MODERN
TREASURY

How to Scale a Ledger

1

Why use a ledger database?

As financial services increasingly become embedded into mobile and web apps, more companies are facing the challenge of storing, transferring, and converting money. Typically, companies start by tracking money within their domain objects: the price of a ride share is a property of the ride; the monthly payout to a gym is a sum across bookings; the amount left to pay on a loan is stored on the loan object.

This approach tends to break down at scale:

- Companies need more sophisticated financial reporting as they grow. Every movement of money must be immutably recorded, keeping track of the source and destination of each dollar.
- Stitching together a clean history of financial events becomes impossible as data models become more complex, companies migrate to service-oriented architectures, or multiple fintech SaaS products that aren't natively integrated are introduced.
- Performance degrades as the number of users increases. A payout cron job starts missing a bank-imposed end-of-day deadline. Credit card authorization decisions take longer than the card network allows.
- Failure cases (we overcharged, a bank account debit was rejected due to insufficient funds, a transaction was fraudulent) are more frequent and difficult to reverse.

All of these problems can be solved with a ledger database.

At its core, a ledger database is a simple data model—Accounts, Entries, and Transactions. A common refrain from engineers is: "a ledger is simple to build, but the business logic on top of a ledger is complex." In fact, a well-designed and scalable core ledger database can simplify business logic and reduce time to market.

A scalable ledger database provides these guarantees:

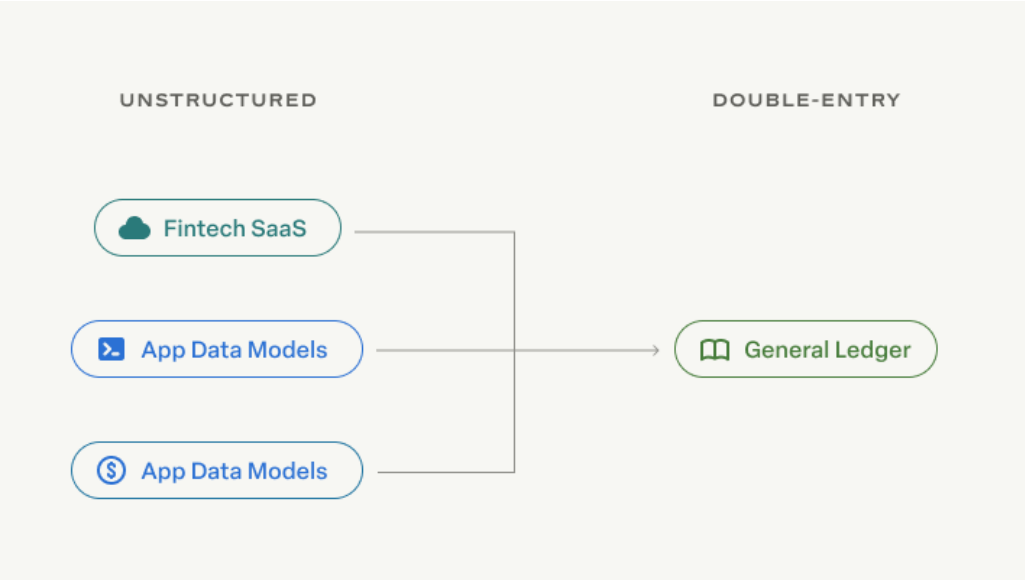
1. **Immutability:** every change to the ledger is recorded such that past states of the ledger can always be retrieved.

2. **Double-entry:** the API enforces that money cannot be moved without specifying the source and destination of funds, following double-entry accounting principles.
3. **Concurrency controls:** money can't be double-spent, even when transactions are written out of order or in parallel.
4. **Efficient aggregations:** it's fast to compute aggregations of financial events, such as the sum of all events in a particular time frame.

In this whitepaper, we aim to show why any company that moves money at scale should invest in a ledger database. We'll explain how a ledger fits into the modern fintech stack, what features it unlocks for customers, both external and internal, and finally, dive deep into Modern Treasury's Ledgers API, discussing its design philosophy and the architecture that enables it to provide its guarantees at scale.

The fintech stack

Product engineers speak the language of their domain models—orders, rides, and bookings. As their products scale, these engineers start hearing another language from the finance team—credits, debits, assets, and liabilities. Invariably, engineers get assigned the dreaded task of translating our domain models into double-entry accounting.



This is no easy feat. Financial events:

1. **Come from disparate sources.** The app may rely on multiple fintech SaaS solutions (like issuer processors, card processors, and ACH processors). App data models may be spread across multiple databases and be owned by different teams of engineers.

Corporate bank accounts contain fungible cash—attributing each dollar to its source is not possible.

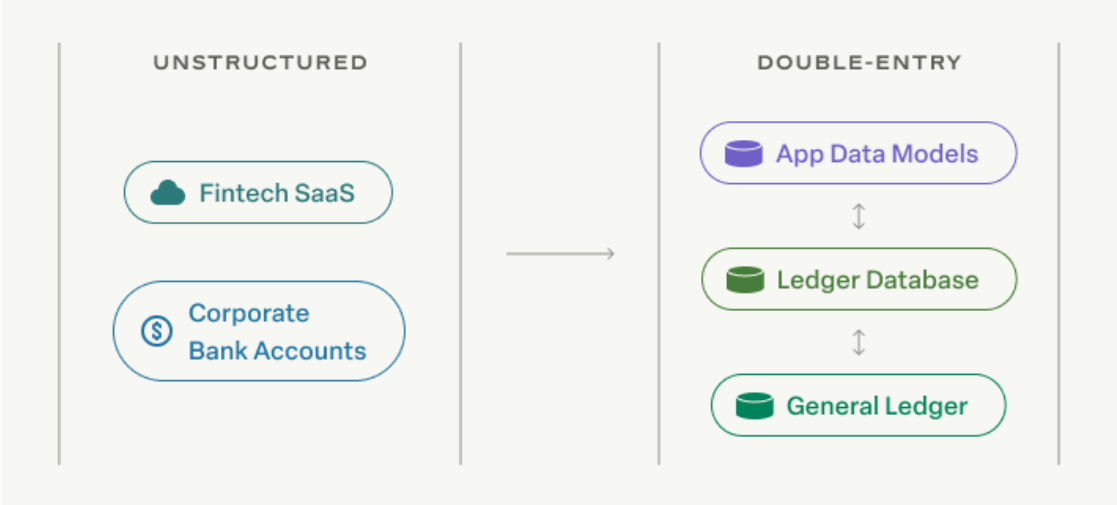
2. **Speak different languages.** Every fintech SaaS product has its own set of APIs that require separate integrations. App data models may be a mix of new and legacy systems, and documentation is likely scarce. As we know well at Modern Treasury, banks send transaction information in a wide variety of arcane formats that are difficult to parse.
3. **Are generated from mutable underlying records.** None of the sources of financial events have a rock-solid, consistent way to reconstruct why a pool of money was a certain value at a certain time.

Many companies that move money as part of their products take the translation approach, however. Finance teams learn to accept that reconciliation can only happen accurately at the aggregate level. Attributing money movement back to the source application event is always a manual task, and in some cases, isn't possible.

A product leader at a large consumer payments company recently told us that every month, the finance team would notice that a few million dollars had gone missing in their ledger, kicking off a mad scramble to figure out where the money had gone. Every engineer that has built money moving software has a similar story. In these stories, it's not the funds that go missing—they're still sitting in a bank account. It's the attribution that's lost—the database record of who they actually belong to.

