{\text{scenario}}$, and the merged state is $L(c_3) + \Delta a + \Delta b$ = [1300, −200, 290, −100, 110, −1000, −400]. The signed sum is zero: the invariant is preserved through the merge.

Every step above is determined by the ordered sequence of deltas, which in turn is determined by the content hashes of the source documents and the rule-set identifiers in the commit metadata. An auditor reconstructing the book from the raw stores would arrive at identical vectors.

8. Reference Implementation

We outline a reference implementation in Python. The ledger vector is represented sparsely, because charts of accounts are typically thousands of accounts wide but individual deltas touch only a handful. Persistence is provided by DuckDB [23], which supports both the append-only commit log and the materialised balance view; the API is exposed via FastAPI.

```
@dataclass(frozen=True)
class Commit:
    parent: str                     # hash of parent commit
    source_hash: str                # SHA-256 of source document
    event_type: str                 # e.g. "cash_sale_with_cogs"
    params: dict                    # parameters of the event
    rule_version: str               # hash of the rule-set
    timestamp: str
    author: str

    def hash(self) -> str:
        payload = canonical_json(asdict(self))
        return sha256(payload).hexdigest()

def post(ledger: SparseVector, delta: SparseVector) -> SparseVector:
    new = ledger.copy()
    for account, amount in delta.items():
        new[account] = new.get(account, 0) + amount
    assert abs(sum(new.values())) < TOLERANCE, "invariant violated"
```

```
    return new

def reconstruct(commits: list[Commit], rules: RuleStore) -> SparseVector:
    state = SparseVector()
    for c in commits:
        rule = rules.get(c.event_type, c.rule_version)
        delta = rule(c.params)
        state = post(state, delta)
    return state
```

The sparse-vector representation makes the cost of posting O(legs per transaction) rather than O(n) in the chart size, and the commit log is append-only, so the naive reconstruction is linear in the number of commits; in practice we maintain a checkpointed materialised balance at each release and reconstruct only the tail since the last release.

9. Discussion

9.1 What algebraic accounting is and is not

Algebraic accounting is not a new accounting standard. It does not change what is recorded or how transactions are recognised under GAAP or IFRS. It is a *system of record* architecture that implements the existing accounting model more faithfully to its mathematical structure. Conventional ERP ledgers implement double-entry, but their mutability, procedural reconciliation, and absence of first-class branching are characteristic of database engineering rather than of the accounting model itself.

9.2 Relationship to triple-entry and blockchain proposals

The framework is compatible with Grigg-style triple-entry [18]: a counterparty-signed receipt becomes the source document for both parties, and its content hash appears in both books' commit-DAGs. It is also compatible with, but does not require, a distributed-ledger implementation of the commit-DAG. The critical primitives — content hashing, hash-chaining, immutable storage, and the algebraic ledger itself — are all implementable in a single-entity centralised system, which is typically what audit and regulatory frameworks assume. The use of distributed consensus is orthogonal and has well-documented costs [20].

9.3 Limitations

Several limitations deserve explicit acknowledgement. First, the $\mathbb{R}^n$ representation elides multi-currency and multi-unit bookkeeping; a complete treatment requires a tensor representation with currency and unit indices, and revaluation then becomes a linear map on that tensor. Second, non-linear rules (tax brackets, foreign-exchange revaluation at the commit time) complicate the linear-algebraic story: the rules become piecewise linear, and the view that "a book

is a point in an affine subspace" becomes a union of such subspaces. Third, the framework is silent on *who* may post what — access control, separation of duties, and approval workflows are orthogonal concerns that must be layered on top. Fourth, compliance with specific jurisdictional requirements beyond SEC 17a-4 has not been analysed here; practitioners should assess each requirement against the specific architecture deployed.

9.4 Comparison to prior approaches

Property	Conventional ERP	Event-sourced ledger	Blockchain ledger	Algebraic accounting (this paper)
Mutable records	Yes	No	No	No
Deterministic reconstruction	Partial	Yes	Yes	Yes (Prop. 6.1–6.2)
Cryptographic integrity	No	Optional	Yes (consensus)	Yes (hash-chain)
First-class branching	No	No	Hard	Yes (§5)
Invariant by construction	No (runtime check)	No (runtime check)	Depends on contract	Yes (Prop. 6.3)
Decentralisation required	No	No	Yes	No
SEC 17a-4 compatibility	Requires overlay	Yes	Yes	Yes

10. Related Work and Extensions

Beyond the primary sources surveyed in §2, the framework has parallels in several other lines of work. The *functional reactive* view of accounting — in which balances are continuously-updated projections of an event stream — aligns with Kleppmann's argument for "turning the database inside-out" [14]. The use of a commit-DAG on structured data has precedents in Noms (now Dolt), Datomic, and TerminusDB [15][16]. The insight that accounting invariants are best enforced by construction rather than by runtime checks has analogues in dependent-type systems; a natural extension of this work is to encode the posting rules in a type system that

refuses to type-check non-balancing rules, so that the invariant is enforced at development time rather than at posting time.