Even worse, these ledger errors ultimately become customer-facing. A small business on a marketplace platform will ask questions when a sale they expected to appear in a payout isn't there. A digital wallet customer will churn if they send money to another customer, but it never arrives. Regulators will ask hard questions if a loan customer initiates an on-time payment, but your system marks them as past due because the payment was recorded late.

The solution is to be double-entry and immutable at the source of **financial events.**



Just like a unique index in a relational database gives engineers confidence, a ledger database that enforces double-entry rules makes it not just difficult for money to go missing, but architecturally impossible.

Why doesn't every company that moves money use a double-entry ledger to power their products? Because building a ledger that both follows double-entry rules and is performant enough to run live apps is a difficult engineering problem. The most successful fintech companies have taken years and dozens of senior engineers to migrate their single-entry systems to double-entry. Only recently has it been possible for companies to get the benefits of double-entry accounting without a massive up-front build and costly maintenance by using purpose-built ledger databases like Modern Treasury Ledgers.

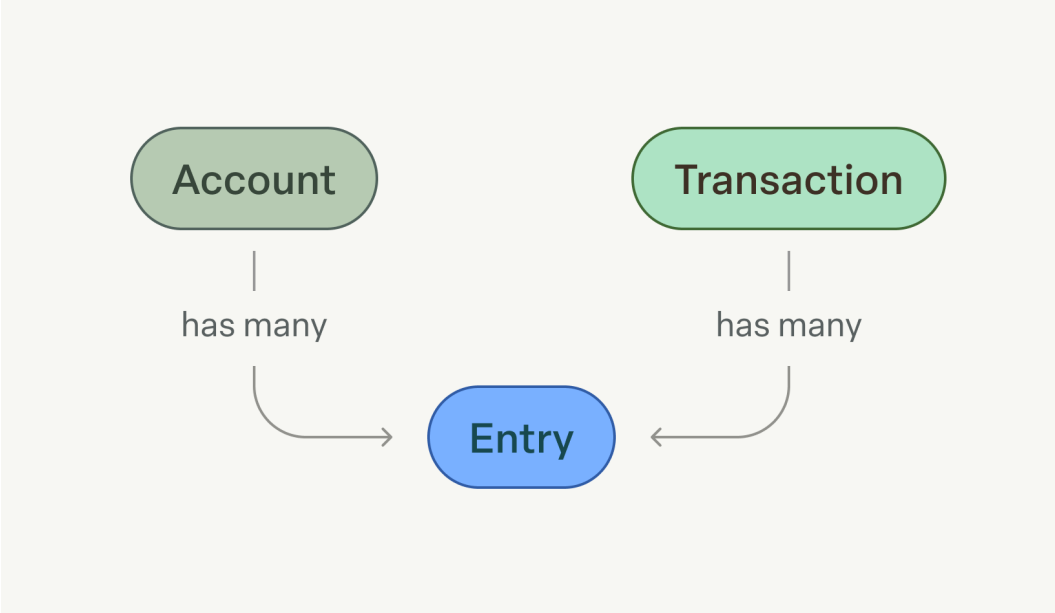
2

Adding double-entry to financial events

What does it take to be immutable and double-entry at the source of financial events? At the most basic level, it means:

1. Logging money movement in a consistent double-entry data model every time it happens.
2. Reading from this data model every time money amounts are shown to internal and external customers.

All financial events—from credit card authorizations to crypto on-ramps—can be described with three core objects:



We'll describe each of these models, how some common financial events map into them, and how they are implemented in the Ledgers API. This section focuses on the core fields and won't describe every feature available in the Ledgers API. See the documentation for a complete reference.

Account

Account

Field	Type	Description
id	uuid	A unique identifier for the account
normal_balance	string	One of credit or debit
posted_balance	Balance	Money that has settled in an Account
pending_balance	Balance	Money that's expected to settle in an Account, inclusive of money that's already settled.
available_balance	Balance	Money that's available to send out of an Account

Balance

Field	Type	Description
amount	integer	The amount in the Account, calculated from credits and debits depending on the Account's normal balance
currency	string	The currency of the balance
currency_exponent	integer	The maximum allowed decimal places for this balance

An Account tracks a sum of money, denominated in a currency. Examples of Accounts include a digital wallet, a company's operating cash, and loan principal.
Money movement between Accounts in a ledger happens instantly. In the real world, however, moving money between bank accounts is always asynchronous. For example, money sent via ACH takes at least a day to show up in a recipient's bank account. Because real money movement can't happen instantly, Accounts should be able to report a few different balances:

- Posted balance: the money that has fully settled in an Account.
- Pending balance: the money that's expected to settle in or out of an Account plus the money that's already settled.
- Available balance: the money that's available to send out of an Account. This balance subtracts money that's expected to leave an Account, but does not include money that's expected to settle in an Account.

Not all ledgers model posted, pending, and available balances—those separate balances can be modeled as separate Accounts. We believe that application ledgers should have these concepts built into the core data model, however. You can't model money movement without them, so they should be simple to use.

Let's follow an example of a credit card to see how these balances are affected by different actions in the Ledgers API.

Event ID	Event	Posted Balance	Pending Balance	Available Balance
1	A credit card starts with a $100 credit limit on the account.	$100	$100	$100
2	A card is swiped to purchase a $10 pizza. This purchase starts out pending.	$100	$90	$90
3	That night, the purchase is settled.	$90	$90	$90
4	The card holder initiates a $10 card payment from their bank account.	$90	$100	$90
5	The card payment from the bank account completes.	$100	$100	$100
6	A hotel places a $50 hold on the card at the beginning of a stay	$100	$50	$50
7	The $50 hold is removed at the end of the stay.	$100	$100	$100

We report these three balances separately so that the customer knows how much can be spent from an Account at any given time. Depending on the use case and risk tolerance, an app may choose to use each balance for a different purpose. For example, a customer may be limited by their available balance for transfers out of their Account, but their past-due status for a loan may be optimistically determined by their pending balance.

Entry

Field	Type	Description
id	uuid	A unique identifier for the account
normal_balance	string	One of credit or debit
posted_balance	Balance	Money that has settled in an Account
pending_balance	Balance	Money that's expected to settle in an Account, inclusive of money that's already settled.
available_balance	Balance	Money that's available to send out of an Account

Account balances are never directly modified. Instead, balances change as Entries are written to the Account. Entries are an immutable log of balance changes on an Account. All fields on the Entry are immutable, except for discarded_at, which is set when an Entry is replaced by a later Entry.

Let's follow the same credit card example to see how each action can be represented as an Entry.

1. A credit card starts with a $100 credit limit on the account.

Create New Entry

Field	Value
id	entry_0
account_id	card_account_id
status	posted
direction	credit
amount	10000
discarded_at	null

2. A card is swiped to purchase a $10 pizza. This purchase starts out pending on the card account.

Create New Entry

Field	Value
id	entry_1
account_id	card_account_id
status	pending
direction	debit
amount	1000
discarded_at	null

3. That night, the purchase settles.

Discard entry_1

Field	Value
id	entry_1
account_id	card_account_id
status	pending
direction	debit
amount	1000
discarded_at	2022-08-26 T20:47:11+ 00:00

Replace with new Entry

Field	Value
id	entry_2
account_id	card_account_id
status	posted
direction	debit
amount	1000
discarded_at	null

3. The card holder initiates a $10 card payment from their bank account.

Create New Entry

Field	Value
id	entry_3
account_id	card_account_id
status	pending
direction	credit
amount	1000
discarded_at	null

4. The card payment from the bank account completes.

Discard entry_3

Field	Value
id	entry_3
account_id	card_account_id
status	pending
direction	credit
amount	1000
discarded_at	2022-08-26 T20:47:11+ 00:00

Replace with new Entry

Field	Value
id	entry_4
account_id	card_account_id
status	posted
direction	credit
amount	1000
discarded_at	null

5. A hotel places a $50 hold on the card at the beginning of a stay.

Create New Entry

Field	Value
id	entry_5
account_id	card_account_id
status	pending
direction	debit
amount	5000
discarded_at	null

6. The $50 hold is removed at the end of the stay.

Discard entry_5

Field	Value
id	entry_5
account_id	card_account_id
status	pending
direction	debit
amount	5000
discarded_at	2022-08-26 T20:47:11+ 00:00

Replace with new Entry

Field	Value
id	entry_6
account_id	card_account_id
status	archived
direction	debit
amount	1000
discarded_at	null

Discarding Entries

As we showed through the above example, Entries have a mutable field discarded_at that gets set when they get replaced by a newer Entry. Only pending Entries can be replaced; posted and archived Entries are never modified. Pending Entries get replaced in the following two circumstances:

1. The pending amount of an Entry needs to change.
2. The state of the Entry needs to change.

Why introduce this mutability in the API? To see why, it helps to consider the alternative. Instead of discarding Entries, we could create a reversal Entry that undoes the original Entry. In this model, moving an Entry from pending to posted would create two Entries instead of just one. Step 5 above would instead be:

Reverse entry_3

Field	Value
id	entry_4
account_id	card_account_id
status	pending
direction	debit
amount	1000

Create new Entry

Field	Value
id	entry_5
account_id	card_account_id
status	posted
direction	credit
amount	1000

While valid, this approach leads to a messy Account history. We can't distinguish between debits that were client-initiated and debits that are generated by the ledger as reversals. Listing Entries on an Account doesn't match our intuition for what Entries actually comprise the current balance of the Account.

Discarding solves this problem by making it easy to see the current Entries (simply exclude any Entries that have discarded_at set), while also not losing any history.

Computing Account balances

Now that all balance changes are logged as Entries, how do we compute Account balances? Here's where the normal_balance field on Account comes into play. Every Account in a ledger is categorized as debit normal or credit normal. Definitionally, Accounts that represent uses of funds (assets, expenses) are debit normal, and sources of funds (liabilities, equity, revenue) are credit normal.

The balances of credit normal Accounts are increased by credit Entries and decreased by debit Entries; debit normal Account balances are increased by debit Entries and decreased by credit Entries.

Why bother with debits, credits, and Account normality at all? Many ledgers try to avoid complications by using negative numbers to represent debits and positive numbers to represent credits. At first glance, this appears to align better with engineers' intuitions.

For Modern Treasury's Ledgers API, we chose to include the credits and debits concepts because, without them, double-entry accounting is messy. Consider a simple flow where a user deposits money into a digital wallet. This flow will affect the company's cash Account and the user's wallet Account. Our intuition is that the cash Account will increase (the company got cash from the user), and also the wallet Account will increase (the user now has a positive balance in the digital wallet).

With a positive/negative number approach, both Accounts increasing is not possible. We have to pick one Account to be negative (so that no money is created or destroyed), and it's not clear which one.

Credits and debits solve this problem. We should debit the cash Account, whose balance increases because it is debit normal. And we should credit the user's wallet Account, whose balance also increases because it is credit normal.
This digital wallet scenario is just one example. For a full primer on double-entry accounting, check out our series on Accounting for Developers.

Here, we'll focus on how to actually implement a double-entry ledger.
With some simple math, each type of balance (pending, posted, and available) can be calculated from the following 5 fields:

- `posted_debits`: Sum of posted debit Entries
- `posted_credits`: Sum of posted credit Entries
- `pending_debits`: posted_debits plus the sum of non-discarded pending debit Entries

- `pending_credits`: posted_credits plus the sum of non-discarded pending credit Entries
- `normal_balance`: One of credit or debit, stored on the Account

A ledger database should be optimized to retrieve these 5 fields quickly, and only compute posted balance, pending balance, and available balance upon request.
Each balance type is then computed as follows:

Posted Balance

```
case normal_balance
when "credit"
  posted_credits - posted_debits
when "debit"
  posted_debits - posted_credits
end
```

Pending Balance

```
case normal_balance
when "credit"
  pending_credits - pending_debits
when "debit"
  pending_debits - pending_credits
end
```

Available Balance

```
case normal_balance
when "credit"
  posted_credits - pending_debits
when "debit"
  posted_debits - pending_credits
end
```

3

Transaction

Transaction

Field	Type	Description
id	uuid	A unique identifier for the Transaction
status	string	One of pending, posted, or archived
entries	Entry[]	The Entries in this Transaction

Some ledger systems stop with just Accounts and Entries. We already have a way to store an immutable history of balance changes with just the first two models. However, a robust ledger system should also have a Transaction model to enable atomic movement of money between Accounts and to enforce double-entry rules.

Consider a simple transfer of money between two digital wallets, one owned by Alice and one by Bob. If Bob sends Alice $10, we can represent that transfer as two Entries.

Field	Value
id	bob_entry
account_id	bob_account_id
status	posted
direction	debit
amount	1000
discarded_at	null

Field	Value
id	alice_entry
account_id	alice_account_id
status	posted
direction	credit
amount	1000
discarded_at	null

Imagine that bob_entry was successfully created, but alice_entry failed to create (maybe the database was having network issues). Now the ledger is in an inconsistent state—Bob was debited money, but Alice didn't get anything.
Transactions solve this consistency problem by allowing us to specify groups of Entries that either must all succeed or all fail. In order to guarantee atomicity, all the non-discarded Entries on a Transaction must share the status of the Transaction. This ensures that all Entries progress in status at the same time, all-or-nothing.

A Ledger API should only allow clients to directly create Transactions, not Entries. This limitation helps ensure clients don't run into consistency problems. That means a Ledger API must manage creating Entries itself. There are three operations to implement, corresponding to the possible states of the Transaction.

Creating a Pending Transaction

This is generally the first step in the lifecycle of a Transaction. We simply need to persist the Entries.

Field	Value
id	bob_entry_1
account_id	bob_account_id
status	pending
direction	debit
amount	1000
discarded_at	null

Field	Value
id	alice_entry_1
account_id	alice_account_id
status	pending
direction	credit
amount	1000
discarded_at	null

Posting a Pending Transaction

Since Entries are immutable, when we move a Transaction from pending to posted, we need to discard and then create new Entries.

1. Discard bob_entry_1 and alice_entry_1

Field	Value
id	bob_entry_1
account_id	bob_account_id
status	pending
direction	debit
amount	1000
discarded_at	2022-08-26T20:47:11+00:00

Field	Value
id	alice_entry_1
account_id	alice_account_id
status	pending
direction	credit
amount	1000
discarded_at	2022-08-26T20:47:11+00:00

2. Create posted Entries.

Field	Value
id	bob_entry_2
account_id	bob_account_id
status	posted
direction	debit
amount	1000
discarded_at	null

Field	Value
id	alice_entry_2
account_id	alice_account_id
status	posted
direction	credit
amount	1000
discarded_at	null

Archive a Pending Transaction

Posted Transactions are immutable, and so cannot be archived. Pending Transactions can be archived, following a similar process to posting.