Several extensions are natural. (i) A *tensor* generalisation to handle multi-currency, multi-entity, multi-period books — here the state becomes an element of $\mathbb{R}^n \otimes \mathbb{R}^c \otimes \mathbb{R}^e$ and revaluation is a linear map on the currency tensor factor. (ii) A *zero-knowledge* extension, in which a counterparty can prove the content of its ledger without revealing it, using techniques developed in the blockchain-accounting literature [19]. (iii) A *continuous-assurance* layer that re-verifies invariants across the commit-DAG in near-real time, providing the audit workflow that conventional year-end audits approximate. (iv) Integration of machine-learning models as posting rules for predictive entries (accruals, allowances) — a direction that preserves determinism if the model and its parameters are themselves content-addressed.

11. Conclusion

We have proposed *algebraic accounting* as a framework that unifies three mature engineering traditions — the algebraic formalisation of double-entry, event sourcing with cryptographic integrity, and distributed-version-control semantics for data — in service of a mathematically rigorous, deterministic, and branchable financial ledger. The framework preserves the invariants of conventional accounting by construction rather than by runtime check, makes every state reconstructable from its sources, and enables first-class counterfactual analysis via branches and merges whose semantics are those of vector reconciliation. The formal properties of determinism, reconstructability, and invariant preservation follow from elementary properties of the vector-addition group. The framework is compatible with existing regulatory requirements and does not require departure from GAAP or IFRS. We hope that making the algebraic structure explicit will accelerate both research at the intersection of accounting and formal methods, and the construction of ledger systems that behave less like accidents of database history and more like the mathematics they were always meant to implement.

References

1. Pacioli, L. (1494). *Summa de arithmetica, geometria, proportioni et proportionalita*. Venice: Paganino Paganini. Foundational text containing the first printed description of double-entry bookkeeping (Particularis de computis et scripturis).
2. Ellerman, D. P. (1985). The mathematics of double entry bookkeeping. *Mathematics Magazine*, 58(4), 226–233. https://www.tandfonline.com/doi/abs/10.1080/0025570X.1985.11977191
3. [2a] Ellerman, D. P. (1986). Double entry multidimensional accounting. *Omega: The International Journal of Management Science*, 14(1), 13–22.

4. Mattessich, R. (1964). *Accounting and Analytical Methods: Measurement and Projection of Income and Wealth in the Micro- and Macro-Economy*. Homewood, IL: Richard D. Irwin.
5. Ijiri, Y. (1967). *The Foundations of Accounting Measurement: A Mathematical, Economic, and Behavioral Inquiry*. Englewood Cliffs, NJ: Prentice-Hall.
6. Balzer, W., & Mattessich, R. (1991). An axiomatic basis of accounting: A structuralist reconstruction. *Theory and Decision*, 31(2–3), 213–240.
7. Mir, F., & Khan, M. (2018). Accounting games: Using matrix algebra in creating the accounting models. *Mathematics*, 6(9), 152. https://www.mdpi.com/2227-7390/6/9/152
8. Fowler, M. (2005). Event sourcing. *martinfowler.com*. https://martinfowler.com/eaaDev/EventSourcing.html
9. Young, G. (2010). CQRS documents (incl. CQRS and event sourcing). *CodeBetter* / self-published. https://cqrs.wordpress.com/documents/
10. McCarthy, W. E. (1982). The REA accounting model: A generalized framework for accounting systems in a shared data environment. *The Accounting Review*, 57(3), 554–578.
11. Merkle, R. C. (1987). A digital signature based on a conventional encryption function. In C. Pomerance (Ed.), *Advances in Cryptology — CRYPTO '87* (pp. 369–378). Springer.
12. Haber, S., & Stornetta, W. S. (1991). How to time-stamp a digital document. *Journal of Cryptology*, 3(2), 99–111. https://link.springer.com/article/10.1007/BF00196791
13. Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf
14. Kleppmann, M. (2017). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. Sebastopol, CA: O'Reilly Media.
15. Kleppmann, M. (2015). Turning the database inside-out with Apache Samza. *Confluent / martin.kleppmann.com*. https://martin.kleppmann.com/2015/03/04/turning-the-database-inside-out.html
16. DoltHub (2021–). Dolt: Git for data. *GitHub*. https://github.com/dolthub/dolt
17. TerminusDB (2018–). TerminusDB: the collaborative revision-control database. https://terminusdb.com
18. Pachyderm (2016–). Pachyderm: data-centric pipelines and versioning. https://www.pachyderm.com
19. Grigg, I. (2005). Triple entry accounting. *iang.org*. https://iang.org/papers/triple_entry.html

20. Dai, J., & Vasarhelyi, M. A. (2017). Toward blockchain-based accounting and assurance. *Journal of Information Systems*, 31(3), 5–21. DOI: 10.2308/isys-51804.

21. Coyne, J. G., & McMickle, P. L. (2017). Can blockchains serve an accounting purpose? *Journal of Emerging Technologies in Accounting*, 14(2), 101–111.

22. U.S. Securities and Exchange Commission. (2022, amended). Rule 17a-4 under the Securities Exchange Act of 1934: Records to be preserved by certain exchange members, brokers and dealers. https://www.sec.gov/rules-regulations

23. Microsoft Learn. (2024). SEC Rule 17a-4, FINRA 4511, & CFTC 1.31 compliance guidance. https://learn.microsoft.com/en-us/compliance/regulatory/offering-sec-docs

24. Raasveldt, M., & Mühleisen, H. (2019). DuckDB: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)* (pp. 1981–1984). https://duckdb.org

---

Rafi Seddiqi | Correspondence: rafi@k3labs.ai | © 2026 K³Labs. This working paper may be cited and distributed freely.