1. Discard bob_entry_1 and alice_entry_1

Field	Value
id	bob_entry_1
account_id	bob_account_id
status	pending
direction	debit
amount	1000
discarded_at	2022-08-26T20:47:11+00:00

Field	Value
id	alice_entry_1
account_id	alice_account_id
status	pending
direction	credit
amount	1000
discarded_at	2022-08-26T20:47:11+00:00

2. Create archived entries.

Field	Value
id	bob_entry_2
account_id	bob_account_id
status	archived
direction	debit
amount	1000
discarded_at	null

Field	Value
id	alice_entry_2
account_id	alice_account_id
status	archived
direction	credit
amount	1000
discarded_at	null

4

Writing To The Ledger For Recording & Authorization

Now that we have introduced the basic data models that comprise a ledger, we can discuss how a ledger fits into a money movement system. There are two common use cases for a ledger:

1. A consolidated record of money movement that happens in other systems. We call this recording.
2. Approving or denying money movement. We call this authorizing.

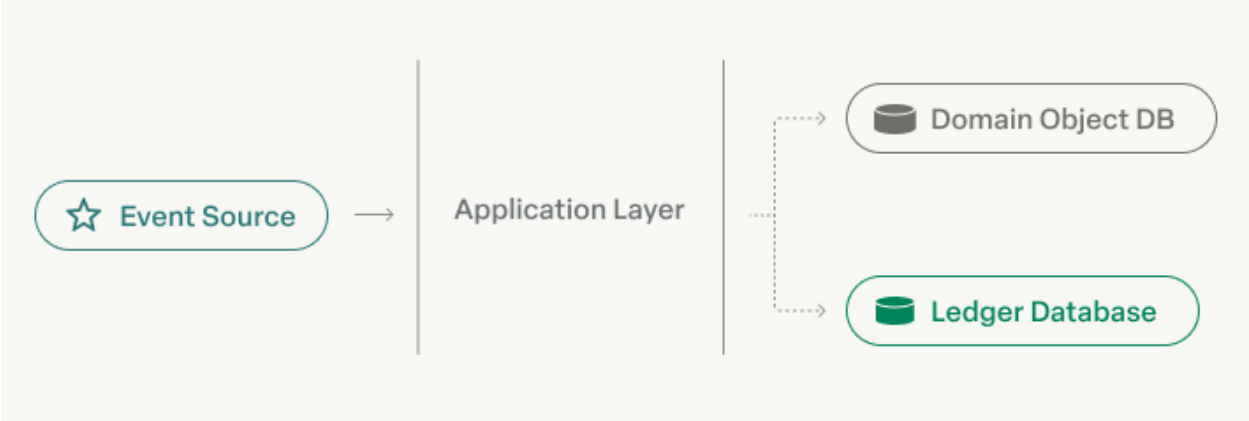
Most ledger implementations we've seen can operate at scale in only one of these use cases. As a result, when new use cases emerge, or new products are developed, companies tend to build multiple ledgers that aren't interoperable with each other. The most powerful ledgers can operate in both models, allowing clients to choose at the Entry level, which guarantees they need depending on performance and consistency requirements.

Recording

Recording is all about taking financial events that occur on other systems (like your fintech SaaS products or your bank), translating them into the core data models, and making them available for query by both internal and external customers. The ledger is an async copy of these financial events and is optimized for:

- High throughput. The possibility of 1000s of writes per second (or higher).
- Complex queries. The system is optimized for filtering and aggregating the core data models.
- Eventual consistency. Reads from the ledger may be stale for a few seconds.

The ledger should not approve or deny the money movement from a recorded Entry. This limitation prevents inconsistency between the ledger and the external system.



1. Event Source: The source of the money movement. This can be any service that moves money, for example, a bank, a card payment processor, or a payments API like Modern Treasury. For recorded Entries, the event source is the ultimate source of truth.
2. Application Layer: Your application code that processes events. It's responsible for translating the event source's data model into your domain objects and Transactions in the Ledger DB.
3. Domain Object DB: Your application data store. This will store any state from the event that doesn't relate to money movement.
4. Ledger DB: The consolidated store of money movement events.

Notice that the Domain Object DB and the Ledger DB are dotted-lined from the Application Layer, indicating that this connection can be asynchronous and eventually consistent. This eventual consistency enables high throughput. You may be asking "how can I move money confidently with eventual consistency?" The key insight is that you can keep a ledger internally consistent even if it is a delayed representation of money movement. The way to achieve this is by supporting client-supplied timestamps and Account balance versions.

Client-supplied timestamps

Since money movement already occurred by the time it's recorded by the ledger, clients need a way to specify when the money movement actually happened. The Ledgers API has an additional field on the Transaction data model that's client-supplied to achieve this.

Transaction

Transaction

Field	Type	Description
effective_at	DateTime	When the Transaction occurred in an external system.

To preserve atomicity, all Entries on a Transaction inherit the Transaction's effective_at timestamp.

Entry

Entry

Field	Type	Description
effective_at	DateTime	When the corresponding Transaction occurred in an external system.

effective_at allows clients to modify historical balances, so that they reflect the balances as they were in an external system. Using effective_at, we can support querying historical balances on accounts.

Typically, the effective_at timestamp sent by the client will be close to the created_at timestamp set by the ledger. The difference between effective_at and created_at is the delay between getting information from the external system into the ledger—usually on the order of seconds or minutes.

Account balance versions

Allowing clients to modify historical balances introduces a problem—if balances in the past can change at any moment, how can we know which transactions correspond to a balance?

Consider this race condition:

1. At time t: A client reads Alice's Account balance
2. At time t+1: A Transaction transaction_1 is written to the ledger with effective time t-1
3. At time t+2: A client asks for all Transactions before T. The result will include transaction_1, but the balance read did not. The client will see an inconsistency between the Transactions and the Account balance.

The Ledgers API allows for consistent Transaction and Account balance reads through fields on Account and Entry.

Account

Account

Field	Type	Description
version	integer	This version is incremented every time an Entry is created or modified on the account.

Entry

Entry

Field	Type	Description
account_version	integer	The version of the Account associated with this Entry.

With these new fields, we can know exactly which Entries correspond with a given Account version. After reading the Account `posted_balance` at the effective time

`2022-08-26T20:47:11+00:00`, if the Account version is 10, the Entries that correspond to that balance can be found by filtering Entries by:

- Account ID
- Status of posted
- `effective_at` less than or equal to `2022-08-26T20:47:11+00:00`
- Account version less than or equal to 10
- Not discarded

Use cases

These common use cases work well with recording:

- **Displaying account details:** account UIs for digital wallets, cards, brokerages, and other account-like products typically have a UI where the balance on an Account is displayed along with recent Entries. Using the Account version and Entry account_version fields, we can ensure that the displayed balance and Entries are always consistent.
- **Payouts:** Marketplaces collect money on behalf of their customers, and pay that money out on a certain cadence (daily, weekly, monthly). These systems must tolerate a few seconds of staleness, because actual money movement is happening through payment processors outside of the ledger. Using effective_at, the ledger ensures that Transactions are in the correct order, even if they are recorded out of order. Account versions enable displaying to a user exactly which Entries correspond to a particular payout.
- **Loan servicing:** Systems that service loans are complicated, but generally work asynchronously. Accruing interest is very similar to a marketplace payout—take a snapshot of a past-due balance, get the Account version, and accrue interest based on the Entries that comprise that balance. Applying a payment to a balance benefits from client-supplied timestamps—these systems should compute past due status based on when a payment was processed, not when the lending ledger recorded the payment.
- **Crypto:** By definition, crypto transactions happen outside of your application's ledger on a blockchain. Often it makes sense to keep a local copy of those transactions in a ledger database for performance reasons. Because the transactions have already happened by the time they are written to the application ledger, they should be recorded with an `effective_at` timestamp matching when the transaction happened on the blockchain.

Authorizing

Authorizing is about approving or denying Transactions based on the state of the associated Accounts. It's characterized by:

- Read-after-write consistency. Updates from Entries are instantly applied to the associated Accounts.
- Lower Transaction throughput. Performance will degrade around 100 Entries per second on individual Accounts.
- Assertions on balances. The system is optimized to maintain invariants on Account balances or versions.

Enforcing approval rules on each Transaction requires a strict ordering of Transactions, which also means that Transactions can only be processed one at a time. This limitation results in higher latencies because Transaction recording can no longer be done in parallel on individual Accounts.

There are two types of rules that the Ledgers API supports, one based on Account versions and another based on Account balance.

Version locking

The simplest approval rule that we can enforce while writing a Transaction is on Account versions. Since the Account version is updated every time an Entry is written to the Account, we can ensure that we are the next Entry written by having the client send an Account version along with the request to create a Transaction. The ledger will reject the write if the current Account version in the database is different from the version sent by the client.

This approval rule is similar to the concept of optimistic locking. We can enforce the rule at the database level using transactions following this algorithm:

1. Start a database transaction
2. Write the ledger entry
3. Update the Account version, including a condition on the current Account version
4. If the Account was updated in 3 (the client-provided version matched the one in the database), then commit the database transaction. Otherwise, roll it back.

Balance locking

Version locking is susceptible to “hot accounts,” what we call Accounts that see a high volume of writes. If the Account version is incrementing at a fast enough pace, some Transactions will never be able to commit. To solve this problem, it helps to step back and think about the main use case for locking in the first place. In almost all cases, clients want version locking in order to enforce balance assertions. For example, the client might want to create a pending Transaction for a card authorization only if there is enough available balance on the card.

To address the most common use case for locking, the Ledgers API supports balance locking at the Entry level. To implement this, we add a few new fields to the Entry data model.

Entry

Entry

Field	Type	Description
pending_balance_amount	BalanceCondition	Conditions that must be true on the Account’s pending balance after the Transaction commits

BalanceCondition

BalanceCondition

Field	Type	Description
lt	integer	Amount that the balance must be strictly less than
lte	integer	Amount that the balance must be less than or equal to
eq	integer	Amount that the balance must be equal to
gte	integer	Amount that the balance must be greater than or equal to
gt	integer	Amount that the balance must be greater than

These new fields allow us to implement balance assertions when writing Transactions. Consider an Account with $100 available balance, with two Transactions written simultaneously on the account, one for $25 and one for $75. With version locking, the flow would be:

$25 Transaction

1. Read Account balance and version, and check that balance is greater than or equal to $25
2. Write Transaction that includes an Entry on the Account with the version read in the previous step.

$75 Transaction

1. Read Account balance and version, and check that balance is greater than or equal to $75
2. Write a Transaction on the Account, which fails because the $25 Transaction was written slightly before
3. Again read Account balance and version
4. Write Transaction

In total, we made a call out to the ledger 6 times. We can do the same in 2 calls with balance locking:

$25 Transaction
Write Transaction with `gte: 0` on the relevant Entry.

$75 Transaction
Write Transaction with `gte: 0` on the relevant Entry.

Not only are we calling the ledger fewer times, but also the API better matches our intent to prevent a certain Account balance from going negative.

Implementing balance locking so that race conditions are handled properly and so that all possible combinations of locks on different balances are respected is beyond the scope of this paper. The logic is based on how version locking works—within a database transaction, attempt to write each Entry finding an Account with the required balance, and then check whether the update actually went through, rolling back the database transaction if it did not.

Use cases

These common use cases require authorizing:

- **Digital wallets:** A digital wallet holds a sum of money for a user that can be withdrawn into a bank account (typically by initiating an ACH credit). It's important that digital wallets ensure that users have sufficient funds to initiate a withdrawal, which necessitates balance locking. Additionally, many digital wallets support closed-loop payments between users. Authorizing Entries enable such instant payments while making sure Account balances can't go below 0.
- **Card authorizations:** In order to respond to a card authorization request, a ledger must be able to know the exact available balance of an Account at a point in time. Balance locking enables this, while also preventing double-spending. Implementations can reserve the authorized amount until it is cleared using pending Transactions.

Automatically recording or authorizing

Most ledgers we've worked with in the past implement only one of the recording or authorizing models. Because the guarantees and use-cases for each model are so different, it's easy to see why. As companies add new products that need recording or authorizing ledgers, they build new ledgers that are optimized for each product. The downside of this approach is that a company needs to staff a crew of engineers who are experts in building ledgers on each product team. It's an expensive strategy.

We've worked with some ledger implementations that get closer to being general purpose by designating certain Accounts as exclusively recording or authorizing. Some ledger implementations even allow different modes to be toggled by on-call engineers—if an Account is experiencing high Transaction throughput, an on-call engineer can put the Account into recording mode.

What we've realized by working with companies using ledgers for many use cases is that even an Account settings implementation is not sufficient. For the Ledgers API to be truly multi-purpose and multi-product, the model must be determined at the Entry level. Here are two examples where the same Account needs to both authorize and record depending on the situation.

- **Digital wallets:** We've already covered why digital wallets need authorizing for withdrawals and closed-loop payments. Some Transactions on digital wallets need to be executed regardless of Account state, however. One example is pay-ins—a user adding money to their wallet from a funding source should be simply recorded. Another is financial charges / credits. If the digital wallet is an interest-bearing account, interest should be added to the balance regardless of the Account state. On the other side, if the Account contains a loan that's accruing interest, the interest charges should be deducted from the balance even if it would make the Account go negative.
- **Card accounts:** Ledgering the many possible events from a card network for an Account that tracks a credit or debit card balance is complex and beyond the scope of this paper. But even processing the two simplest events—authorization and clearing—requires a mix of authorizing and recording. As discussed above, card authorizations require a synchronous read of the Account balance. After a successful authorization, the card network will eventually send a clearing event to indicate that the reserved funds should be marked as finalized. This event should be recorded by the ledger regardless of the Account state.

We can infer whether an Entry needs to be recorded or authorized based on whether the Entry contains an Account version or balance lock. Transactions may contain a combination of locked and not locked Entries. For example, here's a sample card authorization Transaction:

Field	Value
id	card_entry_1
account_id	card_account_id
status	pending
direction	debit
amount	1000
discarded_at	null

Field	Value
id	processor_entry_1
account_id	processor_account_id
status	pending
direction	credit
amount	1000
discarded_at	null

card_entry_1 requirements:

- Corresponds to a purchase on the card holder's Account.
- Should have a balance lock—it should not be authorized if it would bring card_account_id's balance below 0.
- Needs strong consistency. The balance lock cannot be processed without an up-to-date read of the current Account balance.
- The associated Account will not require high throughput. You can't swipe a credit card 1,000 times per second.

processor_entry_1 requirements:

- Corresponds to the money that will need to be sent to the card issuer processor as part of daily settlement. This money eventually makes its way to the issuing bank, which fronted the money for Transactions across your card program during the day.
- Should not have a balance lock. The card auth should be allowed to go through regardless of the state of the issuer processor's Account.
- Needs eventual consistency. The system only needs to read the balance of the issuer processor's Account at the end of the day during the settlement process.
- The associated Account requires high throughput. Every card authorization in your program will include a credit entry that writes to this Account. As your card program grows, this Account can easily experience 1,000's of writes per second.

The throughput and consistency requirements for each Entry are very different. Even so, the Transaction that contains both Entries must be processed all-or-nothing. As a result, scalable ledgers must be able to process Entries on the same Transaction with different consistency and throughput requirements. Recorded Entries should be processed asynchronously and in batches, but also should not be written if their containing Transaction was not authorized. Additionally, they should not be present in reads from the ledger until the containing Transaction is authorized.

This interplay between authorizing and recording Entries on the same Transaction is a key reason why double-entry accounting is such a difficult engineering problem. A single-entry ledger completely avoids this problem by processing Entries separately, at the expense of consistency. It would be easy for a system to accidentally approve a card authorization, but not include that authorization in a daily settlement with the issuer processor. With double-entry, that is not possible.

5

Ledger Guarantees: Immutability & Double-Entry

At the beginning of this white paper, we outlined four guarantees that ledger databases should give. Let's dive deeper into what these guarantees mean and how the Ledgers API is able to provide them.

Immutability

Every state of the ledger must be recorded and can be easily reconstructed.

Immutability is the most important guarantee from a ledger. You may be wondering how the ledger described in this paper can possibly be immutable given that all of the data models are mutable.

- Accounts: The balances change as Entries are written to them.
- Transactions: The state can change from pending to either posted or archived. While the Transaction is pending, the Entries can change too.
- Entries: Entries can be discarded.

These mutable fields on our core data model help the ledger match real world money movement. But ultimately, the data model is a facade that's built on top of an immutable log. This log can be queried in a few different ways to reconstruct past states.

Ledger state by time

We've already covered how effective_at allows us to see what the balance on an Account was at an earlier time. We can also see the Entries that were present on an Account at a timestamp, since no Entries are ever actually deleted. The Entries on Account account_a at effective time timestamp_a can be fetched by filtering:

- Account ID is account_a
- effective_at is less than or equal to timestamp_a
- discarded_at is null or greater than or equal to timestamp_a

This kind of query can be helpful if you are reading from the ledger from a job.

Passing in a timestamp when querying Entries can help the job return consistent results regardless of when it is scheduled.

Ledger state by version

As we've seen before, timestamps aren't sufficient to allow clients to query all past states in ledgers that are operating at scale. For one, we've allowed Entries to be written at any point in the past using the effective_at timestamp. Also, what if Entries share the same effective_at or discarded_at timestamp?

Ledgers should solve the drawbacks of timestamps by introducing versions on the main data models. Account versions that are persisted on Entries allow us to query exactly which Entries correspond to a given Account balance (see "Account balance versions" above).

Account versions aren't enough to see how Entries were grouped together in past states of a Transaction. Since the Entries on a Transaction can be modified until the Transaction is in the posted state, we need to preserve past versions of a Transaction to be able to fully reconstruct past states of the ledger. The Ledger API stores all past states of the ledger with another field on Transaction:

Field	Type	Description
version	integer	The current Transaction version

Transactions report their current version, but also past versions of the Transaction can be retrieved by specifying a version when querying.

Past Transaction versions are used when trying to reconstruct the history of a particular money movement event that involves multiple changing Accounts. Consider a bill-splitting app. A pending Transaction represents the bill when it's first created, and users can add themselves to subsequent versions of the bill before it is posted.

Version 0

Alice starts by paying for the entire bill.

Field	Value
id	transaction_1
status	pending
version	pending
entries	See below

Field	Value
id	bill_account_1
account_id	bill_account_id
status	pending
direction	credit
amount	1000
discarded_at	null

Field	Value
id	alice_entry_1
account_id	alice_account_id
status	pending
direction	debit
amount	1000
discarded_at	null

Version 1

Bob splits the bill with Alice.

Field	Value
id	transaction_1
status	pending
version	1
entries	See below

Field	Value
id	bill_entry_2
account_id	bill_account_id
status	pending
direction	credit
amount	1000
discarded_at	null

Field	Value
id	alice_entry_2
account_id	alice_account_id
status	pending
direction	debit
amount	500
discarded_at	null

Field	Value
id	bob_entry_1
account_id	bill_account_id
status	pending
direction	debit
amount	500
discarded_at	null

Version 2

The bill is posted.

Field	Value
id	transaction_1
status	posted
version	2
entries	See below

Field	Value
id	bill_entry_3
account_id	bill_account_id
status	posted
direction	credit
amount	1000
discarded_at	null

Field	Value
id	alice_entry_3
account_id	alice_account_id
status	posted
direction	debit
amount	500
discarded_at	null

Field	Value
id	bob_entry_2
account_id	bill_account_id
status	posted
direction	debit
amount	500
discarded_at	null

Audit logs

At this point, we've recorded past states of the main ledger data models, but we haven't recorded who made the changes. As ledgers scale to more product use cases, it's common for the ledger to be modified by multiple API keys and also by internal users through an admin UI. We recommend an audit log to live alongside the ledger to record that kind of information.

The Ledgers API audit logs contain these fields:

Field	Type	Description
id	uuid	A unique identifier for the Audit Log
action	string	The action was done on the entity, e.g. Create, Update, Delete
entity	Entity	The polymorphic entity acted upon
source	Source	The polymorphic source of the change
data	json	Unstructured data that describes the change, e.g. what were the fields on the entity before the change and after the change
occurred_at	timestamp	When the change occurred

Double-entry

All money movement must record the source and destination of funds.

Once a ledger is immutable, the next priority is to make sure money can't be created or destroyed out of nowhere. We do this by validating that every money movement records both the source and destination of funds. The principles of double-entry accounting helps us do this in a way that's worked for centuries. We'll focus just on the implementation details here—if you want a primer on how to use double-entry accounting, check out our guide on Accounting For Developers.

Double-entry accounting must be enforced at the application layer. When a client requests to create a Transaction, the application validates that the sum of the Entries with direction credit equals the sum of the Entries with direction debit.

This works well until a Transaction contains Entries from Accounts with different currencies. Imagine a platform that allows its users to purchase BTC, and wants to represent that purchase as an atomic Transaction. Here's one way to structure that Transaction:

Field	Value
id	platform_btc_entry_1
account_id	platform_btc
status	posted
direction	debit
amount	100000000
discarded_at	null

Field	Value
id	platform_usd_entry_1
account_id	platform_usd
status	posted
direction	credit
amount	1894890
discarded_at	null

Field	Value
id	alice_usd_entry_1
account_id	alice_usd
status	posted
direction	debit
amount	1894890
discarded_at	null

Field	Value
id	alice_btc_entry_1
account_id	alice_btc
status	posted
direction	credit
amount	100000000
discarded_at	null

This Transaction involves 4 accounts, 2 for Alice (one in USD and one in BTC, credit-normal), and 2 for the platform (one in USD and one in BTC, debit-normal). This is a valid Transaction, and the credit and debit Entries still match by the original summing method.

The original method breaks down when the Entries across currencies match, but the Transaction isn't balanced. Here's an example:

Field	Value
id	platform_btc_entry_1
account_id	platform_btc
status	posted
direction	debit
amount	100
discarded_at	null

Field	Value
id	alice_btc_entry_1
account_id	alice_btc
status	posted
direction	credit
amount	200
discarded_at	null

Field	Value
id	alice_usd_entry_1
account_id	alice_usd
status	posted
direction	debit
amount	100
discarded_at	null

Even though 100 + 100 = 200, this Transaction is crediting an extra 0.000001 BTC out of nowhere, and Alice was debited $1, which vanished. The correct validation is to group Entries by currency and validate that credits and debits for each currency match.

You might think that because $18,948.90 buys 1 BTC, we could implement currency conversion Transactions with just two entries—one for USD, and one for BTC, and they balance based on the exchange rate of USD to BTC. There are three main issues with that implementation:

1. Because currency exchange rates fluctuate over time, it would be difficult to verify that the ledger was balanced in the past.
2. There isn't a universally agreed-upon exchange rate for all currencies. It would be very difficult for the client and the ledger to agree that a Transaction is balanced.
3. Having just two Accounts doesn't match the reality of how currency conversion is performed. To allow a user to convert dollars to BTC, the platform must have a pool of BTC to give and a pool of dollars to place the user's money. That process will always involve at least four Accounts.

Concurrency controls

It's not possible to unintentionally spend money, or spend it more than once.
With a ledger that's immutable and enforces double-entry semantics, you're all set—until your app is successful and starts scaling. Concurrency controls in the Ledgers API prevent inconsistencies borne out of race conditions. Money shouldn't be unintentionally or double spent, even when Transactions are written on the same Accounts at the same time.

We've already covered how our Entry data model, with version and balance locks, prevents the unintentional spending of money (see "Authorizing" above). Nothing we've designed so far prevents accidentally double-spending money, though. Here's a common scenario:

1. A client sends a request to write a Transaction.
2. The ledger is backed up and is taking a long time to respond. The client times out.
3. The client retries the request to write a Transaction.
4. Concurrently with 3, the ledger actually finished creating the Transaction from 1, even though the client was no longer waiting for the response.
5. The ledger also creates the Transaction from 3. The end state is that the ledger moved money twice when the client only wanted to do it once.

How can the ledger detect that the client is retrying rather than actually trying to move money more than once? The solution is deduplicating requests using idempotency keys.

An idempotency key is a string sent by the client. There aren't any requirements for this string, but it is the client's responsibility to send the same string when it's retrying a previous request. Here's some simplified client code that's resilient to retries:

```
idempotency_key = generate_uuid()

response = None
while response is None or not response.is_successful():
  response = create_transaction(idempotency_key)
```

Notice that the idempotency key is generated outside of the retry loop, ensuring that the same string is sent when the client is retrying.

On the ledger side, the idempotency keys must be stored for 24 hours to handle these transient timeout errors. When it sees an idempotency key that's already stored, it returns the response from the previous request. It's the client's responsibility to ensure that no request that moves money is retried past a 24-hour period.

Efficient aggregations

Retrieving the current and historical balances of an Account is fast.

Our ledger is now immutable, balanced, and handles concurrent writes and client retries. But is it fast? You may have already noticed in the "Entry" section above that calculating Account balances is an O(n) operation, because we need to sum across n Entries. That's fine with 100s or 1,000s of Entries, but it's too slow once we have 10,000s or 100,000s of Entries. It's also too slow if we're asking the ledger to compute the balance of many Accounts at the same time.

The Ledgers API solves this problem by caching Account balances, so that fetching the balance is a constant time operation. As we covered in the section on Accounts, we can compute all balances from pending_debits, pending_credits, posted_debits, and posted_credits—so those are the numbers we need to cache. We propose two caching strategies, one for the current Account balance and one for balances at a timestamp.

Current Account balances

The current Account balance cache is for the balance that reflects all Entries that have been written to the ledger, regardless of their effective_at timestamps. This cache is used for:

- Displaying the balance on an Account when an effective_at timestamp is not supplied by the client.

- Enforcing balance locks. A correct implementation of balance locking relies on a single database row containing the most up-to-date balance.

Because this cache is used for enforcing balance locks, it needs to be updated synchronously when authorizing Entries are written to an Account. This synchronous update is at the expense of response time when creating Transactions.

So that updates to the balance aren't susceptible to stale reads, we trust the database to do the math atomically. Here's an example Entry that we'll use to show how to update the cache:

Field	Value
id	alice_entry_1
account_id	alice_account
status	posted
direction	debit
amount	100
discarded_at	null

The entry contains a posted debit of 100, so we atomically increment the pending_debit and posted_debit fields in the cache when it's written.

We also need to update the cache when a Transaction changes state from pending to posted or archived. If we archived a pending debit of 100, we'd decrease pending_debits by 100 in the cache. Similarly, if the pending debit was posted, we'd increase posted_debits by 100.

Effective time balances

We also need reads for Account balances that specify an effective_at timestamp to be fast. This caching is a much trickier problem, so many ledgers avoid it by not supporting effective_at timestamps at all, at the expense of correctness for recorded Transactions. We've found a few methods that work, and we'll cover two approaches here.

Anchoring

Assuming that approximately a constant number of Entries are written to an Account within each calendar date, we can get a constant time read of historical balances by caching the balance at the end of each date, and then applying the intra-day Entries when we read. Our cache saves anchor points for balance reads—for every date that has Entries, the cache can report the end-of-day balance in constant time.

The effective date cache should store the numbers that are needed to compute all balances.

Field	Type	Description
account	Account	The Account associated with this cache entry
effective_date	string	The date for this entry, in YYYY-MM-DD format
pending_debits	integer	The sum of all pending debits for Entries effective on or before the effective_date
pending_credits	integer	The sum of all pending credits for Entries effective on or before the effective_date
posted_debits	integer	The sum of all posted debits for Entries effective on or before the effective_date
posted_credits	integer	The sum of all posted credits for Entries effective on or before the effective_date

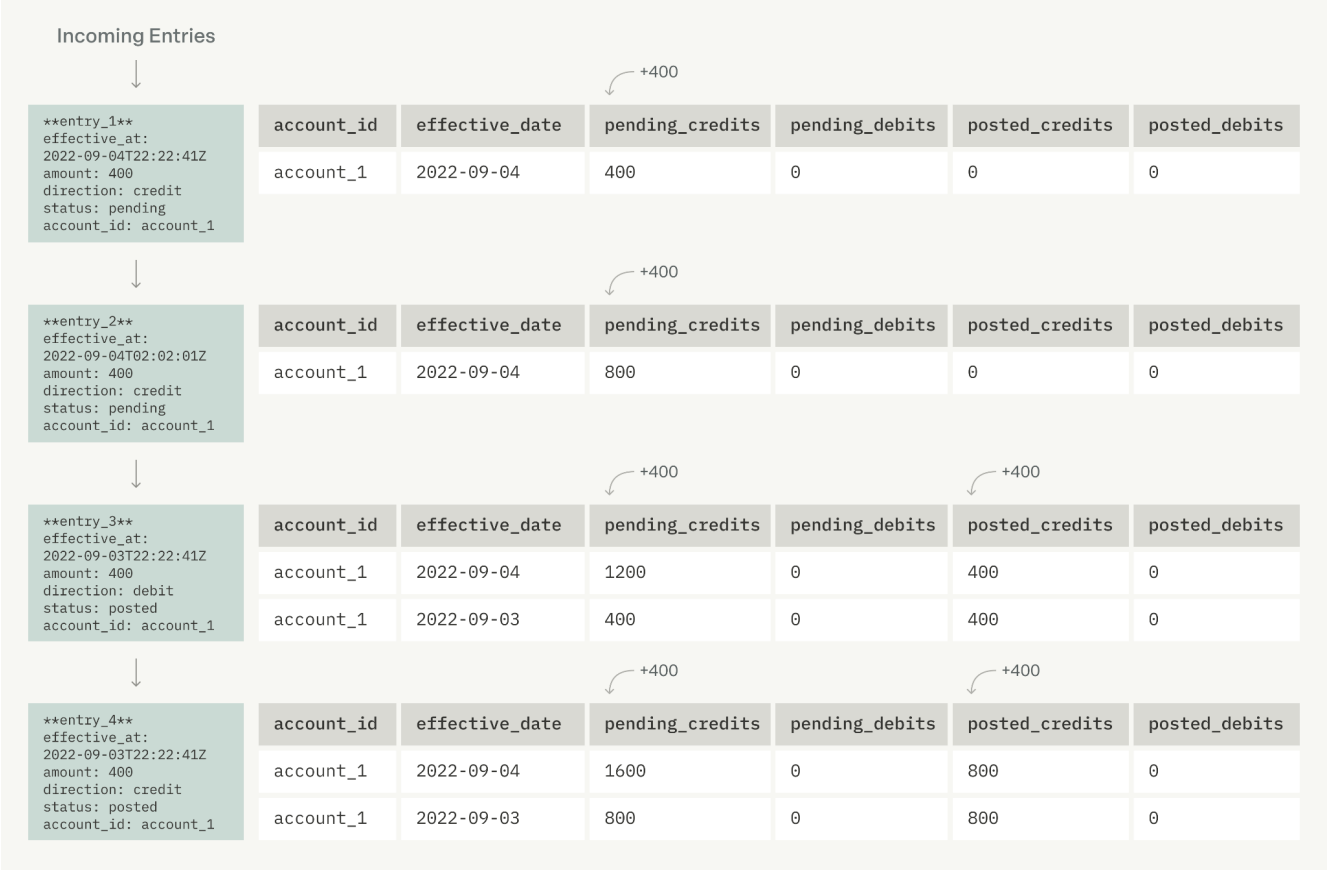
Assuming we've cached the balance for 2022-08-31, we can get the balance for `2022-09-01T22:22:41Z` with this algorithm:

1. Get the cache entry for an effective_date that's less than or equal to 2022-08-31. Store latest_effective_date as the effective date for the cache entry found.

2. Sum all Entries by direction and status for `latest_effective_date <= entry.effective_at <= '2022-09-01T22:22:41Z'`
3. Report the Account balance as the sum of step 1 and step 2.

Keeping the cache up to date is the bigger challenge. While we could update the current Account balance cache with one-row upsert, for the effective date cache we potentially need to update many rows—one for each date between the effective_at of the Transaction to be written, and the greatest effective date across all Entries written to the ledger.

Since it's no longer possible to update the cache synchronously as we write Transactions, we update it asynchronously. Each Entry is placed in a queue after the Transaction is initially written, and we update the cache to reflect each Entry in batches, grouped by Account and effective date.



Since this processing is happening asynchronously, what happens if a client requests the balance of an Account at an effective time, but recent Entries haven't yet been reflected in the cache? We can still return an up-to-date balance by applying Entries still in the queue to balances at read time. Ultimately, an effective_at balance is:

```
balance = cached_by_date_balance + intraday_entries_sum +
queued_entries_sum
```

Resulting balances

When individual Accounts have many intraday Entries, the anchoring approach won't scale. We can increase the granularity of the cache to hours, minutes, or seconds. But we can truly get constant time access to Account balances by storing a resulting balance cache. Each entry in the cache stores the resulting balance on the associated Account after the Entry was applied.

Field	Type	Description
entry_id	uuid	The ID of the associated Entry
pending_credits	integer	The resulting pending credits on the Account associated with the Entry
pending_debits	integer	The resulting pending debits on the Account associated with the Entry
posted_credits	integer	The resulting posted credits on the Account associated with the Entry
posted_debits	integer	The resulting posted debits on the Account associated with the Entry

The cache is simpler now, but updating it is more expensive. When an Entry is written with an earlier effective_at than the max effective_at for the Account, all cache entries for Entries after the new effective_at need to be updated.

A detailed discussion of how the cache is updated is beyond the scope of this paper. It's no longer possible to update every cache entry atomically in the database, so each cache entry needs to keep track of how up to date it is.

Monitoring

Reading Account balances from a cache improves performance, but it also means that balance reads are no longer from the source of truth—the Entries. It's possible for balances to diverge from Entries due to bugs. It's important to be able to quickly notice and remedy when this drift has occurred.

The Ledgers API has:

1. A regularly scheduled process that verifies that each Account's cached balances match the sum of Entries.
2. A way to automatically turn off cache reads for Accounts that have drifted.
3. A backfill process to correct the cache.
4. An on-call rotation and runbook for engineers to address cache drift problems 24/7.

6

Advanced topics

The ledger described in this paper is immutable, double-entry balanced, resilient to concurrent writes, and fast to aggregate Account balances—and still, it will only get you so far. As product requirements become more complex and the throughput required by your system increases, the architecture will start to strain.

Here are four major areas of improvement we've invested in for our Ledgers API that a truly global, scalable ledger will need:

- **Account aggregation:** For many products and reporting needs, aggregating at the Account level is not sufficient. For example, you may want to see combined Transactions and balances for all "revenue" Accounts. On the product side, you may want to implement sub-accounts—groupings of Accounts that, from the user's perspective, share a balance and Transactions. We believe the best way to model these scenarios is with a graph structure we call Account Categories. See our documentation for Ledger Account Categories.
- **Transaction search:** Building Account statements, performing marketplace payouts, and showing real-time Account summaries all require different searches of Transactions. The search may be by date ranges, require custom sorting, and efficient pagination. Additionally, you may need fuzzy search, for example allowing credit card customers to search their Transaction history for a restaurant name rather than the precise statement description. Performant search is a constantly moving target as the volume of Transactions increases and the types of queries change. It requires careful choice of architecture and ongoing index tuning, along with latency and throughput monitoring.
- **High-throughput queues:** The best fintechs in the world easily surpass 5,000 QPS on their ledgers. Depending on the problem domain, a common architecture at that scale is a queue for processing recorded Entries. This queue must be designed to accept unlimited QPS for writes.
- **Account sharding:** At a global scale, the throughput and data storage requirements of a ledger can't be accommodated by a single database. Some ledgers scale by migrating to an auto-sharding database like Google Spanner or TiDB. But more commonly, ledgers implement their own sharding strategy, typically at the Account level. Devising a strategy that can implement double-entry atomicity and can aggregate efficiently across Accounts is very challenging when Accounts aren't

colocated in the same database.

We hope this paper has given a good overview of the importance of solid ledgering and the amount of investment it takes to get right. This guide can be used to ensure your ledger meets the requirements of correctness and performance to move money confidently on behalf of your business or your customers. If you want a ledger that meets these requirements today and can scale with you, check out Modern Treasury Ledgers.